# Trifocal Tensor - report

Clément Champetier - Arthur Tourneret

07/01/2013

## Table of contents

# Part I
# Overview

**Projective geometry - The trifocal tensor** The aim of this project is to develop a program using a trifocal tensor. The idea is to do a little overview of the projective geometry, and understand this fact : it's very easy to establish relationships between multiple images for a human being, but it can be difficult for a computer.

**Initial situation** We have 3 images, and a list of correspondences (at least 7) between the 3 images. Thanks to this data and the Eigen library, we are able to compute a trifocal tensor, which can find the third point when we have 2 points (each point on a different image of course).

# Part II
# Result

## 1 Required items

### 1.1 coded and working

- working on Linux
- differents way to launch application (-h, 3 image names, 3 image names 3 and list names)
- SDL only for the interface
- create correspondences and save them in files
- load correspondences from files
- compute a tensor thanks to the correspondences
- do the transfert, whatever the 2 images clicked

### 1.2 coded and not working

### 1.3 not coded

- verify validity of lists

## 2 Non-required items

### 2.1 coded and work

- create some class to organize our code

### 2.2 coded and do not work

### 2.3 not coded

# Part III
# Mathematics section

## 3 Compute the trifocal tensor

### 3.1 understand the situation

In computer vision, the trifocal tensor is a 3*3*3 array of numbers. This object incorporates all projective geometric relationships between three views.

The situation is simple : we have three views. At first, we establish correspondences between the three images by clicking on points representing the same point in the 3d space (we need at least 7 correspondences). Then we click a pair of matched points in two views. And with the trifocal tensor, we can determine the location of the point in the third view without any further information. That's easy to do for a human being, but we must use mathematics concepts to do that with a computer.

### 3.2 understand the problem

We start by an equation given in the subject :

$$\sum_{k=1}^{3} x_p^k (x_p^{'i} x_p^{''3} T_k^{3l} - x_p^{'3} x_p^{''3} T_k^{il} - x_p^{'i} x_p^{''l} T_k^{33} + x_p^{'3} x_p^{''l} T_k^{i3}) = O^{il} \tag{1}$$

i and l vary from 1 to 2, and p from 1 to n (n = number of correspondences), which generate 4 equations per correspondence (and a total of 4n equations).

This equation can be written as a matrix product :

$$A * t = 0 \tag{2}$$

t is a vector, containing the elements of the trifocal tensor organised in one column of 27 elements. These are our unknown elements. 0 is the null vector. A is a matrix containing the coefficients corresponding to the tensor coordinates.

## 3.3 matrix A

Firstly, we need at least 7 points on each image to compute the trifocal tensor (7 correspondences). This coordinates are used to compute the coefficients in the equations. And for each correspondence, we have 4 equations (thanks to i and l), with :

- i = 1, l = 1
- i = 1, l = 2
- i = 2, l = 1
- i = 2, l = 2

To summarize, we have at least seven correspondences and 4 equations per correspondence : so 28 equations. And if we want to be able to mutiply the 2 matrix (A and t), we need 27 column on the first one. So A matrix is 28 * 27 array of numbers in the case of seven correspondences. In the general case A is a (4*n) * 27 matrix.

## 3.4 fill matrix A

Now, we have to complete the matrix A by filling every element of the matrix with the coefficient corresponding with the the coordinate of the tensor. These coefficients are presents in our equation. So the trick is to understand where put the coefficient of every coordinate of the trifocal tensor in the A matrix.

For every correspondence, we have 4 equations. Thanks to one equation, we are able to fill twelve elements of a row of our matrix because we have 4 coefficients depending on k in our equation and k varies from 1 to 3. So we knew that a lot of elements of the matrix will be at 0 (which is useful to verify our matrix).

The difficulty is to access the good element of A to fill the good case corresponding to the good tensor's coordinate. Finally, we have four nested loops to fill the A matrix. which every time fill four A elements.

$$A(4p+2i+l, 9k+6+l) = x_p^k x_p^{'i} x_p^{''3} \tag{3}$$

$$A(4p+2i+l, 9k+3i+l) = -x_p^k x_p^{'2} x_p^{''2} \tag{4}$$

$$A(4p+2i+l, 9k+6+2) = -x_p^k x_p^{'i} x_p^{''l} \tag{5}$$

$$A(4p+2i+l, 9k+3i+2) = x_p^k x_p^{'2} x_p^{''l} \tag{6}$$

Indices are shifted by one for coding.

After having the A matrix, we have to resolve the equation (2). But we do not want the trivial solution t = 0. For that we use the technique described in the subject : the SVD decomposition, found the tensor as a vector and then fill our tensor as a 3*3*3 matrix. The most appropriate way to understand this part, is to take a look to the class "TrifocalTensor" in the programming section. After this step, we have our trifocal tensor, and we can make a transfert to determinate new point on the third image (after having a pair of matched points in two views).

## 4 The transfert

Now we know the tensor t corresponding to our 3 images. We will be able to do a transfer, i.e. by choosing two points matching on two images, find the one corresponding on the third image. Let's do it. We have the coordinates of the tensor and the coordinates of two points, so we can in the same equation than one of the beginning isolate the coordinates of the third points.

$$x^{''l} x^k(-x^{'i} T_k^{33} + x^{'3} T_k^{i3}) + x^{''3} x^k(x^{'i} T_k^{3l} - x^{'3} T_k^{il}) = O^{il} \tag{7}$$

We have 4 equations and our three unknown coordinates. So we are able with that to find a new equation B*x=0 where x is the point we are searching and B the matrix with coefficients corresponding to each coordinate of x. B is a 4*3 matrix and x a vector of three coordinates. We use again the SVD to find the solution, the coordinates of x. The result was always around the couple ( 0.6, 0.7) for x and y coordinates of the point what wasn't good. This error came maybe from the fact that we didn't consider the third coordinate of x equal to 1 and so we had 4 equations and 3 unknowns instead of 2.

Because our first idea didn't work, we reorganized the equations under the form Bx = b where B is a 4*2 matrix, x a vector of two coordinates (those we search) and b a vector of 4 coordinates. This time, we considered that the third coordinate of the point searched was 1. So we have :

$$x^{''l} x^k(-x^{'i} T_k^{33} + x^{'3} T_k^{i3}) = -x^k(x^{'i} T_k^{3l} - x^{'3} T_k^{il}) \tag{8}$$

and we resolve the system with svd.solve() from eigen wich returns the vector x solution in Bx=b. This time the answer was less aberrant but not good either, the couple was around (450, 400). This error came from the fact that we didn't subtract the width of the images for the coordinates of the points p1 and p2. So the coordinates of the point were those in the sdl window and not in the image itself. Once this was corrected, the transfer worked perfectly. Success.

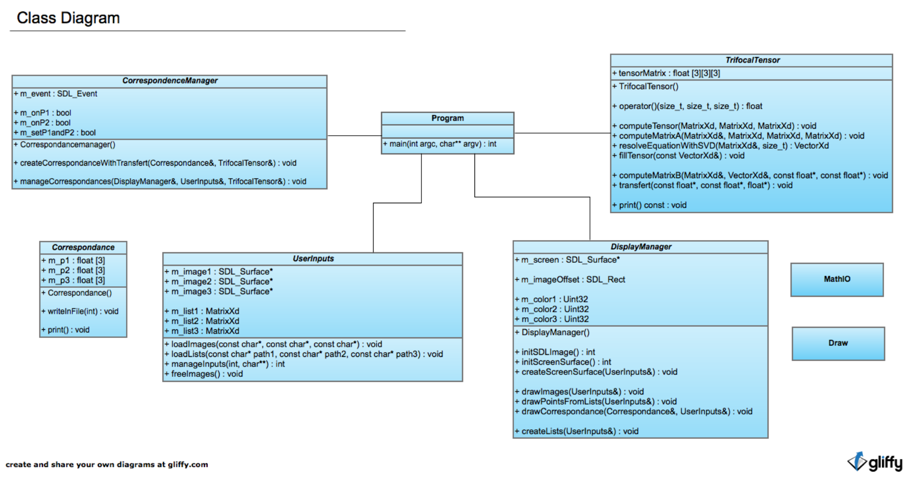
After that, we have to improve the program to manage different cases. So we made in sort that the order in which we clicked on the image for doing the transfert doesn't matter anymore. Before we were obliged to click on the first image, then the second. By managing the cases, in every order it worked.

For that, instead of doing three different tensors and transferts, we decide to send the arguments in a different order to our functions depending of the order images were clicked (compute of the tensor and transferts changed according to the case).

# Part IV
# Programming section

## 5 Class Diagram

We organized the program by created new class. Our idea was to separate the different steps in different objects : each object has his own responsibilities.


Class Diagram
CorrespondenceManager
+ m_event : SDL_Event
+ m_onP1 : bool
+ m_onP2 : bool
+ m_setP1andP2 : bool
+ Correspondancemanager()
+ createCorrespondanceWithTransfert(Correspondance&, TrifocalTensor&) : void
+ manageCorrespondances(DisplayManager&, UserInputs&, TrifocalTensor&) : void
Program
+ main(int argc, char** argv) : int
TrifocalTensor
+ tensorMatrix : float [3][3][3]
+ TrifocalTensor()
+ operator()(size_t, size_t, size_t) : float
+ computeTensor(MatrixXd, MatrixXd, MatrixXd) : void
+ computeMatrixA(MatrixXd&, MatrixXd, MatrixXd, MatrixXd) : void
+ resolveEquationWithSVD(MatrixXd&, size_t) : VectorXd
+ fillTensor(const VectorXd&) : void
+ computeMatrixB(MatrixXd&, VectorXd&, const float*, const float*) : void
+ transfert(const float*, const float*, float*) : void
+ print() const : void
Correspondance
+ m_p1 : float [3]
+ m_p2 : float [3]
+ m_p3 : float [3]
+ Correspondance()
+ writeInFile(int) : void
+ print() : void
UserInputs
+ m_image1 : SDL_Surface*
+ m_image2 : SDL_Surface*
+ m_image3 : SDL_Surface*
+ m_list1 : MatrixXd
+ m_list2 : MatrixXd
+ m_list3 : MatrixXd
+ loadImages(const char*, const char*, const char*) : void
+ loadLists(const char* path1, const char* path2, const char* path3) : void
+ manageInputs(int, char**) : int
+ freeImages() : void
DisplayManager
+ m_screen : SDL_Surface*
+ m_imageOffset : SDL_Rect
+ m_color1 : Uint32
+ m_color2 : Uint32
+ m_color3 : Uint32
+ DisplayManager()
+ initSDLImage() : int
+ initScreenSurface() : int
+ createScreenSurface(UserInputs&) : void
+ drawImages(UserInputs&) : void
+ drawPointsFromLists(UserInputs&) : void
+ drawCorrespondance(Correspondance&, UserInputs&) : void
+ createLists(UserInputs&) : void
MathIO
Draw
create and share your own diagrams at gliffy.com
gliffy

### 5.1 class UserInputs

Contains all data potentially given by the user : the images and the lists of correspondences. This class only load the images and the lists from specific files.

### 5.2 class TrifocalTensor

Contains the tensor, a matrix 3*3*3 and all function we need in order to fill this matrix, using Eigen library (and class MathIO). This class contains the transfer too because it is the tensor which allows to do it.

### 5.3 class DisplayManager

This class manages the display during all the program. This means initialize SDL, create the Screen Surface, and draw the 3 images, and all the correspondences (using class Draw).

### 5.4 class Correspondence

This class represents 3 points, one on each image. This is just in order to have more visibility on the class CorrespondenceManager.

### 5.5 class CorrespondenceManager

Contains the SDL Event (we are interested by the user clic to make correspondences). This class manage the events loop : actions after a user clic.

## 6 Algorithm choices

### 6.1 filling of the A matrix used to compute the tensor

```
initialise A to 0
    for every correspondence p
            for every line of the tensor i
                    for every column of the tensor l
                            for every depth of the tensor k
```

$$A(4p+2i+l, 9k+6+l) = x_p^k {x'}_p^i {x''}_p^3 \tag{9}$$

$$A(4p+2i+l, 9k+3i+l) = -x_p^k {x'}_p^2 {x''}_p^2 \tag{10}$$

$$A(4p+2i+l, 9k+6+2) = -x_p^k {x'}_p^i {x''}_p^l \tag{11}$$

$$A(4p+2i+l, 9k+3i+2) = x_p^k {x'}_p^2 {x''}_p^l \tag{12}$$

### 6.2 find the tensor with SVD decomposition

```
svd(A)
    Matrix V = svd.matrixV()
    Vector tensor = lastColumnOf(V)
```

### 6.3 compute matrix B in Bx = b

```
Matrix B(4,2)
    Vector b(4,1)
    initialise B to 0
    initialise b to 0
            for every line of the tensor i
```

for every column of the tensor l

$$B(2i+l,l) = \sum_{k=1}^{3} x^k(-x'^i T_k^{22} + x'^3 T_k^{i3}) \tag{13}$$

$$b(2i+l) = -\sum_{k=1}^{3} x^k(x'^i T_k^{3l} - x'^3 T_k^{il}) \tag{14}$$

### 6.4 the transfert

svd(B)
searchedPoint = svd.solve(b);

$$x''^1 = searchedPoint[0] \tag{15}$$

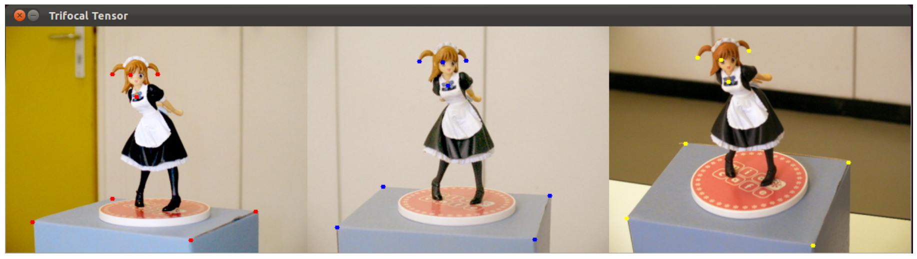
$$x''^2 = searchedPoint[1] \tag{16}$$

$$x''^3 = 1 \tag{17}$$

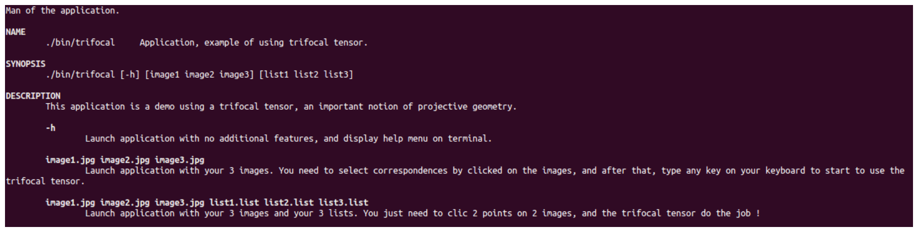
# Part V
# Our application : what we've got

## 7 The images given by the teacher

**./bin/trifocal** The trifocal tensor enable to compute as many correspondences as you want.



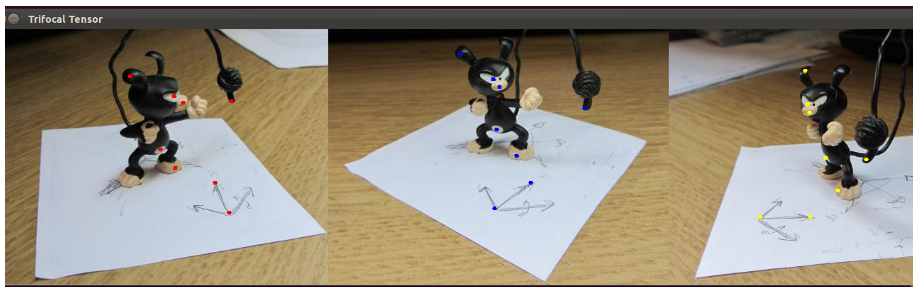
**./bin/trifocal -h** You can launch the application with this option if you want to display the help. There is a description which explain how you can use the application.



```
Man of the application.

NAME
        ./bin/trifocal     Application, example of using trifocal tensor.

SYNOPSIS
        ./bin/trifocal [-h] [image1 image2 image3] [list1 list2 list3]

DESCRIPTION
        This application is a demo using a trifocal tensor, an important notion of projective geometry.

        -h
                Launch application with no additional features, and display help menu on terminal.

        image1.jpg image2.jpg image3.jpg
                Launch application with your 3 images. You need to select correspondences by clicked on the images, and after that, type any key on your keyboard to start to use the
trifocal tensor.

        image1.jpg image2.jpg image3.jpg list1.list list2.list list3.list
                Launch application with your 3 images and your 3 lists. You just need to clic 2 points on 2 images, and the trifocal tensor do the job !
```

## 8 Our images

**./bin/trifocal marsu1.jpg marsu2.jpg marsu3.jpg** You can launch the application with this option, if you want to use your image (we put the marsupilami on our project). If you have list file, you can add them at the end of this command.



## Part VI

# Conclusion and possible improvements

With this project, we learned at various levels.

**Technical level** We used for the first time a revision control system : git. And also, we learned Latex, to make a report with the sofware TexMaker.

**Mathematics level** We discovered a little part of projective geometry, by resolving the problematic of trifocal tensor.

**Project Management** Because we were two on this project, we used a web-based hosting service for software development projects : bitbucket. Combined with git, it was a powerful tool, which allowed us to be more responsive and efficient during the development phase.