Quick start with

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

Contents

1 About this document 2

2 The tutorial code 2
2.1 Building the code with CMake . . . 2
2.2 On your personal machine . . . 3

3 OpenCV 4

4 Mat - The Basic Image Container 5
4.1 Creating explicitly a Mat object . . . 6
4.1.1 Access to Mat elements . . . 8
4.2 Print out formatting . . . 9
4.3 Print for other common items . . . 10

5 Load and Display an Image 12
5.1 Goal . . . 12
5.2 Source Code . . . 12
5.3 Explanation . . . 13
5.4 Result . . . 14

6 Load, Modify, and Save an Image 15
6.1 Goals . . . 15
6.2 Code . . . 16
6.3 Explanation . . . 16
6.4 Result . . . 17

7 Load and display a video 19
7.1 Goals . . . 19
7.2 Code . . . 19
7.3 Explanation . . . 20
7.4 Result . . . 21

8 File Input and Output using XML and YAML files 23
8.1 Goal . . . 23
8.2 Source code . . . 23
8.3 Explanation . . . 24
8.4 Result . . . 24

9 Camera calibration With OpenCV **27**
9.1 Theory 27

1 About this document

This documents collects some of the tutorials that are part of the OpenCV library. These are the most basic yet relevant ones that will help you to get started with the library and complete the TP more easily. Some of them has been adapted, reduced or rewritten with the code you will have to write during the TP in mind. You can always access to the original tutorials and all the other ones from the online documentation that you can find at this address http://docs.opencv.org/2.4.6/doc/tutorials/tutorials.html.

2 The tutorial code

The archive containing the tutorials is organized as follows:

- `data` contains some images that can be used with the programs of this tutorial.
- `doc` contains a copy of this document, the official API reference guide of OpenCV and the original version of all OpenCV tutorials.
- `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

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.

From the starting path of the code, create a new directory, let's call it `build`:

```
mkdir build
cd build
```

This directory 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:

```
cmake ..
```

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

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` directory with the usual

```
make <filename_without_extension>
```

and

```
make clean
```

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

```
make mat_the_basic_image_container
```

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

```
./bin/mat_the_basic_image_container
```

2.2 On your personal machine

The code can be compiled in any machine of the computer rooms at ENSEEIHT. If you want to try it on your personal machine you have to first install the opencv library. The library can be downloaded from this address `http://opencv.org/downloads.html`. At this webpage you can find all the information to compile and install them on any supported platform.

Once you have installed them, you have just to follow the steps in Section 2.1, but you may need to do a

```
cmake .. -DOpenCV_DIR=path/to/OpenCVConfig.cmake/
```

in order to specify where the `OpenCVConfig.cmake` file is. In general it is located in the directory you used to build the library or in a system path, typically `/usr/local/share/OpenCV`, if you install them.

Finally, remember to set the `LD_LIBRARY_PATH` to allow your application to access the project libraries (for example if the OpenCV are not installed in the usual system path).

3 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality, etc.

It has C++, C, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. OpenCV leans mostly towards real-time vision applications and takes advantage of MMX and SSE instructions when available. A full-featured CUDA and OpenCL interfaces are being actively developed right now. There are over 500 algorithms and about 10 times as many functions that compose or support those algorithms. OpenCV is written natively in C++ and has a templated interface that works seamlessly with STL containers.

4 Mat - The Basic Image Container

`Mat` is basically a class having two data parts: the matrix header (containing information such as the size of the matrix, the method used for storing, at which address is the matrix stored and so on) and a pointer to the matrix containing the pixel values (may take any dimensionality depending on the method chosen for storing) . The matrix header size is constant. However, the size of the matrix itself may vary from image to image and usually is larger by order of magnitudes. Therefore, when you're passing on images in your program and at some point you need to create a copy of the image the big price you will need to build is for the matrix itself rather than its header. OpenCV is an image processing library. It contains a large collection of image processing functions. To solve a computational challenge most of the time you will end up using multiple functions of the library. Due to this passing on images to functions is a common practice. We should not forget that we are talking about image processing algorithms, which tend to be quite computational heavy. The last thing we want to do is to further decrease the speed of your program by making unnecessary copies of potentially *large* images.

To tackle this issue OpenCV uses a reference counting system. The idea is that each `Mat` object has its own header, however the matrix may be shared between two instance of them by having their matrix pointer point to the same address. Moreover, the copy operators **will only copy the headers**, and as also copy the pointer to the large matrix too, however not the matrix itself.

```
// creates just the header parts
Mat A, C;

// allocate the matrix from the image file
A = imread(argv[1], CV_LOAD_IMAGE_COLOR);

// Use the copy constructor
Mat B(A);

// Assignment operator
C = A;
```

All the above objects, in the end point to the same single data matrix. Their headers are different, however making any modification using either one of them will affect all the other ones too. In practice the different objects just provide different access method to the same underlying data. Nevertheless, their header parts are different. The real interesting part comes that you can create headers that refer only to a subsection of the full data. For example, to create a region of interest (*ROI*) in an image you just create a new header with the new boundaries:

```
// using a rectangle
Mat D (A, Rect(10, 10, 100, 100) );

// extracts A columns, from 1 (inclusive) to 3 (exclusive).
// like matlab E = A(:,2:3)
// columns and rows start from 0 in C/C++
Mat E = A(Range:all(), Range(1,3));
```

Here is another way to select and use columns and rows of the matrix

```
// add the 6th row, multiplied by 3 to the 4th row
M.row(3) = M.row(3) + M.row(5)*3;

// now copy the 8th column to the 2nd column
// M.col(1) = M.col(7); // this will not work!
Mat M1 = M.col(1);
M.col(7).copyTo(M1);
```

In order to select more than one column (row) you can use the `Mat` method `colRange(int start, int end)` (`rowRange(int start, int end)`). `start` is the index of first column/row to consider, `end` is the exclusive index of the last column/row, *i.e.* `colRange(1, 5)` will return the columns from the second (index 1) to the fifth (index 4).

Now you may ask if the matrix itself may belong to multiple `Mat` objects who will take responsibility for its cleaning when it's no longer needed. The short answer is: the last object that used it. For this a reference counting mechanism is used. Whenever somebody copies a header of a `Mat` object a counter is increased for the matrix. Whenever a header is cleaned this counter is decreased. When the counter reaches zero the matrix too is freed. Because, sometimes you will still want to copy the matrix itself too, there exists the clone() or the copyTo() function.

```
Mat F = A.clone();
Mat G;
A.copyTo(G);
```

Now modifying `F` or `G` will not affect the matrix pointed by the `Mat` header. What you need to remember from all this is that:

- Output image allocation for OpenCV functions is automatic (unless specified otherwise).
- No need to think about memory freeing with OpenCV C++ interface.
- The assignment operator and the copy constructor copies only the header.
- Use the `clone()` or the `copyTo()` function to copy the underlying matrix of an image.

4.1 Creating explicitly a `Mat` object

In the *Load, Modify, and Save an Image* tutorial you will see how to write a matrix to an image file by using the `imwrite` function. However, for debugging purposes it's much more convenient to see the actual values. You can achieve this via the << operator of `Mat`. However, be aware that this only works for two dimensional matrices.

Although `Mat` is a great class as image container it is also a general matrix class. Therefore, it is possible to create and manipulate multidimensional matrices. You can create a Mat object in multiple ways:

- Mat() Constructor

```
Mat M(2,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl << endl;
```

```
M =
 [0, 0, 255, 0, 0, 255;
  0, 0, 255, 0, 0, 255]
```

For two dimensional and multichannel images we first define their size: row and column count wise.

Then we need to specify the data type to use for storing the elements and the number of channels per matrix point. To do this we have multiple definitions made according to the following convention:

```
CV_[number of bits per item][Type Prefix]C[Channels]
```

For instance, `CV_8UC3` means we use unsigned char types that are 8 bit long and each pixel has three items of this to form the three channels. This are predefined for up to four channel numbers. The Scalar is four element short vector. Specify this and you can initialize all matrix points with a custom value. However if you need more you can create the type with the upper macro and putting the channel number in parenthesis as you can see below.

- Use C\C++ arrays and initialize via constructor

```
int sz[3] = {2,2,2};
Mat L(3,sz, CV_8UC(1), Scalar::all(0));
```

The upper example shows how to create a matrix with more than two dimensions. Specify its dimension, then pass a pointer containing the size for each dimension and the rest remains the same.

- `Create()` function:

```
M.create(4,4, CV_8UC(2));
cout << "M = "<< endl << " " << M << endl << endl;
```

```
M =
 [205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205]
```

You cannot initialize the matrix values with this construction. It will only reallocate its matrix data memory if the new size will not fit into the old one.

- MATLAB style initializer: `zeros()`, `ones()`, `eyes()`. Specify size and data type to use:

```
  Mat E = Mat::eye(4, 4, CV_64F);
  cout << "E = " << endl << " " << E << endl << endl;

  Mat O = Mat::ones(2, 2, CV_32F);
  cout << "O = " << endl << " " << O << endl << endl;

  Mat Z = Mat::zeros(3,3, CV_8UC1);
  cout << "Z = " << endl << " " << Z << endl << endl;
```

```
E =
 [1, 0, 0, 0;
  0, 1, 0, 0;
  0, 0, 1, 0;
  0, 0, 0, 1]
```

```
Z =
 [0, 0, 0;
  0, 0, 0;
  0, 0, 0]
```

```
O =
 [1, 1;
  1, 1]
```

- For small matrices you may use comma separated initializers:

```
1   Mat C = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
2   cout << "C = " << endl << " " << C << endl << endl;
```

```
C =
 [0, -1, 0;
  -1, 5, -1;
  0, -1, 0]
```

- Create a new header for an existing `Mat` object and `clone()` or `copyTo()` it.

```
1 Mat RowClone = C.row(1).clone();
2 cout << "RowClone = " << endl << " " << RowClone << endl << endl;
```

```
RowClone =
 [-1, 5, -1]
```

> **Note**: You can fill out a matrix with random values using the randu() function. You need to give the lower and upper value between what you want the random values:
>
> ```
> 1 Mat R = Mat(3, 2, CV_8UC3);
> 2 randu(R, Scalar::all(0), Scalar::all(255));
> ```

4.1.1 Access to `Mat` elements

In order to access to a single element of a matrix you can use its template method Mat::at():

```
1 template<typename T> T& Mat::at(int i, int j)
```

where the type `T` is the type of the matrix we are accessing, so it can be `float`, `double`, `int` *etc*. The indices `i` and `j` are the indices of the element (always 0-based). For example:

```cpp
// a 3x4 matrix of floats
Mat A = Mat(3, 4, CV_32FC1);

for(int i = 0; i < A.rows; ++i)
{
  for(int j = 0; j < A.cols; ++j)
  {
    // set the value of the element
    A.at<float>(i,j) = i*A.cols + j;
    // get the value of the element value
    cout << A.at<float>(i,j) << endl;
  }
}
```

4.2 Print out formatting

In the above examples you could see the default formatting option. Nevertheless, OpenCV allows you to format your matrix output format to fit the rules of:

- Default

```cpp
cout << "R (default) = " << endl << R << endl << endl;
```

```
R (default) =
[91, 2, 79, 179, 52, 205;
  236, 8, 181, 239, 26, 248;
  207, 218, 45, 183, 158, 101]
```

- Python

```cpp
cout << "R (python) = " << endl << R << endl << endl;
```

```
R (python)  =
[[[91, 2, 79], [179, 52, 205]],
  [[236, 8, 181], [239, 26, 248]],
  [[207, 218, 45], [183, 158, 101]]]
```

- Comma separated values (CSV)

```cpp
cout << "R (cvs) = " << endl << R << endl << endl;
```

```
R (csv)       =
91, 2, 79, 179, 52, 205
  236, 8, 181, 239, 26, 248
  207, 218, 45, 183, 158, 101
```

- Numpy

```
cout << "R (numpy) = " << endl << R << endl << endl;
```

```
R (numpy)    =
array([[[91, 2, 79], [179, 52, 205]],
  [[236, 8, 181], [239, 26, 248]],
  [[207, 218, 45], [183, 158, 101]]], type='uint8')
```

- C

```
cout << "R (c) = " << endl << R << endl << endl;
```

```
R (c)        =
(91, 2, 79, 179, 52, 205,
  236, 8, 181, 239, 26, 248,
  207, 218, 45, 183, 158, 101)
```

4.3 Print for other common items

OpenCV offers support for print of other common OpenCV data structures too via the << operator like:

- 2D Point

```
Point2f P(5, 1);
cout << "Point (2D) = " << P << endl << endl;
```

```
Point (2D) = [5, 1]
```

- 3D Point

```
Point3f P3f(2, 6, 7);
cout << "Point (3D) = " << P3f << endl << endl;
```

```
Point (3D) = [2, 6, 7]
```

- `std::vector` via `cv::Mat`

```
  vector<float> v;
  v.push_back( (float)CV_PI);
  v.push_back(2);
  v.push_back(3.01f);

  cout << "Vector of floats via Mat = " << Mat(v) << endl << endl;
```

```
Vector of floats via Mat = [3.1415927; 2; 3.01]
```

- `std::vector` of points

```
1  vector<Point2f> vPoints(20);
2  for (size_t i = 0; i < vPoints.size(); ++i)
3    vPoints[i] = Point2f((float)(i * 5), (float)(i % 7));
4
5  cout << "A vector of 2D Points = " << vPoints << endl << endl;
```

```
A vector of 2D Points = [0, 0; 5, 1; 10, 2; 15, 3; 20, 4; 25, 5; 30, 6; 35, 0; 4
0, 1; 45, 2; 50, 3; 55, 4; 60, 5; 65, 6; 70, 0; 75, 1; 80, 2; 85, 3; 90, 4; 95,
5]
```

Most of the samples here have been included into a small console application. You can download it from `here` or in the core section of the cpp samples.

5 Load and Display an Image

5.1 Goal

In this tutorial you will learn how to:

- Load an image (using imread)
- Create a named OpenCV window (using namedWindow)
- Display an image in an OpenCV window (using imshow)

5.2 Source Code

The source code is in `src/tutorials/display_image.cpp`. The program takes as input an image file and it diplays it in a window

```
1  # include <opencv2 / core / core.hpp>
2  # include <opencv2 / highgui / highgui.hpp>
3  # include <iostream>
4  using namespace cv;
5  using namespace std;
6  int main( int argc, char** argv )
7  {
8    if( argc != 2)
9    {
10     cout <<" Usage: display_image ImageToLoadAndDisplay" << endl;
11     return -1;
12   }
13   Mat image;
14   // Read the file
15   image = imread(argv[1], CV_LOAD_IMAGE_COLOR);
16   if(! image.data )
17   // Check for invalid input
18   {
19     cout << "Could not open or find the image" << std::endl ;
20     return -1;
21   }
22   // Create a window for display.
23   namedWindow( "Display window", CV_WINDOW_AUTOSIZE );
24   // Show our image inside it.
25   imshow( "Display window", image );
26   // Wait for a keystroke in the wind
27   waitKey(0);
28   return 0;
29 }
```

5.3 Explanation

In OpenCV 2 we have multiple modules. Each one takes care of a different area or approach towards image processing. You could already observe this in the structure of the user guide of these tutorials itself. Before you use any of them you first need to include the header files where the content of each individual module is declared.

You'll almost always end up using the:

- *core* section, as here are defined the basic building blocks of the library
- *highgui* module, as this contains the functions for input and output operations

```
# include <iostream>  // for standard I/O
# include <string>    // for strings
```

We also include the *iostream* to facilitate console line output and input. To avoid data structure and function name conflicts with other libraries, OpenCV has its own namespace: *cv*. To avoid the need appending prior each of these the *cv::* keyword you can import the namespace in the whole file by using the lines:

```
using namespace cv;
using namespace std;
```

This is true for the STL library too (used for console I/O). Now, let's analyze the *main* function. We start up assuring that we acquire a valid image name argument from the command line.

```
  if( argc != 2)
  {
    cout <<" Usage: display_image ImageToLoadAndDisplay" << endl;
    return -1;
  }
```

Then create a `Mat` object that will store the data of the loaded image.

```
    Mat image;
```

Now we call the imread function which loads the image name specified by the first argument (`argv[1]`). The second argument specifies the format in what we want the image. This may be:

- `CV_LOAD_IMAGE_UNCHANGED` (<0) loads the image as is (including the alpha channel if present)
- `CV_LOAD_IMAGE_GRAYSCALE` (0) loads the image as an intensity one
- `CV_LOAD_IMAGE_COLOR` (>0) loads the image in the RGB format

```
  // Read the file
  image = imread(argv[1], CV_LOAD_IMAGE_COLOR);
```

After checking that the image data was loaded correctly, we want to display our image, so we create an OpenCV window using the namedWindow function. These are automatically managed by OpenCV once you create them. For this you need to specify its name and how it should handle the change of the image it contains from a size point of view. It may be:

- `CV_WINDOW_AUTOSIZE` is the only supported one if you do not use the Qt backend. In this case the window size will take up the size of the image it shows. No resize permitted!
- `CV_WINDOW_NORMAL` on Qt you may use this to allow window resize. The image will resize itself according to the current window size. By using the | operator you also need to specify if you would like the image to keep its aspect ratio (`CV_WINDOW_KEEPRATIO`) or not (`CV_WINDOW_FREERATIO`).

```
1 // Create a window for display.
2 namedWindow( "Display window", CV_WINDOW_AUTOSIZE );
```

Finally, to update the content of the OpenCV window with a new image use the imshow function. Specify the OpenCV window name to update and the image to use during this operation:

```
1 // Show our image inside it.
2 imshow( "Display window", image );
```

Because we want our window to be displayed until the user presses a key (otherwise the program would end far too quickly), we use the waitKey function whose only parameter is just how long should it wait for a user input (measured in milliseconds). Zero means to wait forever.

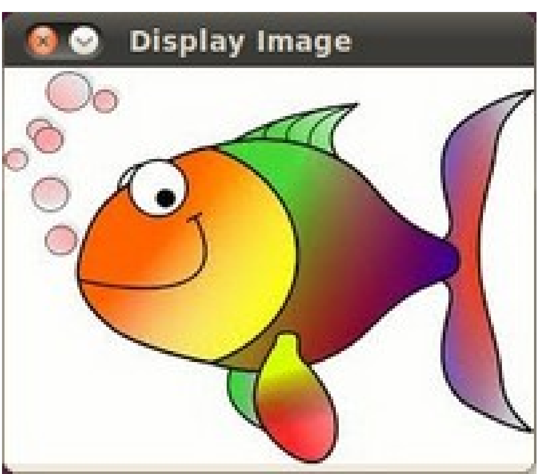
```
1 //Wait for a keystroke in the window
2 waitKey(0);
```

5.4 Result

- Compile your code and then run the executable giving an image path as argument. For exemple from the `build` directory:

  `./bin/display_image ../data/images/HappyFish.png`

- You should get a nice window as the one shown below:



6 Load, Modify, and Save an Image

Note: We assume that by now you know how to load an image using imread and to display it in a window (using imshow). Read the *Load and Display an Image* tutorial otherwise.

6.1 Goals

In this tutorial you will learn how to:

- Load an image using imread
- Transform an image from BGR to Grayscale format by using cvtColor
- Save your transformed image in a file on disk (using imwrite)

6.2 Code

The source code is in `src/tutorials/load_modify_image.cpp`. The program takes as input an image file, it converts it into a greylevel image, it displays the two images inside two separate windows and save the greylevel version into a file.

Here it is:

```
# include <cv.h>
# include <highgui.h>
using namespace cv;
int main( int argc, char** argv )
{
  char* imageName = argv[1];
  Mat image;
  image = imread( imageName, 1 );
  if( argc != 2 || !image.data )
  {
    printf( " No image data \n " );
    return -1;
  }
  Mat gray_image;
  cvtColor( image, gray_image, CV_BGR2GRAY );
  imwrite( "../../data/images/Gray_Image.jpg", gray_image );
  namedWindow( imageName, CV_WINDOW_AUTOSIZE );
  namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );
  imshow( imageName, image );
  imshow( "Gray image", gray_image );
  waitKey(0);
  return 0
}
```

6.3 Explanation

1. We begin by:
   - Creating a Mat object to store the image information
   - Load an image using imread, located in the path given by `imageName`. For this example, assume you are loading a RGB image.
2. Now we are going to convert our image from BGR to Grayscale format. OpenCV has a really nice function to do this kind of transformations:

```
cvtColor( image, gray_image, CV_BGR2GRAY );
```

   As you can see, cvtColor takes as arguments:
   - a source image (`image`)
   - a destination image (`gray_image`), in which we will save the converted image.

- an additional parameter that indicates what kind of transformation will be performed. In this case we use CV_BGR2GRAY (because of imread has BGR default channel order in case of color images).

3. So now we have our new gray_image and want to save it on disk (otherwise it will get lost after the program ends). To save it, we will use a function analagous to imread: imwrite

```
imwrite( "../../images/Gray_Image.jpg", gray_image );
```

   Which will save our gray_image as Gray_Image.jpg in the folder images located two levels up of my current location.

4. Finally, let's check out the images. We create two windows and use them to show the original image as well as the new one:

```
namedWindow( imageName, CV_WINDOW_AUTOSIZE );
namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );
imshow( imageName, image );
imshow( "Gray image", gray_image );
```

5. Add add the waitKey(0) function call for the program to wait forever for an user key press.

6.4 Result

When you run your program you should get something like this:

And if you check in your folder (in my case images), you should have a newly .jpg file named *Gray_Image.jpg*:

Congratulations, you are done with this tutorial!

7 Load and display a video

7.1 Goals

In this tutorial you will learn how to load a video from a file and visualize the stream in a OpenCV window.

7.2 Code

The code is in `src/tutorials/load_and_display_video.cpp`. The program takes as input the name of video and displays it inside a OpenCV window.

```cpp
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
using namespace cv;
using namespace std;
int main( int argc, char** argv )
{
  if( argc != 2)
  {
    cout <<" Usage: display_video VideoToLoadAndDisplay" << endl;
    return -1;
  }
  Mat image;
  // Read the file
  VideoCapture capture;
  capture.open( argv[1] );
  // Check if the video is loaded
  if(! capture.isOpened() )
  {
    cout << "Could not open or find the video" << std::endl ;
    return EXIT_FAILURE;
  }
  // Create a window for display.
  namedWindow( "Display window", CV_WINDOW_AUTOSIZE );
  // infinite loop to read all the frames iteratively
  while(1)
  {
    capture >> image;
    //check if there are still frames
    if( image.empty())
    {
      // exit the infinite loop
      break;
    }
    // Show our image inside it.
    imshow( "Display window", image );
    if(waitKey(20) == 'q')
      break;
  }
  return EXIT_SUCCESS;
}
```

7.3 Explanation

Essentially, all the functionalities required for video manipulation is integrated in the `VideoCapture` C++ class. A video is composed of a succession of images, we refer to these in the literature as frames.

The first task you need to do is to assign to a `VideoCapture` class its source. You can do this either via the constructor or its `open` function. If this argument is an integer then you will bind the class to a camera, a device. The number passed here is the ID of the device, assigned by the operating system. If you have a single camera attached to your system its ID will probably be zero and further ones increasing from there. If the parameter passed to these is a string it will refer to a video file, and the string points to the location and name of the file. In our case we pass the string which is contained in the vector of arguments of `main`:

```
VideoCapture capture;
capture.open( argv[1] );
```

If we wanted to use the constructor of the class to pass the string, the code would be simply

```
VideoCapture capture( argv[1] );
```

To check if the video (or the camera) has been successfully opened and loaded we can use the class function `isOpened`:

```
if(! capture.isOpened() )
{
  cout << "Could not open or find the video" << std::endl ;
  return EXIT_FAILURE;
}
```

The frames of the video are just simple images. Therefore, we just need to extract them from the `VideoCapture` object and put them inside a `Mat` one. The video streams are sequential. You may get the frames one after another by the read or the overloaded >> operator:

```
capture >> image;
```

Theis operation will store the new frame in `image`. If no frame could be acquired (either cause the video stream was closed or you got to the end of the video file), the object `image` will be empty. We can check this with a simple if:

```
if( image.empty())
{
  // exit the infinite loop
  break;
}
```

Once we got the new frame in `image` we can display it inside the window as usual:

```
// Show our image inside it.
imshow( "Display window", image );
```

7.4 Result

- Compile your code and then run the executable giving an image path as argument. For example from the directory `build` you can run:

```
./bin/load_and_display_video ../data/video/calib.avi
```

- You should get a window displaying the video as the one shown below:

8 File Input and Output using XML and YAML files

8.1 Goal

You'll find answers for the following questions:

- How to print and read OpenCV data structures using YAML or XML files?
- Usage of OpenCV data structures such as FileStorage.

8.2 Source code

You can find the code in the `src/tutorial/file_input_output.cpp`.

Here's a sample code of how to achieve all the stuff enumerated at the goal list. The program takes as input the name of XML file (*e.g.* `foo.xml`), it generates some random matrices and then save them to the given file. The code shows how to write OpenCV matrices to xml file and how then load the data from the generated xml file.

```
int main(int ac, char** av)
{
    string filename = av[1];
    { //write
        Mat R = Mat(3, 3, CV_8UC3);
        // fill the matrix with uniformly-distributed random values
        randu(R, Scalar::all(0), Scalar::all(255));

        Mat T = Mat(3, 1, CV_32FC3);
        // fill the matrix with normally distributed random values
        randn(T,  Scalar::all(0), Scalar::all(1));

        FileStorage fs(filename, FileStorage::WRITE);
        fs << "R" << R;                                         // cv::Mat
        fs << "T" << T;
        fs.release();                                          // explicit close
        cout << "Write Done." << endl;
    }

    {//read
        cout << endl << "Reading: " << endl;
        FileStorage fs;
        fs.open(filename, FileStorage::READ);
        if (!fs.isOpened())
        {
            cerr << "Failed to open " << filename << endl;
            help(av);
            return 1;
        }
        Mat R, T;
        fs["R"] >> R;                                        // Read cv::Mat
        fs["T"] >> T;
        cout << endl << "R = " << R << endl;
        cout << "T = " << T << endl << endl;
    }
    return 0;
}
```

8.3 Explanation

Here we talk only about XML and YAML file inputs. Your output (and its respective input) file may have only one of these extensions and the structure coming from this. They are two kinds of data structures you may serialize: *mappings* (like the STL map) and *element sequence* (like the STL vector>. The difference between these is that in a map every element has a unique name through what you may access it. For sequences you need to go through them to query a specific item.

1. **XML File Open and Close.** Before you write any content to such file you need to open it and at the end to close it. The XML data structure in OpenCV is `FileStorage`. To load or create a xml file you can use either its constructor or the `open()` function of this:

```
string filename = av[1];
FileStorage fs(filename, FileStorage::WRITE);
...
fs.open(filename, FileStorage::READ);
```

   Either one of this you use the second argument is a constant specifying the type of operations you'll be able to on them: `WRITE`, `READ` or `APPEND`. The extension specified in the file name also determinates the output format that will be used. The output may be even compressed if you specify an extension such as `.xml.gz`.

   The file automatically closes when the `FileStorage` objects is destroyed. However, you may explicitly call for this by using the *release* function:

```
fs.release();    // explicit close
```

2. **Input\Output of OpenCV Data structures.** In order to write a matrix, we use the operator << in a very similar way as we use it for the output operator of the standard C++ library (*e.g.*`cout « "Hello world!"`). For writing the data structure we need first to specify its name (it is not mandatory that it has the same name as the object).

```
  // write Mat
  fs << "R" << R;
  fs << "T" << T;
```

   Reading the data from the file is a simple addressing (via the `[ ]` operator) and read via the >> operator:

```
  // Read Mat
  fs["R"] >> R;
  fs["T"] >> T;
```

8.4 Result

Well mostly we just print out the defined numbers. On the screen of your console you could see:

```
Write Done.
```

```
Reading:

R = [91, 2, 79, 179, 52, 205, 236, 8, 181;
  239, 26, 248, 207, 218, 45, 183, 158, 101;
  102, 18, 118, 68, 210, 139, 198, 207, 211]
T = [-1.3660194, -0.063247398, 1.35166; 0.79910886, -0.26290667, -0.52100545; 1.8329793, 0.3


Tip: Open up out.xml with a text editor to see the serialized data.
```

Nevertheless, it's much more interesting what you may see in the output xml file:


```xml
<?xml version="1.0"?>
<opencv_storage>
  <R type_id="opencv-matrix">
    <rows>3</rows>
    <cols>3</cols>
    <dt>"3u"</dt>
    <data>
      91 2 79 179 52 205 236 8 181 239 26 248 207 218 45 183 158 101 102
      18 118 68 210 139 198 207 211
    </data>
  </R>
  <T type_id="opencv-matrix">
    <rows>3</rows>
    <cols>1</cols>
    <dt>"3f"</dt>
    <data>
      -1.36601937e+00 -6.32473975e-02 1.35166001e+00 7.99108863e-01
      -2.62906671e-01 -5.21005452e-01 1.83297932e+00 3.66539598e-01
      1.29202414e+00
    </data>
  </T>
</opencv_storage>
```


Or the YAML file:


```yaml
%YAML:1.0
R: !!opencv-matrix
   rows: 3
   cols: 3
   dt: "3u"
   data: [ 91, 2, 79, 179, 52, 205, 236, 8, 181, 239, 26, 248, 207, 218,
       45, 183, 158, 101, 102, 18, 118, 68, 210, 139, 198, 207, 211 ]
T: !!opencv-matrix
   rows: 3
   cols: 1
```

```
dt: "3f"
data: [ -1.36601937e+00, -6.32473975e-02, 1.35166001e+00,
    7.99108863e-01, -2.62906671e-01, -5.21005452e-01, 1.83297932e+00,
    3.66539598e-01, 1.29202414e+00 ]
```

9 Camera calibration With OpenCV

Cameras have been around for a long-long time. However, with the introduction of the cheap *pinhole* cameras in the late 20th century, they became a common occurrence in our everyday life. Unfortunately, this cheapness comes with its price: significant distortion. Luckily, these are constants and with a calibration and some remapping we can correct this. Furthermore, with calibration you may also determinate the relation between the camera's natural units (pixels) and the real world units (for example millimeters).

9.1 Theory

For the distortion OpenCV takes into account the radial and tangential factors. For the radial one uses the following formula:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k^3 r^6)$$
$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k^3 r^6)$$

So for an old pixel point at (x, y) coordinate in the input image, for a corrected output image its position will be $(x_{corrected} y_{corrected})$. The presence of the radial distortion manifests in form of the "barrel" or "fish-eye" effect.

Tangential distortion occurs because the image taking lenses are not perfectly parallel to the imaging plane. Correcting this is made via the formulas:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy]$$

So we have five distortion parameters, which in OpenCV are organized in a 5 column one row matrix:

$$Distortion_{\text{coefficients}} = [k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Now for the unit conversion, we use the following formula:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Here the presence of the w is cause we use a homography coordinate system (and $w = Z$). The unknown parameters are f_x and f_y (camera focal lengths) and (c_x, c_y) what are the optical centers expressed in pixels coordinates. If for both axes a common focal length is used with a given a aspect ratio (usually 1), then $f_y = f_x * a$ and in the upper formula we will have a single f focal length. The matrix containing these four parameters is referred to as the *camera matrix*. While the distortion coefficients are the same regardless of the camera resolutions used, these should be scaled along with the current resolution from the calibrated resolution.

The process of determining these two matrices is the calibration. Calculating these parameters is done by some basic geometrical equations. The equations used depend on the calibrating objects used. Currently OpenCV supports three types of object for calibration:

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