Technische Universität Hamburg-Harburg
Vision Systems

Prof. Dr.-Ing. R.-R. Grigat

3D Computer Vision Programming Project

B. Sc. Matthias Schlüter
M. Sc. Sven Painer

November 30, 2015

1 Introduction

In this programming project you will apply your knowledge about 3D Computer Vision in a practical task. You will track some markers in video sequences from a stereo rig and triangulate these points to get the motion of the markers in the 3D space.

For this project, C++ with the open source library *OpenCV*[1] will be used. This library contains all necessary algorithms to manage the project. To be independent of the programming environment, *CMake*[2] is used to create projects for the development enviroment of your choice.

OpenCV 3.0 is still under development and the documentation is well-hidden at the moment. You can find it at `http://docs.opencv.org/trunk/`.

2 Preparation

As a first step, it is necessary to install the tools needed for compiling the project. If you are unsure which compiler to use, we suggest using *Visual Studio*[3] 2012 or 2013 (Express edition is sufficient) on Windows. You also need to install *CMake* and a *SVN* tool, e.g. *TortoiseSVN*[4]. The last step is to download and unzip *OpenCV*. After you unzipped *OpenCV*, you have to add *OpenCV* to the environment variable `PATH`. The folder to add to the `PATH` variable is `build/x86/vc11/bin` inside the unzipped *OpenCV* folder for *Visual Studio 2012* and `build/x86/vc12/bin` for *Visual Studio 2013*.

After you installed all tools, you are ready to create your *SVN* repository. You will find the template for the programming project inside this repository. To create your repository, go to `https://tux.ti1.tu-harburg.de/3dcv/signin.php` and follow the instructions. In the end you will get an email containing the data to access your repository. You can now check out a local copy.

After checking out the files, you can create a project for your development environment by using *CMake*. Start the *CMake* GUI and choose the checked out folder as source folder. Choose also the build folder. This folder will be used for the output of the compiler. You could e.g. use a folder `build` inside the repository, but be sure not to add it to the version control. The simplest way is to add it to the ignore list. Next, click `Configure` and follow the instructions to choose the build environment. Now there will be variables in the middle of the window. Activate the option `Grouped` and open the group `OpenCV`. Check if the variable `OpenCV_DIR` points to the directory `build` inside your unzipped *OpenCV* folder. Otherwise change it accordingly and press `Configure` again. Now you are able to create the project by clicking on `Generate`. The project for your build environment is created inside the build folder and you can close *CMake*.

[1] `http://www.opencv.org/`
[2] `http://www.cmake.org/`
[3] `http://www.microsoft.com/de-de/download/details.aspx?id=34673`
[4] `http://tortoisesvn.net/`

If you are using another build environment than *Visual Studio*, you can continue with the next step, but be sure that the program can be compiled and run. You have to find out about possibly necessary steps for yourself. When using *Visual Studio*, open the solution `Project3DCV.sln` from the build folder. First, change the project that will be used for running the program by right click on the project `Project3DCV` and clicking on `Set as StartUp Project`. You should now be able to compile and run the program. The console closes right after the program has been finished, so you are not able to read the content of the console. When you start the program without debugging by choosing this option from the `DEBUG` menu or hitting `CTRL+F5`, the console will stay open until you press a key after the program has been finished.

3 Task

In the programming project you will create a program that tracks markers in video files from a stereo camera rig and triangulates the marker positions to get the positions in the 3D space. This section will describe the steps you have to do in order to finish the task. As the focus of the project is on the methods of 3D Computer Vision, you will get a template where the steps that are not related to 3D Computer Vision, like loading the data from files, are already finished. You will only have to concentrate on programming the interesting parts in the context of the lecture. The template also contains some helper functions that are described in Section 5. Be sure to check the available functions before you start programming as they will help you in debugging your program. When you are familiar with the helper functions, you can continue with this section as it will guide you step by step through the programming task.

3.1 Namespace

It is no good idea to declare anything in C++ without a namespace. Two libraries declaring different classes of the same name, e.g. `Matrix`, could not be used in a program at the same time. Therefore, C++ has the concept of namespaces. If symbols are unique within a namespace, then different libraries including equal class names can be used, because these classes are located in different namespaces. The classes and functions of the programming project are declared in the namespace `CVLab`. To use the classes or functions you have to prepend them with the namespace name and the scope operator (`::`) or you have to use the command `using namespace`. You do not have to take care of the namespace in the `cpp` files of the template as they all declare the namespace `CVLab` to be used in the beginning of the file.

3.2 Constants

Constant values should not be written directly into the program at the point of their usage. Instead, they should be declared as a constant variable in a central place. This has

two advantages. On the one hand it is likely that the value will be used in different places of the program. If you have to change the value, you would have to search for each and every occurence to change the value if it is not stored centrally. On the other hand you will have an overview of all values that can be changed to control your program. The constants for the programming project are declared in the namespace `Constants` inside the namespace `CVLab`. You can find the constants in the file `Constants.hpp`. You will not apply the predefined constants yourself, but you should know where to find their definitions in the program. You are also welcome to extend this file by your own constants.

3.3 Main Program

The entry point of your program is the function `main` that is defined in the file `main.cpp`. This function already contains the code to parse the command line arguments and load calibration data and the sequence. The program has to be called with three parameters. The first defines the folder where the calibration data can be found, the second defines the folder with the sequence and the third defines the file to write the results to. You have to set the parameters to call the program from the IDE. In *Visual Studio* this is done by clicking right on `Project3DCV` and then clicking on `Properties`→`Configuration Properties`→`Debugging`. You can then define the arguments in the field `Command Arguments`. Just enter the right paths and separate them by spaces. Wrap each path in quotation marks. Afterwards, the program should not complain about missing arguments any more.

As you can see, the code to load the calibration data and the sequence are already present, but the rest of the algorithm has not been implemented yet. You have to add the missing code of the algorithm yourself at locations indicated by `TODO` comments. In general it is recommended to call function templates in the main program before defining your classes for the different parts of the algorithms. Only this way you will be able to test your code appropriately. Since templates for all required functions have already been provided, you are able to compile and execute the program from the beginning. Those predefined templates initially only throw exceptions telling you which functions have not been implemented so far. The main program will catch these exceptions, print the message to the console and exit the program. You therefore work on the project by implementing the main program step by step, executing it and looking for some error message indicating the function to be implemented next.

3.4 Calibration

The camera calibration of the stereo rig has already been performed for you by using the *Caltech Camera Calibration Toolbox for Matlab*[5] and the calibration images that have already been given to you via *Stud.IP*. The results have been exported to *CSV* files

[5] `http://www.vision.caltech.edu/bouguetj/calib_doc/`

and can be imported into the program by using the class `Calibration`. This class has already been finished and you should have a look at the class declaration to see which calibration data you can retrieve. You can download the exported calibration data from *Stud.IP*.

3.5 Loading Sequence

Loading of the sequences is done by the class `Sequence`. This class loads the videos for the left and right camera, converts the images to grayscale and undistorts the images. It also loads the positions of the markers in the first frame. The marker positions are given in integer coordinates. As the marker position will most likely not be at an integer position, it can be refined. This is done by using the *OpenCV* function `cornerSubPix`. Have a look at the documentation of this function to understand how it works.

The member variable `markers` contains the marker positions for the first frame for both cameras. `markers[0]` is a vector of `Point2f` giving the marker positions for the first camera and `markers[1]` a vector of markers positions for the second camera. The positions in the vectors have to be corresponding, meaning that the first position in the vector for the first camera has to be the position for the same marker as the first position in the vector for the second camera. After the positions have been loaded by the method `readMarkers`, they have the same ordering as the *CSV* files and they are not guaranteed to be corresponding. Therefore, the method `sortMarkers` is called after loading the marker positions and it has to sort both vectors so that the positions are corresponding. This is your first task. Implement the method `sortMarkers` so that the positions in the vectors are corresponding after executing the method. You can use the helper function from Section 5.4 to check your results. The markers are plotted in different colors by this method, so be sure that the markers are plotted in the same colors in both images.

3.6 Tracking

After the sequences have been loaded and the marker positions are sorted to be corresponding, the marker positions in all frames of the sequence have to be estimated. This is done by the class `Tracking`. It is implemented as a functor and the function call operator will do the job of tracking the marker position in one video, so the tracking procedure has to be called for both cameras independently. It is your task to implement the tracking algorithm by implementing the function call operator of the class `Tracking`. The methods to be used for this are related to the *Optical Flow* and a tutorial for the *Optical Flow* in *OpenCV* can be found at `http://docs.opencv.org/trunk/d7/d8b/tutorial_py_lucas_kanade.html`. You can check your results with the helper function from Section 5.5.

3.7 Triangulation

When the positions of the markers for every frame of both cameras is known, the marker positions in the 3D space for each frame can be estimated by triangulation. This is done by the class `Triangulation`. It is implemented as a functor and the function call operator can triangulate the positions for a single frame or a whole sequence. As the variant for the whole sequence just calls the single frame variant for each and every frame of a sequence, it is already implemented. The single frame variant has to be implemented by you.

First, the projection matrix for both frames has to be calculated. It is a good idea to assume the first camera to be at the origin and calculate the position of the second camera accordingly. In the next step, the correspondences have to be corrected. Because of noise, the epipolar constraint

$$x'^{\mathrm{T}} F x = 0 \tag{1}$$

will not be fulfilled, but this is necessary for the triangulation. Therefore, the correspondences have to be corrected to fulfill the epipolar constraint. A solution for this problem can be found in Section 12.5 of *Multiple View Geometry* by Hartley and Zisserman. Keep in mind, there is also an *OpenCV* function implementing this method. After the matches have been corrected, the triangulation can be executed and there is also an *OpenCV* function for it.

The triangulated points are given in the coordinate system of the first camera, which is not the same as the world coordinate system. Therefore, the points have to be transformed into the world coordinate system. In the end, the points have to be dehomogenised and saved into the result vector. You can check your results with the helper function from Section 5.6.

As not the absolute position of the markers is of interest, but the motion, it has to be calculated. This is done in the class method `calculateMotion`. This method gets the triangulated marker data and will calculate the motion from it. The motion is nothing but the marker position relative to the first marker in the first frame. So you simply have to substract this position from each and every point. Again, you can check your results with the helper function from Section 5.6.

3.8 Save Results

The last step is to save your results to the output file. There is a helper function in Section 5.7 that will do the job for you, you just have to call it from the main program.

4 Evaluation

The first step in the evaluation is the compilation of your program. A successful compilation is mandatory to pass the programming project. Only the content of your *SVN* repository will be evaluated. Your repository is checked out by the test system, a project

for the compiler is created using *CMake* and then the program is compiled on the test system. If the compilation fails, you will directly fail the programming project and there will be no further checks.

After the program has been compiled, it will be run on eight different sequences. You can download four of these sequences from *Stud.IP* to test the program for yourself, the other four sequences are kept secret. In these sequences, two markers are moved by high precision motors and the ground truth data of the motions is known. You will find a text file inside the sequence folders describing the motions performed in the sequences.

The test system will perform these steps automatically, so it is important to remove all code that will wait for user input. The program should exit with the return code `EXIT_SUCCESS` after it wrote the results to the output file. If the program ends with a success code other than `EXIT_SUCCESS` this is considered as failing. The program will be called with different command line arguments to perform the tests on the different sequences. You will pass the programming project if your program can be compiled, executed with success on all sequences and has an error of less than 5 mm in each frame of all the sequences it is executed on. The maximum error of 5 mm is chosen arbitrarily. The sample solution has performed with a maximum error of 1 mm, caused by calibration inaccuracy and rounding errors. You can check your solution for consistency with the helper function of Section 5.6. In the description of the sequence you can find the expected motion and can compare it to the result, e.g. if only one motor has been moved during recording of the sequence, the motion of the markers has to be parallel to the corresponding coordinate axis.

You have to solve the programming project individually. You are not allowed to implement in a group and submit the same source code. We will check your submission for plagiarism and in uncertain cases you will have to prove that you solved the task for yourself. In case of plagiarism, all people involved will directly fail the programming project. Of course you are still allowed and encouraged to discuss questions about the programming project in working groups or on *Stud.IP*, but only based on minimized examples of the problem and not including major parts of your source code.

5 Helper Functions

The template provides some functions that can help you in developing and debugging your program. You can find the declarations of these functions in the file `tools.hpp`. There you will also find a comment for each function that describes the parameters. If your are interested in the definitions of these functions and want to learn more about *OpenCV*, you can also have a look at the file `tools.cpp`.

5.1 Read a Matrix

The function `readMatrix` reads an *OpenCV* matrix from a *CSV* file. *CSV* stands for comma-separated values. This is a special format of text files. Each line in the file

corresponds to a row in the matrix. The columns in the matrix are then separated by commas. The function automatically determines the number of rows and columns in the file. The elements of the resulting matrix are always of type `float`.

5.2 Check Matrix Dimensions

The function `checkMatrixDimensions` checks the dimensions of a given matrix and is useful for checking pre- and postconditions of an algorithm. The function gets the matrix and the expected dimensions and throws an exception if these do not match. If only one dimension should be checked, the other dimension can be ignored by giving a negative value for the parameter. A name for the matrix can also be given that will be used in the error message. The error message will also contain the true dimensions of the matrix.

5.3 Show an Image

The function `showImage` shows an image and pauses the program until a key is pressed. It simply calls the *OpenCV* functions `imshow` and `waitKey`. The function has an optional parameter for the window title. If there already exists a window with the same title, the image in the window will be overwritten by the new one. There is also a flag telling the function whether to wait for user input or simply continue with the program.

5.4 Show an Image with Markers

The function `showImageMarkers` shows an image with marker positions illustrated by colored crosses. The ordering of the colors is always the same, so the color of the first marker will always be the same color when you call this function. Therefore, it is possible to call the function for one image of each camera and check whether the marker ordering is the same for both cameras. The function has also optional parameters for the window title and a wait flag. These are described in Section 5.3 as the function internally calls `showImage` with these parameters.

5.5 Show a Sequence with Markers

The function `showSequenceMarkers` shows a whole sequence with marker positions illustrated by colored crosses in each frame. This function calls `showImageMarkers` for each frame of the sequence, so the descriptions in Section 5.4 also apply to this function. The only difference is that the wait flag is only applied after the last frame, so the program will not wait for user input after each frame.

5.6 Show Triangulation Results

The function `showTriangulation` shows a 2D plot of the triangulation results. As the motion in the sequences is only performed in x- and y-direction, the movement in the z-axis is omitted in the plot. The plot is held very easy, so it is only qualitative and does not show any quantitative elements such as axis scaling. It only intents to help you by seeing if a motion is for example only performed in the x-axis and not in the y-axis. The result is always scaled to fit into the plot. Therefore, you can call this function with the absolute triangulation result as well as the relative motion. The parameters `title` and `wait` are the same as in Section 5.3 as the function `showImage` is called internally to show the plot.

5.7 Save Triangulation Results

The function `writeResult` writes the triangulation results to a *CSV* file. For a short description of *CSV* see Section 5.1. Each marker position is stored in a separate line. For each marker, the index of the frame, the index of the marker and the x-, y- and z-coordinate are stored, separated by commas. This file will be used in the evaluation of your program.

5.8 Log Messages

The function `logMessage` can be used to write log messages to the console. It writes the given message to `cout`, prepended by the current time.