A Quick Introduction to OpenCV 2.2

The `OpenCV` library version 2.2 was released in 2010 and contains compared to earlier version a new API with more intuitive commands, often similar to `Matlab`. This article gives a brief introduction to the usage of this API for image processing and simple algebraic calculations on the new matrix datatype `Mat`.

June 2011, Konrad Wenzel, University of Stuttgart. konwen@gmx.de

1 Installation and Usage

For installation check out the Wiki of `OpenCV` on the Willowgarage Website:
`http://opencv.willowgarage.com/wiki/`
In the section *Downloads* you'll find the necessary files for your OS and the very useful *Install Guides*, e.g. for MS Visual Studio 2010.

During the setup process of your project you need to add the paths to the `OpenCV` include and library filder to your project. Also, the used `.lib` files must be defined. This is also explained in the *Install Guides*.

Very useful for the later work is also the *C++ Cheatsheat* on the same website, containing a list of all commands. The more detailed documentation and reference can be found here:
`http://opencv.willowgarage.com/documentation/cpp/index.html`

2 First steps

2.1 Read image + view it

To get started try to read an image into a matrix (type `Mat`) and view it:

```cpp
#include <opencv2/core/core.hpp>                // OpenCV core functions
#include <opencv2/highgui/highgui.hpp>          // visualization and image IO

using namespace cv;

int main(void)
{
        // Read image and write to matrix - here converted to grayimage by "0"
        Mat myimage = imread ( "myimage.png", 0 );
        // Show it in a window
        imshow ( "MyWindowTitle", myimage);
        // Wait for a keypress ( for x [ms], 0 means infinity )
        waitKey(0);
}
```

2.2 Apply blur

Now try to blur the image using the already implemented command `blur` which can be found along many other filters in the section *Image Filtering* in the documentation. If your IDE, like *Visual Studio* with *IntelliSense*, can list all commands of a certain namespace just enter `cv::` and select `blur`.

```
// Make new matrix
Mat myimage_blurred;
// Apply blur (box filter) with filtersize 3x3
cv:: blur ( myimage, myimage_blurred, cv::Size(3,3) );
// Show blurred image
imshow ( "Blurred", myimage_blurred );
waitKey(0);
```

Note: you can skip the `cv::` which represents the namespace and already included by `using namespace std;`, but it's helpful to mark that this is an OpenCV command.

2.3 Write image

To write the image to a file again, you can use the function `imwrite`:

```
// Write to bitmap file
imwrite ( "myimage_blurred.bmp", myimage_blurred );
```

3 Matrix data type 'Mat'

The `Mat` type is a container for matrices and supports different element data types. It contains several attributes, such as `Mat::rows` or `Mat::cols` representing the dimensions.

```
// Display the image dimensions
cout << "My␣image␣has␣the␣following␣dimensions:␣";
cout << myimage.rows << "␣rows␣by␣" << myimage.cols << "columns" << endl;
// Display the number of channels (should be one for a greyscale image)
cout << "The␣number␣of␣channels␣is:␣" << myimage.channels << endl;
// Check if image is 8 Bit unsigned character single channel
if ( myimage.depth() == CV_8U )
        cout<<"The␣element␣type␣is␣8␣Bit␣unsigned␣char" << endl;
```

3.1 Element data types

The following element datatypes are supported:

Depth	Typename	Range	OpenCV Name
8 Bit integer	unsigned char	0 ..256	CV_8U
8 Bit integer	(signed) char	-128 .. +128	CV_8S
16 Bit integer	unsigned short	0 .. 65000	CV_16U
16 Bit integer	(signed) short	-32768 .. +32768	CV_16S
32 Bit integer	(signed) int	-2147483648 .. +2147483647	CV_32S
32 Bit floating	float	1.5E-45 .. 3.4E38	CV_32F
64 Bit floating	double	5.0E-324 .. 1.7E308	CV_64F

Choose the depth depending on the application - for instance an usual 8 Bit image with integer values written into an 64 Bit double array consumes 8 times more memory. Most images available do only have an 8 Bit depth, except some special applications such as imagery from high dynamic range (HDR) cameras. While a greyscale image has only one channel, a color RGB image has 3 channels. The method `Mat::type` returns the resulting type, e.g. `CV_8UC3` for a 8 Bit 3 channel image.

The element data type can be converted from one type to another:

```
// Make new matrix and convert 8 Bit integer image to 32 Bit floating image
Mat myimage_floating;
myimage.convertTo ( myimage_floating, CV_32F );
```

Multi channel images can be converted to greyscale images using the function for converting color spaces:

```
Mat a = imread ( "mycolorimage.png" );  // RGB image, e.g. 8 Bit 3 channel
Mat a_grey;
cvtColor ( a, a_grey, CV_RGB2GRAY );    // a_grey is 8 Bit single channel
```

3.2 Matrix handling

Important: keep in mind, that the `Mat` datatype is only a header, containing several attributes such as the size and **only** the pointer to the first element of the data allocated in the heap memory. This means that the assign operator '=' is dangerous when you want to make a copy of the data, because the data itself is not copied but only the adress of the data.

At the beginning this seems to be confusing and not useful, but in fact it is very useful in order to save memory and increase speed in many applications. Also, *Regions of Interest* (ROI) in matrices can be assigned easily.

In order to assign an actual copy, the method `Mat::clone()` can be used:

```
// Get copy of matrix header
Mat b = myimage;
// Multiply all the data by 2 (see next section for operator)
b *= 2;
// => now myimage AND b were multiplied by 2!


// Better: make a copy of the header, copying the data also
Mat b = myimage.clone();
// Multiply data by 2
b *= 2;
// => now only b and not myimage was modified
// Alternatives
a.copyTo ( b );
a.convertTo ( b, a.depth() );
b = a * 200;            // calculations also enforce a copy, but be careful
```

3.3 Region of Interest (ROI)

In order to get just a subsection of an image, you can define a ROI matrix like this:

```
// Make new matrix
Mat a = Mat::ones ( 300, 300, CV_32F );
// Get rowrange and columnrange - count is zero-based and right exclusive
cv::Range rowrange ( 100, 201 ); // row 100:200 is included, row 201 not
cv::Range colrange ( 200, 251 );
// Get region of interest
Mat a_roi = a ( rowrange, colrange );
// Now, make a zero filled matrix with the size of the ROI
Mat a0 = Mat::zeros ( a_roi.size(), CV_32F );
// Copy the zeros to the ROI, and by that change a part of 'a' to zero
a0.copyTo ( a_roi );
```

3.4 Algebraic matrix operations

Similar to `Matlab`, `OpenCV` can perform algebraic operations:

```
// Matrix filled with ones ( also zeros and eye possible )
Mat a = Mat::ones ( 100, 50, CV_32F ); // 100 rows, 50 columns, floating
Mat b = Mat::ones ( 50, 100, CV_32F ); // 100 columns, 50 rows, floating

// Multiplication by a scalar
a = 255 * a;

// Substraction, addition of a scalar
a = a - 20;
b += 30;                 // same as b = b + 30

// Transpose
a = a.t();               // 50 rows, 100 columns now

// Addition of Matrices
Mat c = a + b;

// Element-wise multiplication
Mat d = a.mul ( b );     // is not a*b! similar to a.*b in Matlab

// Inversion
Mat n = (c.t() * c).inv();

// Logical operations
Mat m1 = b > a; // 8 Bit unsigned char mask, with 0 (false) and 256 (true)
```

3.5 Matrix element access

The easiest way to access a matrix element is the `Mat::at` method:

```
// Make new matrix
Mat a = Mat::ones ( 300, 300, CV_32F );
// Display element row 100, column 50 ( warning: zero based! )
cout << "Element␣(100,50)␣value:␣" << a.at<float>( 100, 50 );
```

The element datatype must be specified correctly, as listed in the table shown in subsection 3.1. Thus, a 32 Bit floating matrix, as used here, requires the element datatype `float` for an appropriate access. If the wrong type is specified, the function steps to the wrong adress in memory and does not return the true datatype. Usually an exception is thrown, which leads to a crash of your program.

The `Mat::at` function is quite slow if you want to use it very often in a row, since the position must be calculated each time. If you want to go through an whole image, you can implement a faster access using the method `Mat::ptr` like that:

```
Mat a = Mat::ones ( 300, 300, CV_32F );
// Go through all rows of the image ( 0 based index! )
for ( int r = 0; r < a.rows; r++ )
{
        // Get a pointer to the current row of the matrix element data type
        float * ptr_a = a.ptr<float> ( r );
        // Go through all columns of the image
        for ( int c = 0; c < a.cols; c++ )
                // Do something, eg. this multiplication
                ptr_a[c] = 100 * ptr_a[c];
}
```

3.6 Multi channel matrices

The elements of matrices having multiple channels are stored in a row, which means that for each pixel a value for each channel is assigned. For instance an RGB image has 3 values stored in a row for each pixel. Each pixel is again stored rowwise. In order to access the information in the green channel of a pixel, the function `Mat::ptr` is used (do **not** use `Mat::at`):

```
// Read RGB image
Mat a = imread ( "myrgbimage.png" );
// Access element - pointer to row 50 + column 100*3 channels + channel 2 (1,green)
unsigned char * pointer_img = a.ptr<unsigned char>(50) + 100*3 + 1;
cout << "Green␣channel␣at␣row␣50,␣column␣100):␣" << *pointer_img << endl;
```

3.6.1 Fill small matrices with data

The `OpenCV` API provides some very elegant ways to create and fill small matrices. First of all you can fill a matrix very easily like that:

```
// Make matrix
Mat a ( 3, 3, CV_64F );
// Fill it rowwise
a << 1, 2, 3, 4, 5, 6, 7, 8, 9;
// Leads to:    | 1 2 3 |
//              | 4 5 6 |
//              | 7 8 9 |
```

An easy way to create the same matrix from an existing array:

```
// Create array
double arr[9] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
// Create Mat with a data pointer to the array
Mat a ( 3, 3, CV_64F, (void*) arr );
```

Now the data pointer is replaced and you can even change values in the array to change the values in the matrix, as long you do not force the matrix to allocate an own memory as described in subsection 3.2.

3.6.2 Matrix element access using templates

In order to access matrices more easily you can use the template `Mat_<type>` type, where the element datatype is defined in the declaration and must not be specified again as in the functions `Mat::at<type>` or `Mat::ptr<type>`:

```
// Make template matrix of element type double
Mat_<double> a ( 3, 3 );
// Fill it with data
a << 1, 2, 1, 3, 2, 3, 4, 2, 4;
// Change the first element
a(0,0) = 10;    // same as a.at<double>(0,0) == 10
```

If the matrix only represents a vector where either only 1 column or 1 row exists, the matrix access can be simplyfied to:

```
// Make matrix with 1 column
Mat_<double> a = Mat_<double>::ones ( 3, 1 );
// Access the second element
cout << "The␣second␣element:␣" << a(1) << endl;
```