An Augmented Reality application

A step-by-step approach

Vision par Ordinateur, 2015-2016
3^e année IMA, Parcours Multimédia

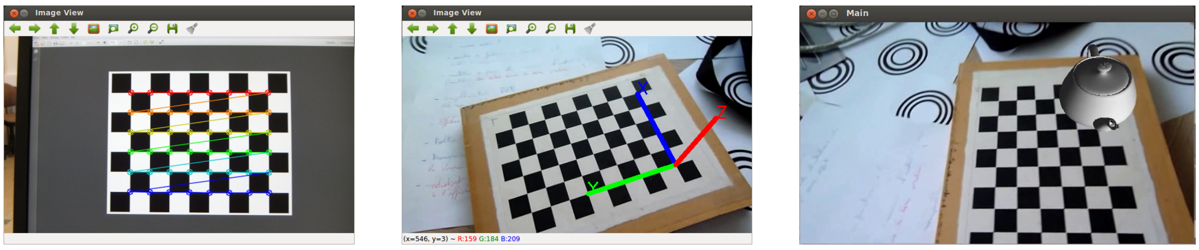
November 2015

Contents

1 Objectives

The objective of these TP sessions is to build a simple augmented reality application using OpenCV for the camera tracking part and OpenGL for the rendering. The exercises that will be proposed are meant to gradually introduce you to the OpenCV libraries and build all the function that are needed for a camera tracker in an incremental way, so that you will be able to test and debug each function, before plugging all together in the camera tracker class.

The main idea is to use a chessboard as a marker and render on top of that the augmented reality. The final application should render an OpenGL teapot on top of the chessboard, as shown



(a) (b) (c)

Figure 1: Some of the steps that will get you to implement the AR application.

in Figure 1.(c). We will get there step by step, first detecting the chessboard (*c.f.* Figure 1.(a)), then create some basic rendering in OpenCV (*c.f.* Figure 1.(b)), and at last plugging everything in the OpenGL pipeline.

1.1 References

All you need to know about OpenCV will be given to you in the form of some tutorials that explains the basic data structures that we are going to use. Moreover, we will refer to the on-line documentation for the functions: in this text the OpenCV functions we need to use are normally associated with a hyper-link that refers you to the on-line documentation. More in general, you can also find some other resources here:

- Official page of OpenCV project `http://opencv.org/`
- Online documentation of the API `http://docs.opencv.org/`
- Online tutorials `http://docs.opencv.org/doc/tutorials/tutorials.html`.

2 The provided code environment

Uncompress the archive that you received. This directory is organized as follows:

- `doc` contains a pdf copy of this text, a pdf copy of the reduced OpenCV tutorials, a pdf copy of the full, original OpenCV tutorial, and a pdf copy of the API reference manual of OpenCV (*i.e.* the copy of the online documentation).
- `data` contains some images, videos and other data that will be used through the TP.
- `src` contains the source files that you have to modify and complete; they are organize in directories:
  - `tutorials` contains the code used in the tutorials, in case you need to try it;
  - `tp` contains the files that you need to modify and complete through all the sessions of the TP.
- `3dparty` this directory contains a library that will be used at the end of the TP. You don't have to worry (or do anything...) about this directory.

2.1 Building the code with CMake

CMake is a cross-platform free software program that helps managing the build process of software using a compiler-independent method. In simpler words, this means that CMake is an utility that helps to set up the compilation environment for a given source code, independently from the compiler and the building system that is actually used to generate the code, be it Linux's `make`, Apple's `Xcode`, or Microsoft Visual Studio.

In the case of this TP, CMake will check for all the libraries that are needed to compile our programs and it will automatically generate the corresponding `Makefile`. Let's see how this work.

Extract the zipped file into a directory in your home. Make sure that the path where you extract it does not contain spaces. For example if you extract it in your home:

```
sgaspari@serval:~$ tar -xzvf adm.tar.gz
sgaspari@serval:~$ cd adm
```

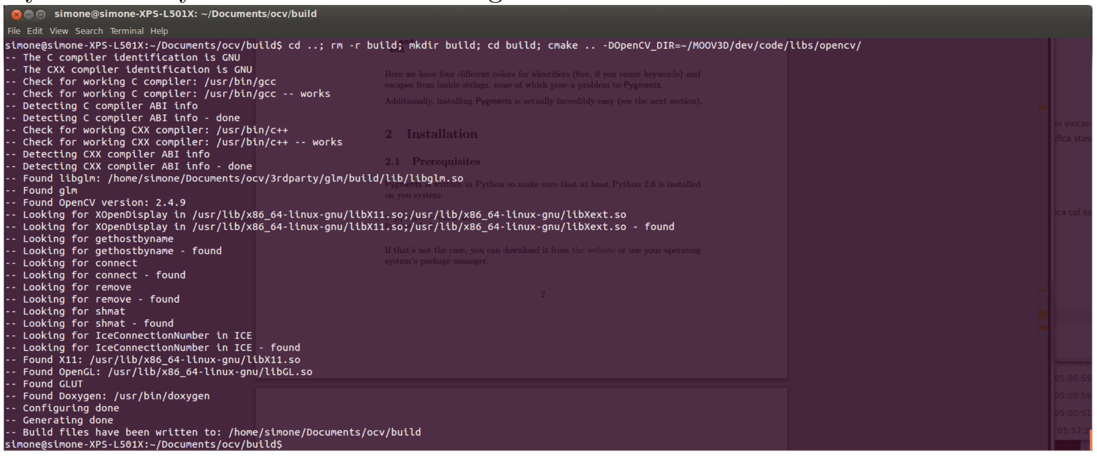
Then, create a new directory, let's call it `build`:

```
sgaspari@serval:~/adm$ mkdir build
sgaspari@serval:~/adm$ cd build
```

This directory will be your compilation directory where the `make` is executed and it will contain the results of the compilation, *i.e.* all the executables that you will generate. Now we can tell CMake to configure and create the relevant makefile for us:

```
sgaspari@serval:~/adm/build$ cmake ..
```

CMake will look for all the necessary libraries and generate the makefile for all the programs.[1] On your screen you should see something like this:


simone@simone-XPS-L501X: ~/Documents/ocv/build
File Edit View Search Terminal Help

```
simone@simone-XPS-L501X:~/Documents/ocv/build$ cd ..; rm -r build; mkdir build; cd build; cmake .. -DOpenCV_DIR=~/MOOV3D/dev/code/libs/opencv/
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Found libglm: /home/simone/Documents/ocv/3rdparty/glm/build/lib/libglm.so
-- Found glm
-- Found OpenCV version: 2.4.9
-- Looking for XOpenDisplay in /usr/lib/x86_64-linux-gnu/libX11.so;/usr/lib/x86_64-linux-gnu/libXext.so
-- Looking for XOpenDisplay in /usr/lib/x86_64-linux-gnu/libX11.so;/usr/lib/x86_64-linux-gnu/libXext.so - found
-- Looking for gethostbyname
-- Looking for gethostbyname - found
-- Looking for connect
-- Looking for connect - found
-- Looking for remove
-- Looking for remove - found
-- Looking for shmat
-- Looking for shmat - found
-- Looking for IceConnectionNumber in ICE
-- Looking for IceConnectionNumber in ICE - found
-- Found X11: /usr/lib/x86_64-linux-gnu/libX11.so
-- Found OpenGL: /usr/lib/x86_64-linux-gnu/libGL.so
-- Found GLUT
-- Found Doxygen: /usr/bin/doxygen
-- Configuring done
-- Generating done
-- Build files have been written to: /home/simone/Documents/ocv/build
simone@simone-XPS-L501X:~/Documents/ocv/build$
```

This has to be done only once at the beginning, from now on you have just to compile the code you modify in `src` using the usual `make` command. Indeed, in the `build` directory you can see the `Makefile` we will use to compile the code. The code can be compiled form the `build`

[1] In case you want to try the code on a different machine, you need to first install the OpenCV libraries (from here) and then do `cmake .. -DOpenCV_DIR=path/to/OpenCVConfig.cmake` in order to specify the directory where you build them. More in general, you need to provide the path to the file `OpenCVConfig.cmake`, which you can find *e.g.* with `locate OpenCVConfig.cmake` from your shell.

directory with the usual

```
sgaspari@serval:~/adm/build$ make <filename_without_extension>
```

and

```
sgaspari@serval:~/adm/build$ make clean
```

to delete all the executable and the compilation objects. Try for example to build the code for the first tutorial with

```
sgaspari@serval:~/adm/build$ make mat_the_basic_image_container
```

The executable will be placed in `build/bin/` and you can run the code with

```
sgaspari@serval:~/adm/build$ ./bin/mat_the_basic_image_container
```

You can compile all the tutorials in a batch with

```
sgaspari@serval:~/adm/build$ make tutorials
```

Finally, you can see all the possible target for `make` with

```
sgaspari@serval:~/adm/build$ make help
```

2.1.1 Creating project files to work in Eclipse

If you want create a project file for Eclipse to that you can use the auto-completion and the pop-up documentation of the code, you can create it with `CMake` in a different directory:

```
sgaspari@serval:~/adm/$ mkdir myARproject
sgaspari@serval:~/adm/$ cd myARproject
sgaspari@serval:~/adm/myARproject$ cmake -G"Eclipse CDT4 - Unix Makefiles" ../adm:
```

2.2 Quick start with OpenCV

In these exercises we will use the data structure `Mat` provided by OpenCV to deal with images and matrices. We will use the OpenCV functions to create the windows and show the images inside them, to load images and to process video streams. We have already seen an introduction to these topics in class, you can use the document containing the short tutorials as reference (these topics are covered in the first 4 tutorials).

In each exercise, whenever a new OpenCV function is used, you will find a link to its on-line documentation.

3 Detecting the chessboard

In this first exercise we want to build a program that given an image as input, detect the a chessboard in the image and draw the detected chessboard corners (see Figure 2). This will be the first step towards our camera tracker as we will use the chessboard as a marker on top of which we will render a virtual object. To this end, we will rely on the OpenCV function for detecting calibration patterns. Currently OpenCV supports three types of object for calibration (see Figure 3):

- Classical black-white chessboard
- Symmetrical circle pattern
- Asymmetrical circle pattern

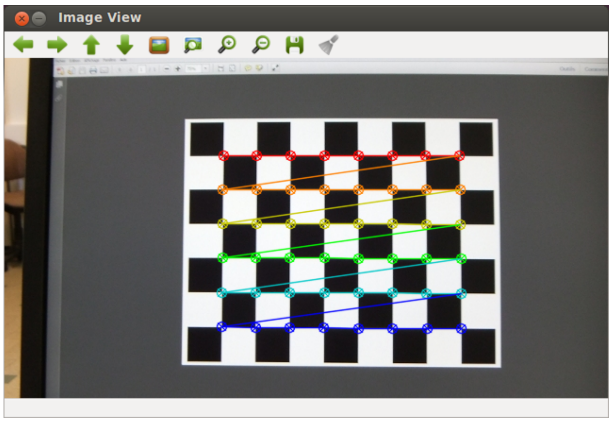


Figure 2: Example of expected output. The OpenCV function detects the corners of the chessboard, which are marked with a cross surrounded by a circle (⊗). The size of the pattern (in terms of corners) is given by the number of inner corners of the chessboard: in the example above the size of the chessboard is 8×6 (width×height)

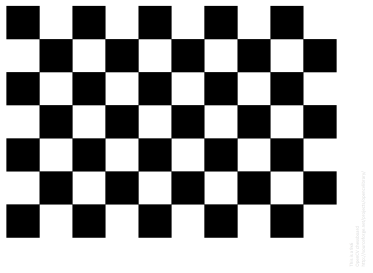
Assignment

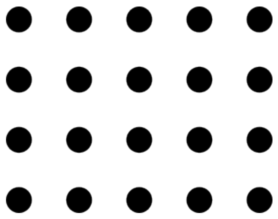
Complete the program in `src/tp/checkerboard.cpp`. The program takes as input the size of the pattern to dectect, the type of pattern and the name of the image to process. Here is the complete documentation:

```
Detect a chessboard in a given image
Usage: ./bin/checkerboard -w <board_width> -h <board_height> -pt chess <image file>

-w <board_width>                             # the number of inner corners per one of board dimension
-h <board_height>                            # the number of inner corners per another board dimension
[-pt <pattern=[circles|acircles|chess]>]     # the type of pattern: chessboard or circles' grid
<image file>
```

For example, to process the image `data/images/left01re.jpg` we can launch the program with (from `build` directory):


(a) A 9×6 chessboard


(b) A 5×4 circle pattern

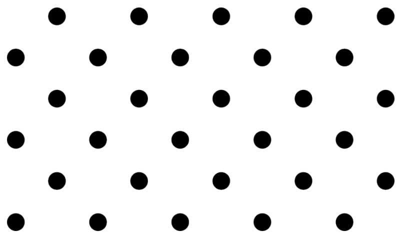

(c) A 5×6 asymmetric circle pattern

Figure 3: Some examples of the 3 type of patterns that can be used for calibration purposes by OpenCV

```
./bin/checkerboard -w 8 -h 6 ../data/images/left01re.jpg
```

Look at Figure 3 to see how the size of the pattern is computed in terms of corners. In `data/images/` you can find other test images (not all the images give a positive detection).

The program imports the function that detects the chessboard from `src/tp/tracker/utility.cpp`. This will be the first function we will add to our camera tracker.

1. Complete the function `detectChessboard` in `src/tp/tracker/utility.cpp` that, given an input image, the type and the size of the pattern, detect the chessboard in the image (if any) and return a boolean value to confirm the detection and the list of detected corners/points of the pattern.

2. Complete the `main` with the call to `detectChessboard` and the code to display the image. Use the second tutorial of OpenCV *Load and Display an Image* as reference to create the window and display the image.

When you run the program you can see the detected chessboard. Also, the detection time will be printed on the shell. As you can see it is around $\sim$ 20ms, which means the detection algorithm can detect the chessboard up to a rate of 50 frame per seconds.

Hints

- The parsing of the input arguments is already implemented with the function `parseArgs`, nothing to be done there. Also you will find another function `help` that prints on the screen the help of the program. All the programs we are going to implement in these TP have similar functions, you don't have to modify them.

- The functions and the pieces of code you have to implement or add are normally preceded by a comment explaining what you have to do and which function you have to call. Normally these comments are surrounded by this pattern `/****************/`.

- An enumerative type is already defined in `tracker/tracker/utility.hpp` containing the 3 different calibration patterns

```
enum Pattern { CHESSBOARD, CIRCLES_GRID, ASYMMETRIC_CIRCLES_GRID };
```

  They can be used to call the relevant detection functions (*c.f.* next item).

- Depending on the type of the input pattern you use either the `findChessboardCorners` or the `findCirclesGrid` function. For both of them you pass on the current image, the size of the board and you'll get back the positions of the patterns. Furthermore, they return a boolean variable that states if in the input image we could find or not the pattern.
- For the chessboard the position of the corners is only approximate. We can improve and refine their position by calling the `cornerSubPix` function.
- Finally, for visualization feedback purposes we will draw the found points on the input image with the `drawChessboardCorners` function.

4 Detecting the chessboard on a video

Now that we can detect a chessboard on an image, let's move one step forward and write a program that detected the chessboard in a given video stream.

Assignment

Complete the program in `src/tp/checkerboardvideo.cpp`. The program is very similar to the previous one, it takes as input the size of the pattern to detect, the type of pattern and the name of the video to process. Here is the complete documentation:

```
Detect a chessboard in a given video
Usage: bin/checkerboardvideo -w <board_width> -h <board_height> [-pt <pattern>]  <video file>

 -w <board_width>                          # the number of inner corners per one of board dimension
 -h <board_height>                         # the number of inner corners per another board dimension
 [-pt <pattern=[circles|acircles|chess]>]  # the type of pattern: chessboard or circles' grid
 <video file>
```

In `data/video` you can find the video of a chessboard, we will use that video from now on for all the our programs. For the chessboard detection you can launch:

```
./bin/checkerboardVideo -w 9 -h 6 ../data/video/calib.avi
```

There are not many changes to do w.r.t. the previous version:

1. The program will import the function `detectChessboard` you already developed from `utility.hpp`.
2. Complete the `main` with the code for opening a video and getting the current frame, the call to `detectChessboard`, and the code to display the image. Use the OpenCV tutorial *Load and Display a Video* as reference.

When you run the program you should notice that the detection works in real-time when the chessboard is completely visible. On the other hand the detection process is slowed down if one or more corner of the chessboard are occluded: this is enough to prevent the detection. For now, we will stick with the tracking by detection paradigm. We will see later how we can improve the tracking performance.

Hints

- `VideoCapture` is the class that loads the video and allows to access to each frame.
- At the end of each iteration of the processing loop, the program wait for user input for 10ms: if any key is pressed the program ends.

  ```
  if( waitKey( 10 ) >= 0)
          break;
  ```

  For debugging purposes, you can replace it with the following

  ```
  if( waitKey( -1 ) == 'q')
          break;
  ```

`waitKey( -1 )` will stop the execution until a key is pressed. If the key Q is pressed then the program will exit the loop and terminate. Otherwise it will go on with the next iteration.

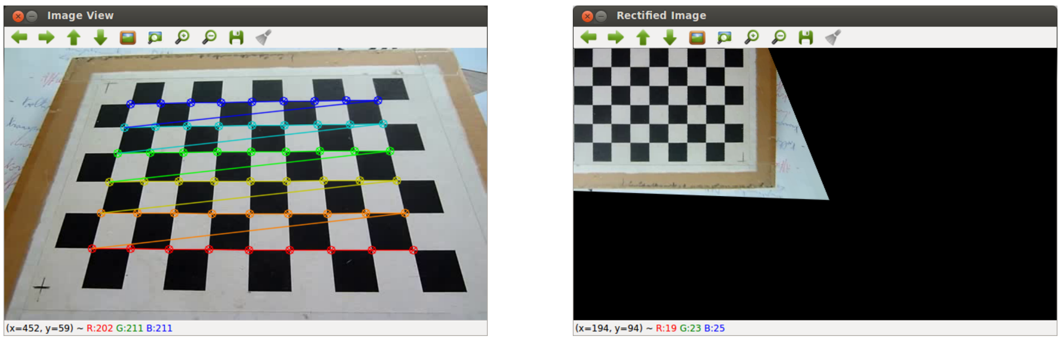


Image View
(x=452, y=59) ~ R:202 G:211 B:211
Rectified Image
(x=194, y=94) ~ R:19 G:23 B:25

Figure 4: The program has two windows, one showing the original video, the other the rectified version (if the chessboard is detected).

5 Perspective rectification

In order to estimate the position of the camera w.r.t. the chessboard we need to estimate the homography relating the 3D plane of the chessboard and the plane image. In this exercise we will modify the previous code in order to estimate the homography in a robust way every time the chessboard is detected. In order to verify that the homography is well estimated, we will use the homography to remove the perspective distortion of the chessboard. If the homography is well estimated, in the rectified image the chessboard should appear as seen "from above" (*c.f.* Figure 4).

Assignment

Complete the program in `src/tp/checkerboardvideoRectified.cpp`. The program has the same synopsis of the last program of Section 4:

```
Usage: bin/checkerboardvideo -w <board_width> -h <board_height> [-pt <pattern>]  <video file>
```

The program must create 2 windows, one to display the original video, the other the rectified version (provided that the chessboard is detected).

1. Again, the program imports the function `detectChessboard` that you have implemented earlier from `tracker/utility.hpp`.

2. Complete the `main` with the code for creating 2 windows, opening a video and getting the current frame, and the call to `detectChessboard`.

3. In order to estimate the homography we need the coordinates of the points on the image plane and the corresponding points on the chessboard. For the formers we will just use the corner detected with `detectChessboard`. For the points on the chessboard, we can fix an arbitrary reference system on the chessboard and generate the points accordingly: for example we can consider the top left corner of the chessboard as the origin with the x-axis and the y-axis oriented as depicted in Figure 5

   Complete the function `calcChessboardCorners` that you find in `src/tp/tracker/utility.cpp`, which generates the points on the chessboard (for the purpose of this exercise they will be

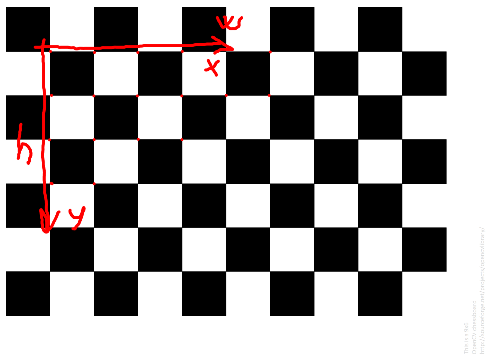


Figure 5: An arbitrary reference system on the chessboard.

2D points).

4. With the correspondences we have found, estimate the homography in a robust way using the RANSAC algorithm

5. Once the homography is estimated, apply the homography to the current frame in order to remove the perspective distortion.

6. Finally display the two images, the original one and the rectified one, inside the relevant window.

Hints

- To estimate the homography OpenCV provides the function `findHomography`. Look at the online documentation for the various parameters. Remember to pass the flag `CV_RANSAC` to use a robust estimation method.

- Remember: if you write `H = findHomography( x1, x2...)` then $\mathbf{x}_2 = \mathtt{H}\,\mathbf{x}_1$. The parameters `x1` and `x2` can be invariably the image points and the points on the chessboard, or viceversa. You will just obtain two different homographies which are one the inverse of the other, *i.e.*

  `findHomography( x1, x2...) == (findHomography( x2, x1...))`$^{-1}$.

  In order to choose the order, consider which transformation you have to pass to `warpPerspective` later. . . (*c.f.* next hint).

- To remove the perspective distortion using the estimated homography we can use the function `warpPerspective` which applies a perspective transformation to an image. Use the default parameters for the borders and the interpolation.

6 Camera calibration

In order to build a camera tracker that estimates the camera position w.r.t. the chessboard we need to know the intrinsic parameters of the camera. In particular we will need to estimate the calibration matrix `K` and the distortion coefficients $\mathbf{d}$ of the optical distortion model. OpenCV provides some functions that allow to calibrate the camera from a set of images (or a video) of a calibration pattern. In order to calibrate the camera we will rely on a program that comes with OpenCV. If you need a refresh about the calibration theory and the internal parameters of the camera, you can read the tutorial *Camera Calibration with OpenCV*.

Assignment

For the task of calibrating the camera you are not required to write any code. We will instead use the program `src/tp/calibration.cpp` that is provided by OpenCV. The program has many options and can work with different input. We will use the video `data/video/calib.avi` as input. The program will load the video and will try to detect a chessboard. When a certain number of positive detections are achieved, it will use the detected points on the chessboard to estimate the camera parameters, and then it will save them in a file.

1. Compile the program and run it once without passing any arguments. It will display its usage. The parameters between square brackets (`[ ]`) are optional.
2. We want to use our `data/video/calib.avi` to calibrate the camera. We need to call the program passing the following options:
   - the size of the board (`-w` and `-h`) as we did for the previous programs
   - we want to use a video as input (`-V` option)
   - we want to set the number of positive detection to use in order to calibrate the camera (`-n` option), *e.g.* 10.
   - 1500ms
   - we want a simplified model for the optical distortion without the tangential component (`-zt` option)
   - we want to specify the name where the intrinsic parameters are saved (`-o` option); the output file can be either an xml file, *e.g.* `calib.xml`.
3. When you launch the program it will detect the chessboard on the video, but, in order to start the calibration process you need to press G.
4. At the end of the process you should have the xml file in your directory. The content should look like this

```
<?xml version="1.0"?>
<opencv_storage>
<calibration_time> "Mon 21 Oct 2013 07:16:31 PM CEST" </calibration_time>
<image_width>640</image_width>
<image_height>360</image_height>
<board_width>9</board_width>
<board_height>6</board_height>
<square_size>1.</square_size>
```

```
<!-- flags: +zero_tangent_dist -->
<flags>8</flags>
<camera_matrix type_id="opencv-matrix" >
        <rows>3</rows>
        <cols>3</cols>
        <dt>d</dt>
        <data>
                4.8428509485313901e+02 0. 3.2009932991075260e+02 0.
                4.8365482958967795e+02 1.6283922248766723e+02 0. 0. 1.</data></camera_matrix>
<distortion_coefficients type_id="opencv-matrix"
        <rows>5</rows>
        <cols>1</cols>
        <dt>d</dt>
        <data>
                -9.3396873795605365e-02 2.3550692129669618e-01 0. 0.
                -3.8791765186556898e-01</data></distortion_coefficients>
<avg_reprojection_error>1.7869347177607542e-01</avg_reprojection_error>
</opencv_storage>
```

The calibration matrix `K` is stored inside the `<camera_matrix>` tag and the distortion coefficients are stored inside the `<distortion_coefficients>` tag. We will see soon how to recover them from the file and use them for our purposes.

6.1 Removing the optical distortion

Once we have calibrated the camera, we can use the intrinsic parameters and the distortion coefficients to remove the optical distortion from the video.

Assignment

We want to implement a program that given a video and the relevant camera parameters as input, will remove the optical distortion and show the undistorted image. Also we want to be able to see the image difference between the distorted and undistorted image (*c.f.* Figure 6). The user must be able to switch between each image using the key O for the original image, U for the undistorted version and D for the image difference.

Complete the program in `src/tp/checkerboardvideoUndistort.cpp`. The program synopsis is:

```
Undistort the images from a video
Usage: ./bin/checkerboardvideoUndistort -c <calib file> <video file>
                 -c <calib file>  # the name of the calibration file
                 <video file>     # the name of the video file to process
```

where the calibration file is the file we have generate with the calibration process.

1. Complete the local function `loadCameraParameters` that loads the parameters from the xml file.
2. Complete the `main` with the usual code to create the window and open the video file
3. Complete the code inside the loop applying the undistortion and computing the image differece.

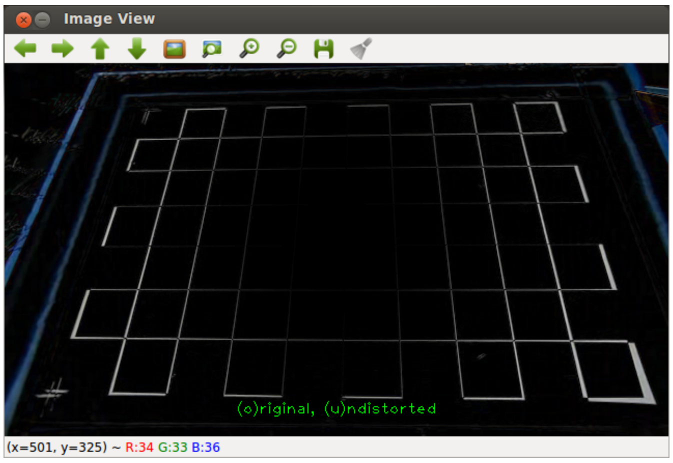


Figure 6: The difference image between the original and the distorted one. Brighter pixels correspond to higher difference between the two images.

When running the program, if you switch from the original image to the undistorted version you should notice that the image "moves", changes more on the region far from the center of the image, where the center of distortion lies. This is because the distortion varies with the square of the distance from the distortion center. You can see this effect in the difference image: the center of the image is generally darker than the outer regions, since the difference between the distorted and undistorted version is smaller.

Hints

- In order to load data from a xml file you can take a look to the quick tutorial *File Input and Output using XML and YAML files.* Also you can refer to the class `FileStorage`.
- We just need to read the calibration matrix stored in the `<camera_matrix>` tag and the distortion coefficients stored in the `<distortion_coefficients>` tag of the xml calibration file saved earlier.
- In order to undistort the image OpenCV provides the function `undistort`.
- In order to compute the image difference OpenCV provides the function `absdiff`

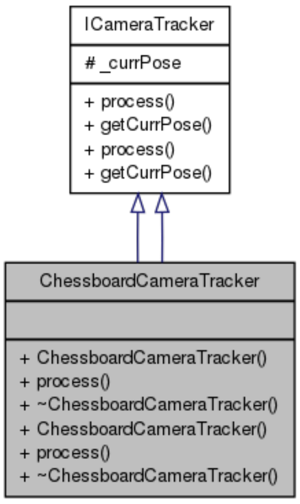


Figure 7: The class diagram for the camera tracker based on the tracking by detection approach.

7 The camera tracker

We have now most of the functions we need to build our first camera tracker. This first version of the camera tracker implements a *tracking by detection* method: each frame is processed independently to detect the chessboard and eventually compute the camera pose. We will use two main classes:

- `Camera`, which contains all the information about the camera (calibration matrix, distortion coefficients...). It has one main function and 3 members:
  - `init()`, is the function that initializes the class loading the camera parameters from a file.
  - `matK`, `distCoeff`, `imageSize` are the member variables containing the calibration matrix, the distortion coefficients and the size of the image, respectively.
- `ChessboardCameraTracker` that is in charge to process the image and compute the pose of the camera w.r.t. the chessboard. Since we will implement two versions of the camera tracker, we use a virtual class `ICameraTracker` that is the base class for the camera tracker algorithms that we are implementing (*c.f.* Figure 7). As you can see there are two main methods:
  - `process()`, is the main method that takes a frame as input and return the pose of the camera if a chessboard is detected.
  - `getPose()`, it just returns the estimated pose for the camera.

Assignment

In order to build the camera tracker you have to complete some functions of `Camera` and `ChessboardCameraTracker` and the main file `src/tp/tracking.cpp`, which actually use the two classes for displaying the reference frame on the detected chessboard. The program takes as usual the chessboard size, the calibration file and a video as input and it display the video with the augmented reference frame:

```
Detect a chessboard and display an augmented reference system on top of it
Usage: ./bin/tracking -w <board_width> -h <board_height> -c <calib file> <video file>
         -w <board_width>      # the number of inner corners per one of board dimension
         -h <board_height>     # the number of inner corners per another board dimension
         -c <calib file>       # the name of the calibration file
         <video file>          # the name of the video file
```

1. Complete the `Camera` class in `src/tp/tracker/Camera.cpp` by implementing the method `init()` that takes as input the name of a file containing the calibration parameters and it loads them in the member variables.

2. Implement the function `decomposeHomography()` in `src/tp/tracker/utility.cpp`: the function takes as input a homography matrix (3×3) and the camera calibration matrix, and it returns the 3×4 pose matrix.

3. Complete the `Tracker` class in `src/tp/tracker/ChessboardCameraTracker.cpp` by implementing the method `process()`:

   - The function takes as input the current frame, the camera parameters, the size and the type of chessboard and it returns `true` if the chessboard is detected, along with the pose of the camera. It also modifies the input image to correct the optical distortion.
   - The function is very similar to what we saw in Section 5, it detects the points on the chessboard, computes the homography and then decomposes it to compute the camera pose. Remember to undistort the image first.

4. Complete the function `drawReferenceSystem()` in `src/tp/tracker/utility.cpp`: the function takes as input an image, the camera parameters, the camera pose and it draws a reference system according to the camera pose (*c.f.* Figure 8).

5. Complete the function `main` in `src/tp/tracking.cpp`:

   - It initializes the object `cam`, an instance of the `Camera` class, with the calibration file we generated earlier,
   - It loads the input video and processes each frame of the video by calling the `process()` method of the object `tracker`, an instance of the `ChessboardCameraTracker` class.
   - If the tracker detects the chessboard it draws the reference system in the image.

6. The reference system should look like the one in Figure 8... You may want to apply a proper rotation to the 3D points you draw to get a similar reference system.

Hints

- Remember the homography decomposition: we can estimate an homography $\mathtt{H}$ up to a scale factor of the real one, *i.e.* $\mathtt{H} = \begin{bmatrix}\mathbf{h}_1 & \mathbf{h}_2 & \mathbf{h}_3\end{bmatrix} = \lambda \mathtt{K} \begin{bmatrix}\mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t}\end{bmatrix}$. The simplest way

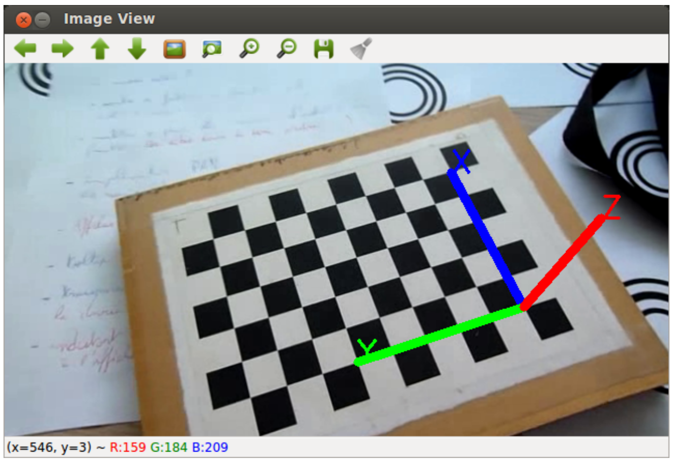


Figure 8: The reference system drawn on top of the chessboard.

to decompose is to consider:

$$\begin{cases} \mathbf{r}_1 & = \lambda \mathtt{K}^{-1}\mathbf{h}_1 \\ \mathbf{r}_2 & = \lambda \mathtt{K}^{-1}\mathbf{h}_2 \\ \mathbf{r}_3 & = \mathbf{r}_1 \times \mathbf{r}_2 \\ \mathbf{t} & = \lambda \mathtt{K}^{-1}\mathbf{h}_3 \end{cases}$$

with

$$\lambda = \frac{1}{||\mathtt{K}^{-1}\mathbf{h}_1||} = \frac{1}{||\mathtt{K}^{-1}\mathbf{h}_2||}.$$

- In order to undistort the image OpenCV provides the function `undistort()`.
- In order to draw the reference system we can draw 3 lines corresponding to the image of the 3 axes. We know the 3D coordinates of the points on the axis, so we can project them onto the image and draw the lines connecting them:
  - To project the points you can use the function `myProjectPoints` in `utility.cpp` which is a wrapper of the original OpenCV function `projectPoints()`: it takes the same arguments, except for the rotation `rvec` and translation `tvec` that are merged into a single matrix (the pose matrix).

```
/**
 * Wrapper around the original opencv's projectPoints
 * @param[in] objectPoints the 3D points
 * @param[in] poseMat the pose matrix
 * @param[in] cameraMatrix the calibration matrix
 * @param[in] distCoeffs the distortion coefficients
 * @param[out] imagePoints the projected points
 */
void myProjectPoints( cv::InputArray objectPoints,
                                          const cv::Mat &poseMat,
                                          cv::InputArray cameraMatrix,
                                          cv::InputArray distCoeffs,
                                          cv::OutputArray imagePoints);
```

  - For drawing the line and putting the text for the axis you can use the OpenCV functions `line()` and `putText()`.

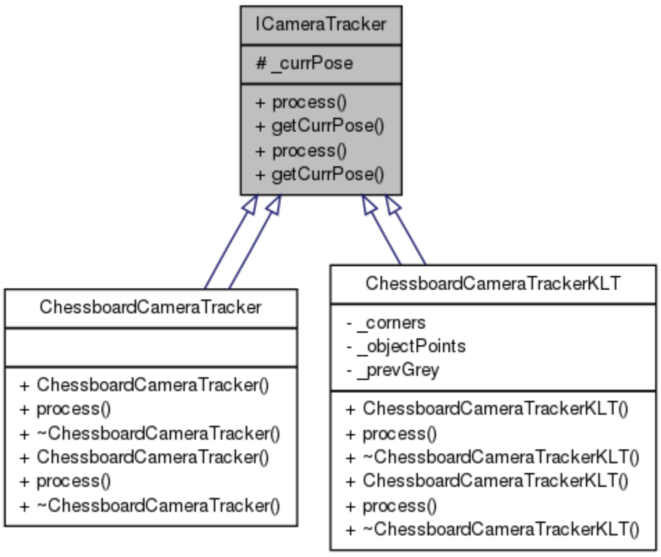

ICameraTracker
# _currPose
+ process()
+ getCurrPose()
+ process()
+ getCurrPose()
ChessboardCameraTracker
+ ChessboardCameraTracker()
+ process()
+ ~ChessboardCameraTracker()
+ ChessboardCameraTracker()
+ process()
+ ~ChessboardCameraTracker()
ChessboardCameraTrackerKLT
- _corners
- _objectPoints
- _prevGrey
+ ChessboardCameraTrackerKLT()
+ process()
+ ~ChessboardCameraTrackerKLT()
+ ChessboardCameraTrackerKLT()
+ process()
+ ~ChessboardCameraTrackerKLT()

Figure 9: The class diagram for the camera tracker based on the feature tracking approach.

8 The KLT version

As you may have noted, the main problem with the *tracking by detection* approach is that occlusions prevent the detection of the chessboard and thus the estimation of the camera pose. This is a limit of the approach due to the fact that the frames are processed independently, without, *e.g.*, using any information from the previous frame. A different approach for camera tracking based on markers is the *detect and track*: the marker is detected once and then its position in the image is tracked along the subsequent frames. In this exercise we will build a camera tracker that detect the chessboard and then track the corners using the Kanade-Lucas-Tomasi method (KLT).

We will implement a new class, `ChessboardCameraTrackerKLT`, which is again a derived class of the base class `ICameraTracker` (*c.f.* Figure 9). It has a specialized method `process()` for detecting and tracking the chessboard, and three new member variables:

- `_prevGrey` always contains the grey level version of the last frame. It has to be update at the end of each processing step.
- `_corners` contains the detected corners of the chessboard that are tracked from one frame to another.
- `_objectPoints` contains the 3D points of the chessboard.

Assignment

In order to build the camera tracker you have to implement the function `process()` of the class `ChessboardCameraTrackerKLT` and the `main` in `src/tp/trackingKLT.cpp`, which is basically the same as in Section 7 but using a KLT tracker. The synopsis is basically the same:

```
Detect and track a chessboard with the KLT algorithm and display the reference system on top of it
Usage: ./bin/tracking -w <board_width> -h <board_height> -c <calib file> <video file>
         -w <board_width>      # the number of inner corners per one of board dimension
         -h <board_height>     # the number of inner corners per another board dimension
         -c <calib file>       # the name of the calibration file
         <video file>          # the name of the video file
```

1. Complete the `process()` function so that:
   - The tracker will try to detect the chessboard if it is tracking too few points (or none), *e.g.* it is tracking less than 10 points. In this case it will try to detect the chessboard and eventually compute the camera pose.
   - Otherwise it will track the points using the optical flow algorithm. When the new points are estimated, the position of the camera is computed using these points. The optical flow algorithm returns the position of the features on the new image along with a vector `status` containing either 0 or 255 to indicate whether the estimate for the new point position is reliable or not: remember to filter out those points having the corresponding status element set to 0.
   - For the camera pose we will use the PnP algorithm with RANSAC.
2. Complete the function `main` in `src/tp/trackingKLT.cpp`: we will now use a `tracker` object from the `ChessboardCameraTrackerKLT`.

Hints

- The detection part is the same as before, but this time, as we are using PnP to compute the pose, we need the points on the chessboard as 3D points: complete the function `calcChessboardCorners3D()` in `utility.cpp` that generates the 3D points.
- To track the points using the optical flow you can use the OpenCV function `calcOpticalFlowPyrLK()`
- As said above, remember to filter the points having a `status` set to 0: just skip those points when copying the new positions in `_corners` (*c.f.* the comments in the code).
- To compute the pose with the PnP algorithm you can use `mySolvePnPRansac` in `utility.cpp`, which is a wrapper of the original OpenCV function `solvePnPRansac()`: it is a simplified version which takes the same arguments, except for the rotation `rvec` and translation `tvec` that are merged in a single matrix (the pose matrix).

```cpp
/**
 * Wrapper around the original opencv's solvePnPRansac
 *
 * @param[in] objectPoints the 3D points
 * @param[in] imagePoints the image points
 * @param[in] cameraMatrix the calibration matrix
 * @param[in] distCoeffs the distortion coefficients
 * @param[out] poseMat the pose matrix
 * @param[out] the list of indices of the inliers points [optional]
 */
void mySolvePnPRansac(cv::InputArray objectPoints,
                                            cv::InputArray imagePoints,
                                            cv::InputArray cameraMatrix,
                                            cv::InputArray distCoeffs,
                                            cv::Mat &poseMat,
                                            cv::OutputArray inliers=cv::noArray())
```

9 Adding the OpenGL rendering

Now that we have built a reliable camera tracker, we can concentrate on the rendering part. We will use OpenGL to render the 3D object on top of the chessboard.

So far we have seen how to render a virtual object using the OpenGL pipeline. Now we have to render the virtual object over each frame of the video. Here is a simple way to do it: at each frame we draw a rectangle in front of the OpenGL camera and place on top of it the current frame as texture. This will just render the frame in an OpenGL window. This is our "background" image and on top of that we will draw the virtual object as usual using the information about its pose and orientation given by the camera tracker.

One question should arise: if we render a rectangle in front of the camera how can the camera "see" the virtual object if it is occluded by the rectangle itself? Here comes the trick: we can at first draw the rectangle with the texture of the current frame and then clear the *z-buffer* of the image (*i.e.* reset all the values) so that any object we draw next will overdrawn the relevant part of the frame.

You can have a look at the file `src/tp/videoOGL.cpp`. It reads a video file as input and it displays it in an window. You can compile it and run with, *e.g.*, the video of the chessboard.

If you look at the code you should recognize the familiar functions needed to render objects and textures in OpenGL. In particular:

- `glInit` does the initialization and it binds the integer `gCameraTextureId` to an OpenGL texture
- `updateTexture` is the function that constantly updates the texture `gCameraTextureId` with the current video frame contained in the OpenCV `Mat` object `gResultImage`.
- `drawBackground` is the function that actually draws the rectangle with the current frame texture in front of the OpenGL camera.
- In `displayFunc` you can see the call to the function `drawBackground` followed by the call to `glClear( GL_DEPTH_BUFFER_BIT )` that resets all the depth values of the pixels.
- Finally, in `main`, you can see that the OpenGL pipeline is a bit different than the usual: instead of calling `glutMainLoop()` to enter in the OpenGL processing loop, the program processes the video frames as done so far in OpenCV and for each frame it calls the function `glutMainLoopEvent()`. This GLUT function forces the call to the display callback that refreshes the window.

We can use this program as template to build the program that renders the virtual objects with OpenGL.

Assignment

Complete the file `src/tp/videoOGLTeapot.cpp`: first add the rendering of a virtual object in a fixed position of the image, and then t attach the camera tracker to estimate the correct pose of the object in the scene. The program takes the usual input parameters

```
Detect a chessboard in a given video and visualize a 3D object on top of it
Usage: videoOGLTeapotTP -w <board_width> -h <board_height> -c <calib file> <video file>
         -w <board_width>         # the number of inner corners per one of board dimension
         -h <board_height>        # the number of inner corners per another board dimension
         -c <calib file>          # the name of the calibration file
         <video file>             # the name of the video file
```

The main difference between this version and the previous one is the two global variables `gModelViewMatrix` and `gProjectionMatrix`: these are the two (pointers to the) matrices that contains the OpenGL model-view and projection matrices, respectively. In order to render the teapot we need to set, for each video frame, the model-view matrix for the virtual object: this is given by the camera pose estimated by the tracker. At the same time we also need the projection matrix that models the camera projection, which depends on the calibration parameters of the camera. We will see later how to work with these matrices.

First try to display a teapot over the image in fixed given position and then attach the camera tracking in order to render it in the proper position (*c.f.* Figure 10). Here some steps you can follow:

1. complete the `glInit()` function in order to enable/disable the depth test, the shading, the lights, the material for the teapot and the projection matrix.
2. complete the `displayFunc()` function: here you have to set the model-view matrix and draw the teapot. Remember to switch the lighting on and off when drawing the background and the teapot.
3. complete the `main()` function: you need to define the OpenGL projection matrix using the calibration parameters of the camera and the model-view matrix, for now with a default value for each frame.
4. now if you compile[2] and run the program you should see the teapot rendered over the video frames (*c.f.* Figure 10.(a)).
5. the last step is to attach the camera tracker to estimate the pose and render the teapot accordingly: add a camera tracker object (with KLT algorithm for the feature tracking), add the call to process inside the loop and assign the model-view matrix to the estimated camera pose matrix.

Hints

- So far, in order to set the projection and the model-view matrix we always applied some OpenGL transformations like `glRotatef()` or `gluPerspective()`. OpenGL offers another, more direct method to set the matrices. The function `glMultMatrixf()` (here the doc) allows to multiply the current transformation matrix by a specific transformation matrix. Since in our case the model-view and projection matrix are given already in the proper format we can set them by first setting them to the identity and then by multiplying them by the respective matrices.

[2]The first time you compile the code you will notice that another library will be downloaded and built. This is the GLM library we will use in the last exercise. Once it has done, you can run `cmake ..` again in your `build` directory to avoid it is compiled again every time you compile your source code.

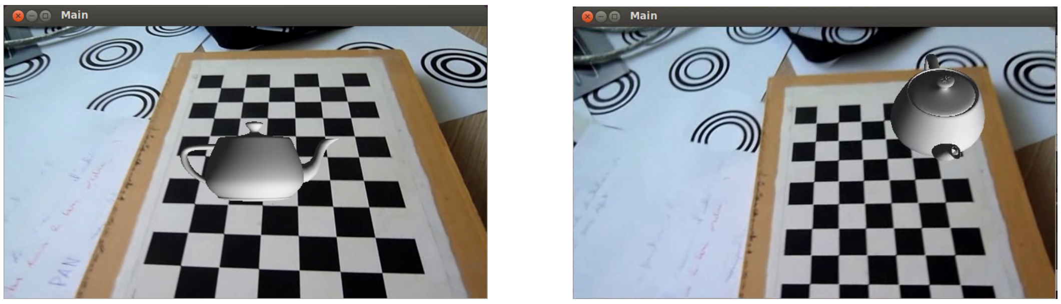


Figure 10: The teapot rendered just over the video frames in a fixed position (a), and then on top of the chessboard using the camera tracker (b).

- In order to set the projection matrix use the method `Camera::getOGLProjectionMatrix()` defined in `Camera` class: this allows to build an OpenGL projection matrix starting from the estimated internal parameters of the camera. For the near and far clipping plane you can use a value of 10 and 10000, respectively.

- In order to set the model-view matrix to a default value, you can find in `main` that a default "dummy" matrix is declared with some position values (`dummyMatrix`). We can make the `gModelViewMatrix` pointer point to the `data` field of this default matrix. It is not straightforward, thought: OpenGL manages matrices in different way than OpenCV. OpenGL uses a column-major order to store the matrix, *i.e.* the data is stored in the linear array column by column, so that the first element of the array corresponds to the first element of the first column, the second element corresponds to the second element of the first column and so on. OpenCV, instead, uses a row-major order to store data (the second element of the array is the second element of the first row and so on. . .). Therefore, in order to convert one system to another, one has to take the transpose version:
  - look at the code around line 410, you see how the conversion is made: the OpenCV matrix `dummyMatrix` is first transposed, cast to a `Mat` class, and then the field `data` passed to the pointer (with the casting to `float*`, for the correct data type).

10 Rendering an OBJ 3D object

The last step of this TP is to replace the usual OpenGL teapot with a generic 3D object loaded from an Wavefront OBJ file. You are already familiar with the OBJ format for storing a 3D model. For this TP we will use the open-source library GLM (the webpage) which provides a more complete interface to the OBJ format (*i.e.* it manages textures, materials *etc.*). You can find the library in `3rdparty/glm`[3].

The library provides a structure, `GLMmodel`, that contains all the data of the 3D model and the following functions (you can also check the available functions with the relevant documentation in `3rdparty/glm/build/include/glm.h`):

- `GLMmodel* glmReadOBJ(const char* filename)` loads the 3D models from the OBJ file and returns the pointer to the relevant structure.
- `GLfloat glmUnitize(GLMmodel* model)` is used to scale the model to the unit size, *i.e.* after the call to this function the model is scaled to be contained in a unitary cube.
- `GLvoid glmScale(GLMmodel* model, GLfloat scale)` scales the model to the given factor
- `GLvoid glmDraw(GLMmodel* model, GLuint mode)` draws the model using the relevant OpenGL primitives

Assignment

Modify `src/tp/videoOGLTeapot.cpp` to display an OBJ file that is load from file instead of the teapot. The filename for the OBJ can be passed with the `-o` option. The complete synopsis of `src/tp/videoOGLTeapot.cpp` is indeed:

```
Detect a chessboard in a given video and visualize a 3D object on top of it
Usage: videoOGLTeapotTP -w <board_width> -h <board_height> -c <calib file> <video file>

        -w <board_width>         # the number of inner corners per one of board dimension
        -h <board_height>        # the number of inner corners per another board dimension
        -c <calib file>          # the name of the calibration file
        [-o <obj file>]          # the optional obj file containing the 3D model to display
        <video file>             # the name of the video file
```

1. declare a global variable `model` as a pointer to `GLMmodel`: this will use through the code to load and display the 3D model
2. now we need to load the model: in `main`, before the processing loop load the model using `glmReadOBJ()`, the filename is in `objFile`. If any error happens when loading the file, the function will return `NULL`.
3. if the model is loaded, scale it to the unit size and then scale it to the same scale you used for the teapot. If the model is not loaded, just print a warning on the screen: we will display the teapot instead.

[3]The library will be downloaded, built and installed the first time you try to compile a source depending on it. Once it is done, you can run `cmake ..` again in your `build` directory, so that it won't be recompiled every time you compile your source code

4. we can now draw the model: in `displayFunc` check whether we have a model and then draw it, otherwise draw the teapot as usual.

5. before running the code you need to set the path to the static library libglm.so: in your shell you need to run the following command:
   ```
   export LD_LIBRARY_PATH=/home/<yourhome>/<path/to/code>/3dparty/glm/build/lib:$LD_LIBRARY_PATH
   ```
   This has to be done once, at the beginning of any shell session[4].

6. in `data/models` you can find some 3D model with textures to test the program with.

Hints

- Here is the documentation for `glmDraw` : you may want to pass `GLM_SMOOTH` and `GLM_TEXTURE` as drawing mode.

```
/* glmDraw: Renders the model to the current OpenGL context using the
 * mode specified.
 *
 * model    - initialized GLMmodel structure
 * mode     - a bitwise OR of values describing what is to be rendered.
 *            GLM_NONE    -  render with only vertices
 *            GLM_FLAT    -  render with facet normals
 *            GLM_SMOOTH  -  render with vertex normals
 *            GLM_TEXTURE -  render with texture coords
 *            GLM_FLAT and GLM_SMOOTH should not both be specified.
 */
GLvoid glmDraw(GLmodel* model, GLuint mode);
```

[4]Otherwise you can append the command at the end of your `~/.bashrc` file so that the path is automatically set every time you launch a new shell