
The OpenCV Tutorials

Release 2.3

June 20, 2011

CONTENTS

1	Required packages	5
2	Getting OpenCV source code	7
2.1	Getting the latest stable OpenCV version	7
2.2	Getting the cutting-edge OpenCV from SourceForge SVN repository	7
3	Building OpenCV from source using CMake, using the command line	9
4	Installation in Windows	11
5	Using OpenCV with gcc and CMake	13
5.1	Steps	13
6	Using OpenCV with Eclipse (plugin CDT)	17
6.1	Prerequisites	17
6.2	Making a project	17
6.3	Running the executable	29
7	Display an Image	33
7.1	Goal	33
7.2	Code	33
7.3	Explanation	34
7.4	Result	34
8	Load and Save an Image	37
8.1	Goals	37
8.2	Code	37
8.3	Explanation	38
8.4	Result	39
9	Changing the contrast and brightness of an image!	41
9.1	Goal	41
9.2	Cool Theory	41
9.3	Code	42
9.4	Explanation	43
9.5	Result	44
10	Adding (blending) two images using OpenCV	45
10.1	Goal	45
10.2	Cool Theory	45

10.3	Code	45
10.4	Explanation	46
10.5	Result	47
11	Adding a Trackbar to our applications!	49
11.1	Goals	49
11.2	Code	49
11.3	Explanation	50
11.4	Result	51
12	Basic Drawing	53
12.1	Goals	53
12.2	OpenCV Theory	53
12.3	Code	54
12.4	Explanation	54
12.5	Result	57
13	Fancy Drawing!	59
13.1	Goals	59
13.2	Code	59
13.3	Explanation	59
13.4	Result	63
14	Smoothing Images	69
14.1	Goal	69
14.2	Cool Theory	69
14.3	Code	71
14.4	Explanation	72
14.5	Results	74
15	Eroding and Dilating	75
15.1	Goal	75
15.2	Cool Theory	75
15.3	Code	78
15.4	Explanation	80
15.5	Results	81

The following links describe a set of basic OpenCV tutorials. All the source code mentioned here is provide as part of the OpenCV regular releases, so check before you start copy & pasting the code. The list of tutorials below is automatically generated from reST files located in our SVN repository.

Note: YouTube videos yet to come...we have to think about them!

As always, we would be happy to hear your comments and receive your contributions on any tutorial.

- **INSTALLATION**

- *Installation in Linux*

	Title: Installation steps in Linux Compatibility: > OpenCV 2.0 We will learn how to setup OpenCV in your computer!
---	--

- *Installation in Windows*

	Title: Installation steps in Windows Compatibility: > OpenCV 2.0 You will learn how to setup OpenCV in your Windows Operating System!
---	---

- **USAGE AND COMPILATION**

- *Using OpenCV with gcc and CMake*

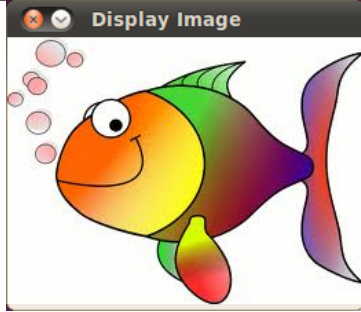
	Title: Using OpenCV with gcc (and CMake) Compatibility: > OpenCV 2.0 We will learn how to compile your first project using gcc and CMake
---	--

- *Using OpenCV with Eclipse (plugin CDT)*

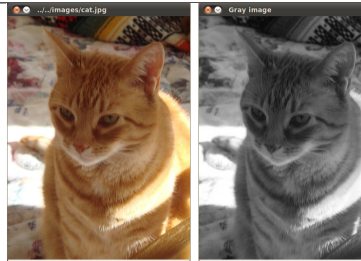
	Title: Using OpenCV with Eclipse (CDT plugin) Compatibility: > OpenCV 2.0 We will learn how to compile your first project using the Eclipse environment
---	---

- **BEGINNERS SECTION**

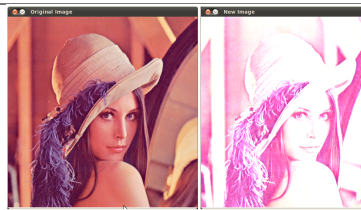
- *Display an Image*

	Title: Display an Image Compatibility: > OpenCV 2.0 We will learn how to display an image using OpenCV
---	--

- *Load and Save an Image*

	Title: Load and save an Image Compatibility: > OpenCV 2.0 We will learn how to save an Image in OpenCV...plus a small conversion to grayscale
---	---

- *Changing the contrast and brightness of an image!*

	Title: Changing the contrast and brightness of an image Compatibility: > OpenCV 2.0 We will learn how to change our image appearance!
---	---

- *Adding (blending) two images using OpenCV*

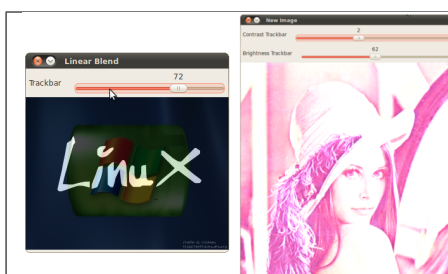


Title: Linear Blending

Compatibility: > OpenCV 2.0

We will learn how to blend two images!

– *Adding a Trackbar to our applications!*

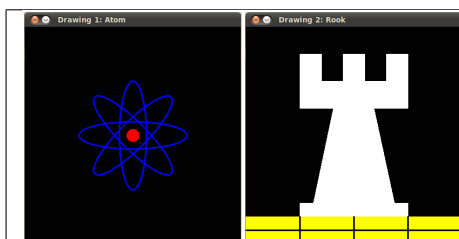


Title: Creating Trackbars

Compatibility: > OpenCV 2.0

We will learn how to add a Trackbar to our applications

– *Basic Drawing*

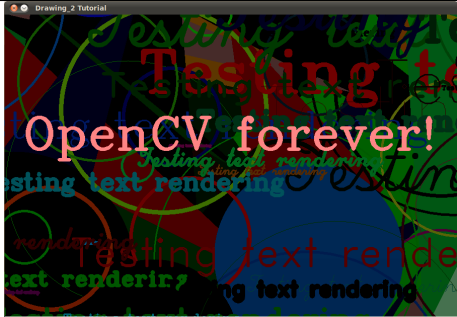


Title: Basic Drawing

Compatibility: > OpenCV 2.0

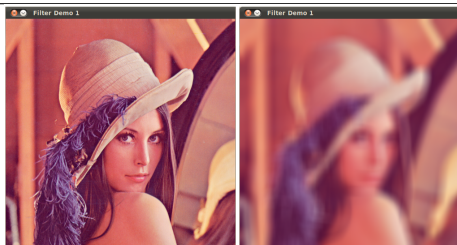
We will learn how to draw simple geometry with OpenCV!

– *Fancy Drawing!*

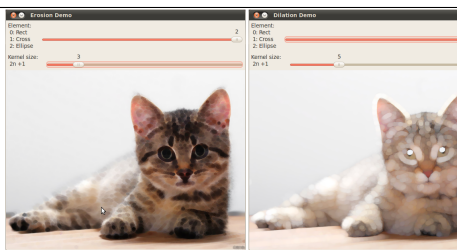
	Title: Cool Drawing Compatibility: > OpenCV 2.0 We will draw some <i>fancy-looking</i> stuff using OpenCV!
---	--

- **IMAGE PROCESSING**

- *Smoothing Images*

	Title: Smoothing Images Compatibility: > OpenCV 2.0 Let's take a look at some basic linear filters!
---	---

- *Eroding and Dilating*

	Title: Erosion and Dilation Compatibility: > OpenCV 2.0 Let's <i>change</i> the shape of objects!
---	---

These steps have been tested for Ubuntu 10.04 but should work with other distros.

REQUIRED PACKAGES

- GCC 4.x or later. This can be installed with

```
sudo apt-get install build-essential
```
- CMake 2.6 or higher
- Subversion (SVN) client
- GTK+2.x or higher, including headers
- pkgconfig
- libpng, zlib, libjpeg, libtiff, libjasper with development files (e.g. libjpeg-dev)
- Python 2.3 or later with developer packages (e.g. python-dev)
- SWIG 1.3.30 or later
- libavcodec
- libdc1394 2.x

All the libraries above can be installed via Terminal or by using Synaptic Manager

GETTING OPENCV SOURCE CODE

You can use the latest stable OpenCV version available in *sourceforge* or you can grab the latest snapshot from the SVN repository:

2.1 Getting the latest stable OpenCV version

- Go to <http://sourceforge.net/projects/opencvlibrary>
- Download the source tarball and unpack it

2.2 Getting the cutting-edge OpenCV from SourceForge SVN repository

Launch SVN client and checkout either

1. the current OpenCV snapshot from here: <https://code.ros.org/svn/opencv/trunk>
2. or the latest tested OpenCV snapshot from here: http://code.ros.org/svn/opencv/tags/latest_tested_snapshot

In Ubuntu it can be done using the following command, e.g.:

```
cd ~/<my_working_directory>
svn co https://code.ros.org/svn/opencv/trunk
```


BUILDING OPENCV FROM SOURCE USING CMAKE, USING THE COMMAND LINE

1. Create a temporary directory, which we denote as <cmake_binary_dir>, where you want to put the generated Makefiles, project files as well the object files and output binaries
2. Enter the <cmake_binary_dir> and type

```
cmake [<some optional parameters>] <path to the OpenCV source directory>
```

For example

```
cd ~/opencv
mkdir release
cd release
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX= /usr/local
```

3. Enter the created temporary directory (<cmake_binary_dir>) and proceed with:

```
make
sudo make install
```


INSTALLATION IN WINDOWS

For now this is just a stub article. It will be updated with valuable content as soon as possible. Make sure to check back for it!

USING OPENCV WITH GCC AND CMAKE

Note: We assume that you have successfully installed OpenCV in your workstation.

The easiest way of using OpenCV in your code is to use [CMake](#). A few advantages (taken from the Wiki):

- No need to change anything when porting between Linux and Windows
- Can easily be combined with other tools by CMake(i.e. Qt, ITK and VTK)

If you are not familiar with CMake, checkout the [tutorial](#) on its website.

5.1 Steps

Create a program using OpenCV

Let's use a simple program such as DisplayImage.cpp shown below.

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat image;
    image = imread( argv[1], 1 );

    if( argc != 2 || !image.data )
    {
        printf( "No image data \n" );
        return -1;
    }

    namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
    imshow( "Display Image", image );

    waitKey(0);
}
```

```
    return 0;
}
```

Create a CMake file

Now you have to create your CMakeLists.txt file. It should look like this:

```
project( DisplayImage )
find_package( OpenCV REQUIRED )
add_executable( DisplayImage DisplayImage )
target_link_libraries( DisplayImage ${OpenCV_LIBS} )
```

Generate the executable

This part is easy, just proceed as with any other project using CMake:

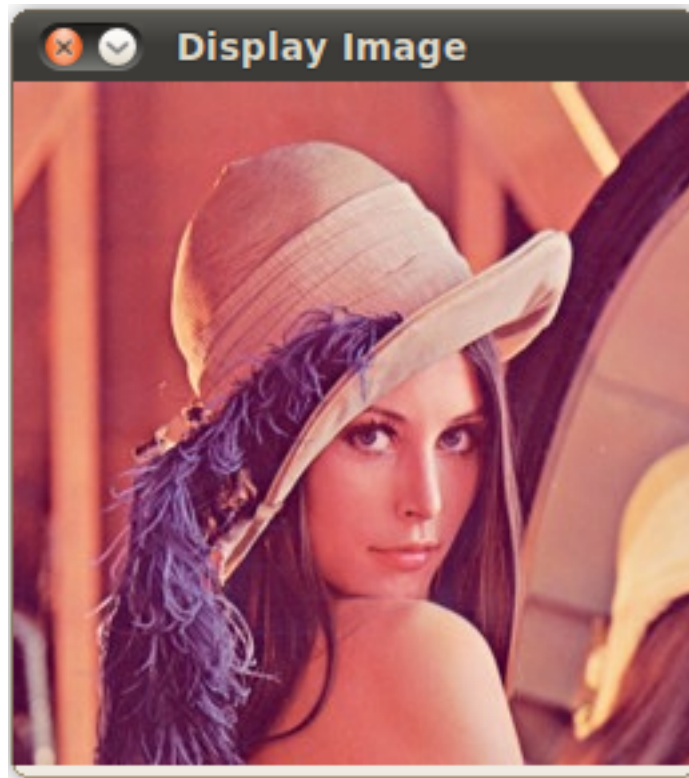
```
cd <DisplayImage_directory>
cmake .
make
```

Result

By now you should have an executable (called DisplayImage in this case). You just have to run it giving an image location as an argument, i.e.:

```
./DisplayImage lena.jpg
```

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



USING OPENCV WITH ECLIPSE (PLUGIN CDT)

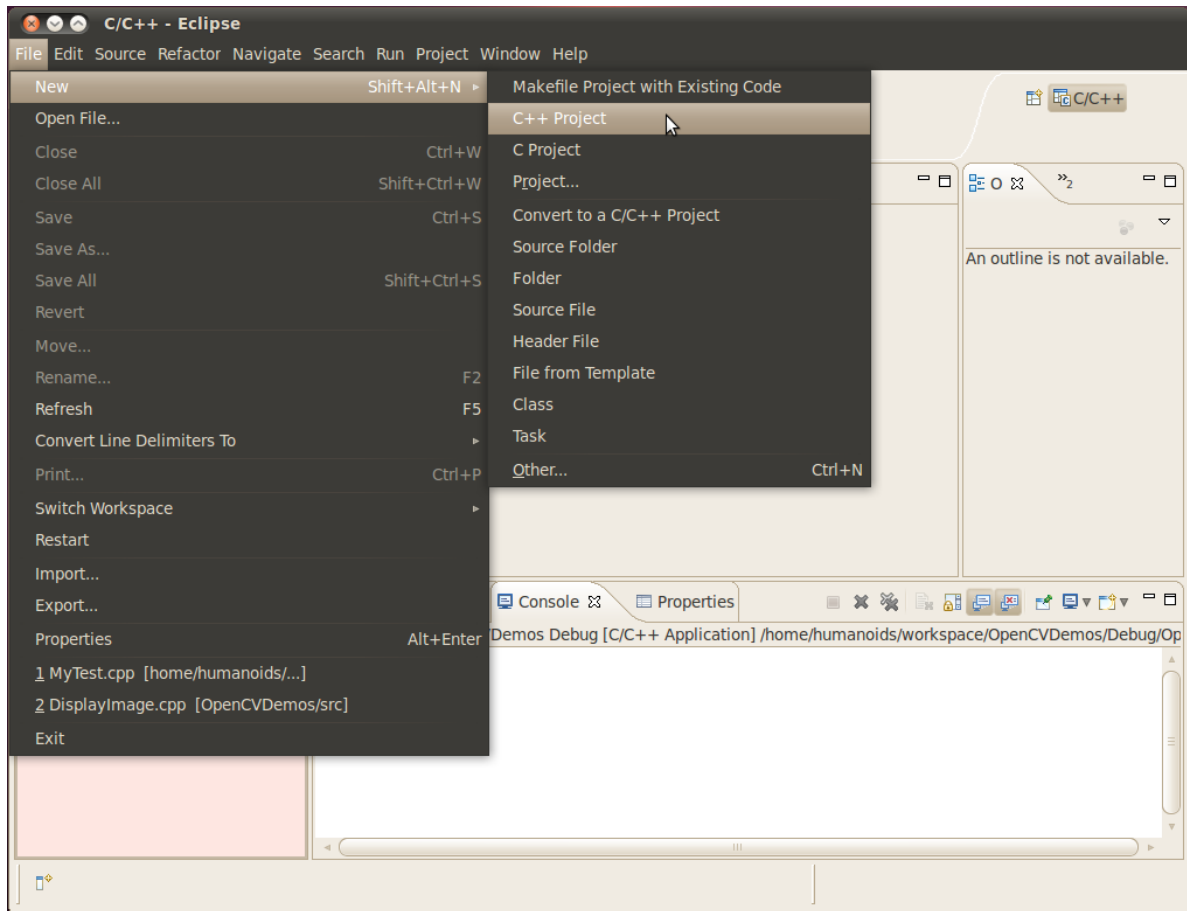
Note: For me at least, this works, is simple and quick. Suggestions are welcome

6.1 Prerequisites

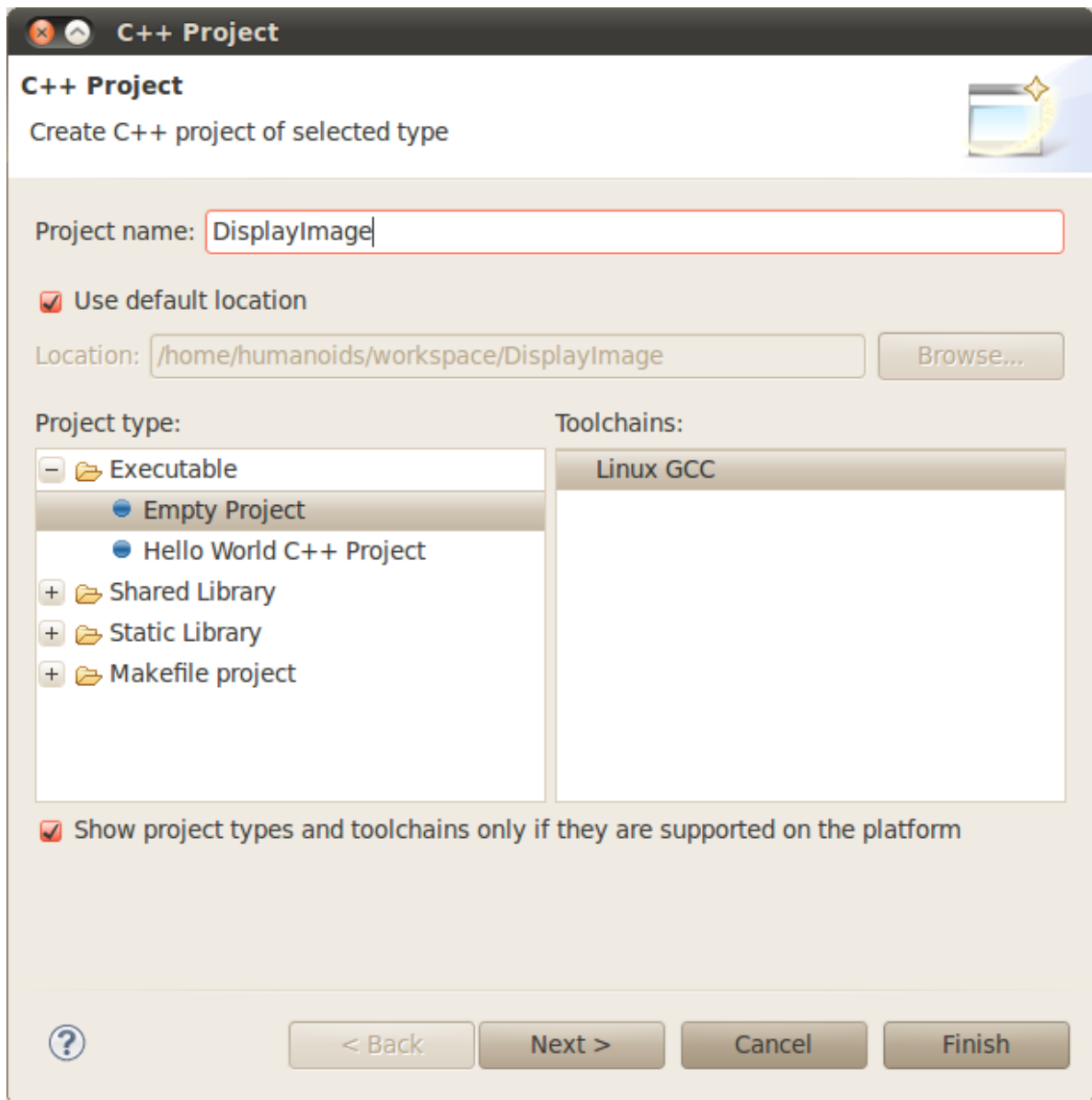
1. Having installed [Eclipse](#) in your workstation (only the CDT plugin for C/C++ is needed). You can follow the following steps:
 - Go to the Eclipse site
 - Download [Eclipse IDE for C/C++ Developers](#) . Choose the link according to your workstation.
2. Having installed OpenCV. If not yet, go [here](#)

6.2 Making a project

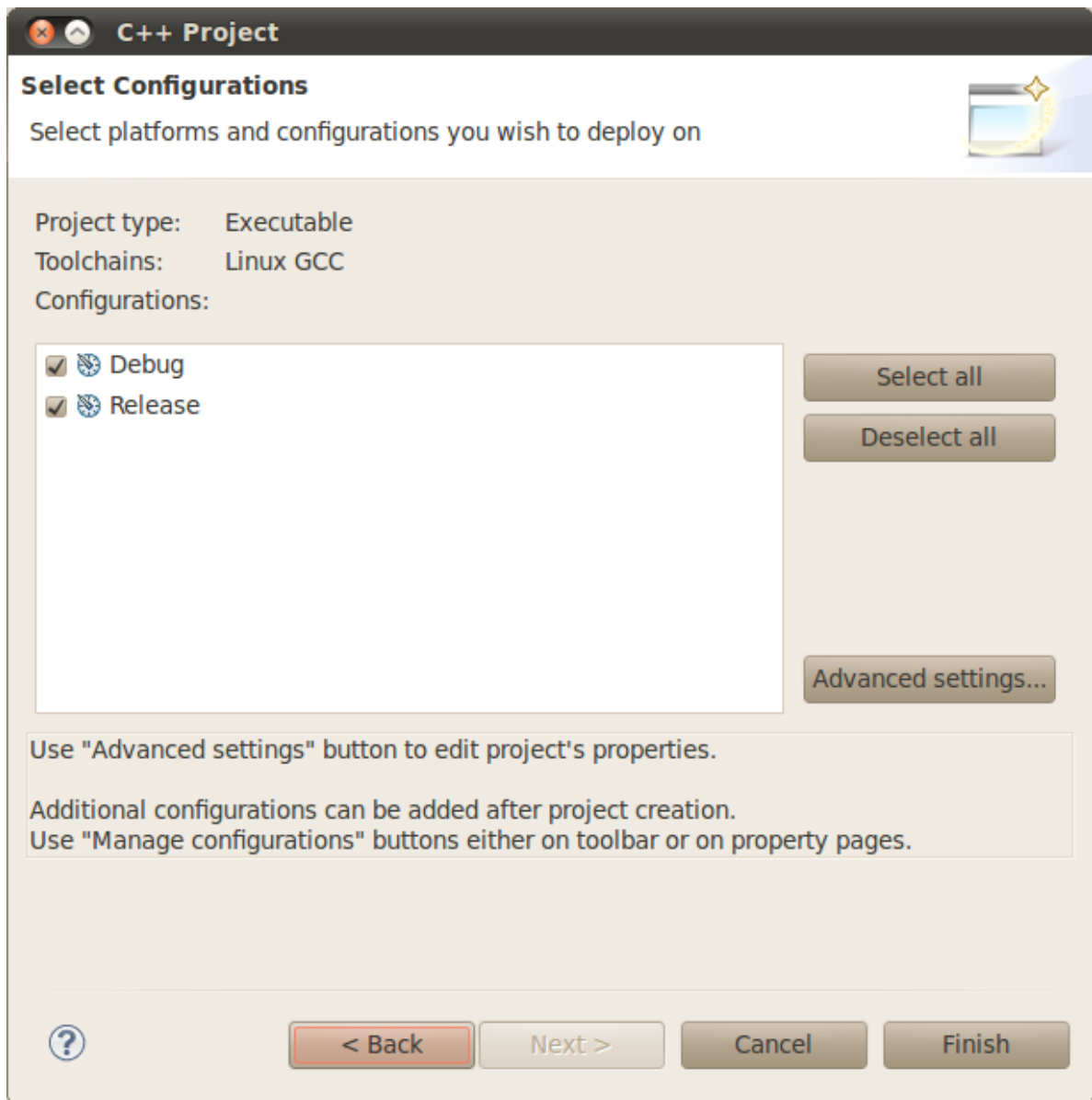
1. Start Eclipse. Just run the executable that comes in the folder.
2. Go to **File -> New -> C/C++ Project**



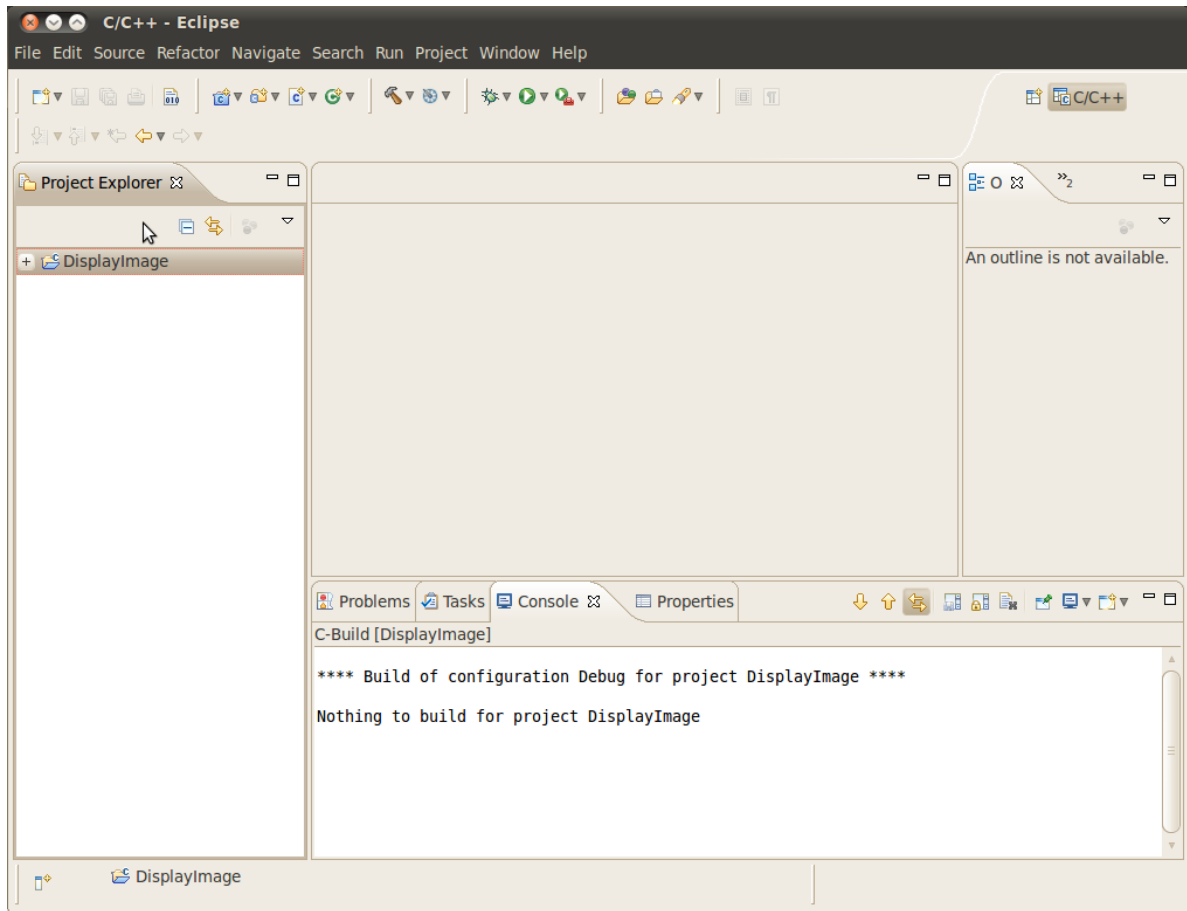
3. Choose a name for your project (i.e. DisplayImage). An **Empty Project** should be okay for this example.



4. Leave everything else by default. Press **Finish**.

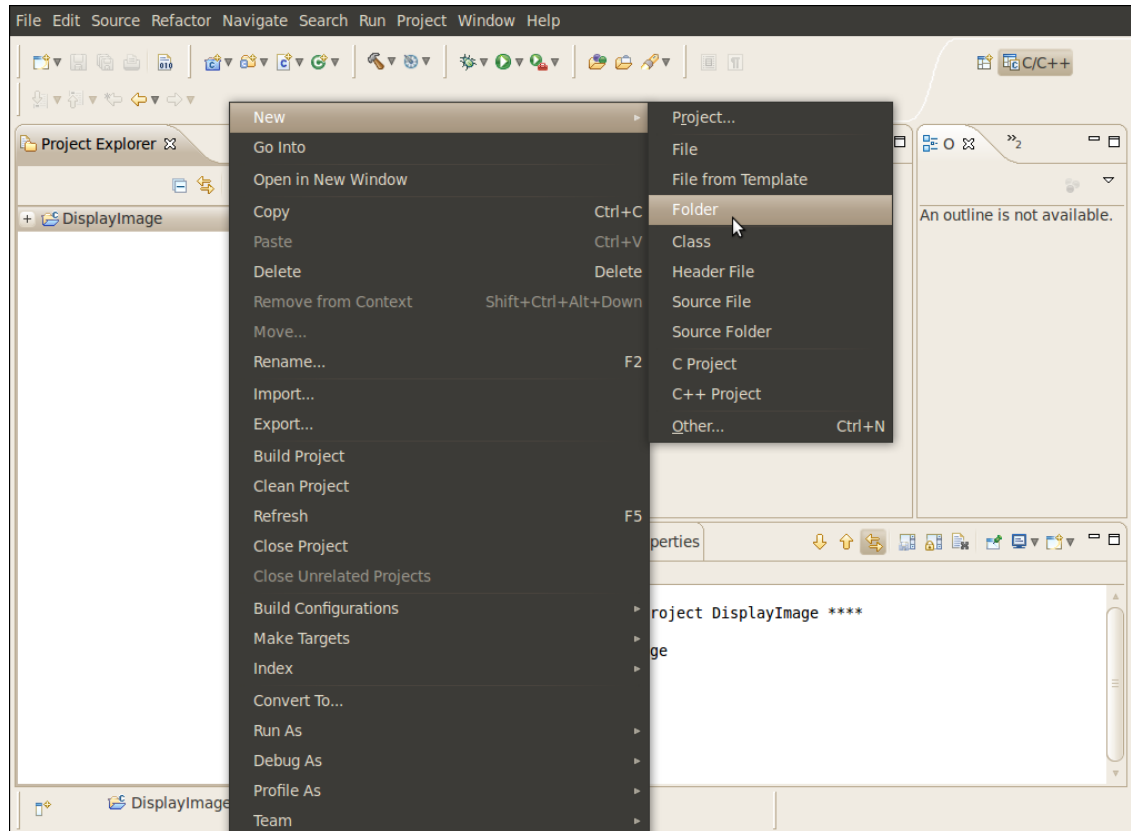


5. Your project (in this case DisplayImage) should appear in the **Project Navigator** (usually at the left side of your window).

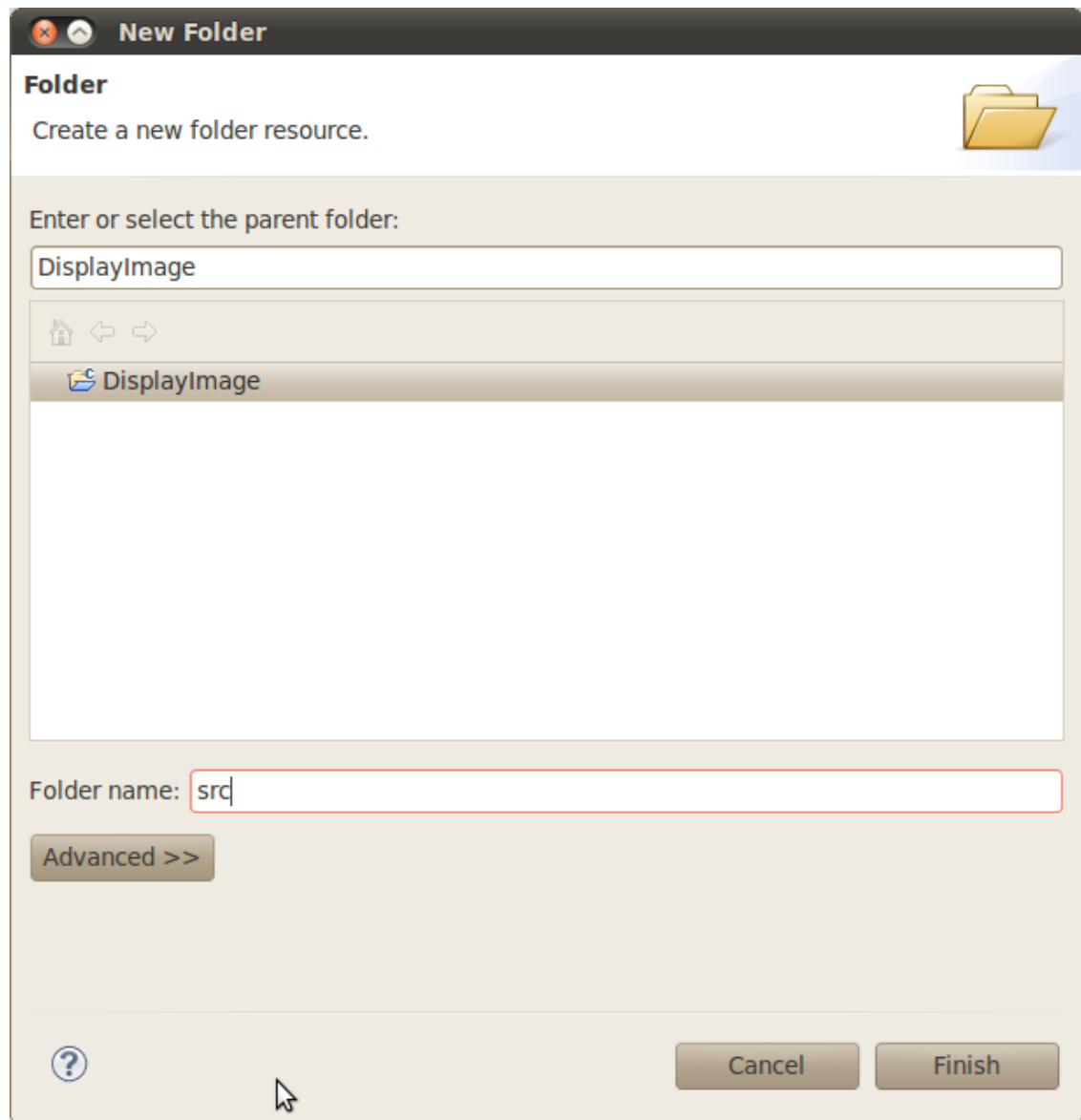


6. Now, let's add a source file using OpenCV:

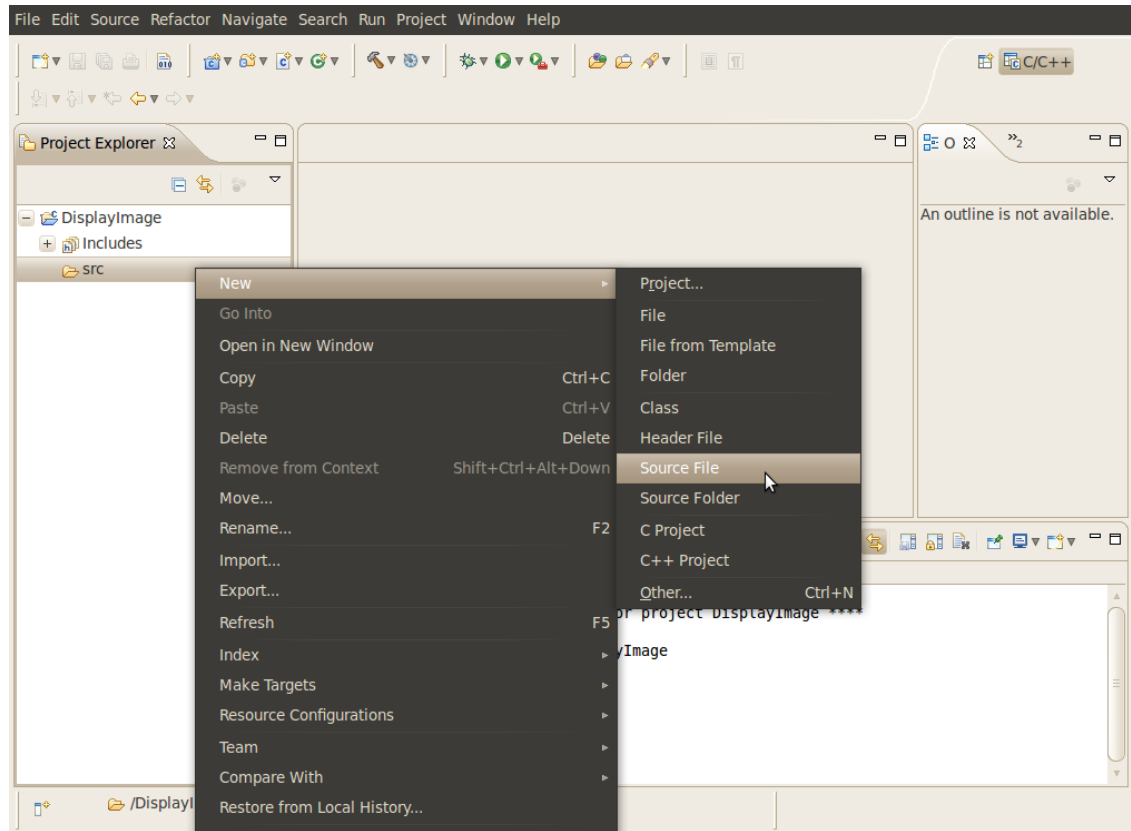
- Right click on **DisplayImage** (in the Navigator). **New -> Folder** .



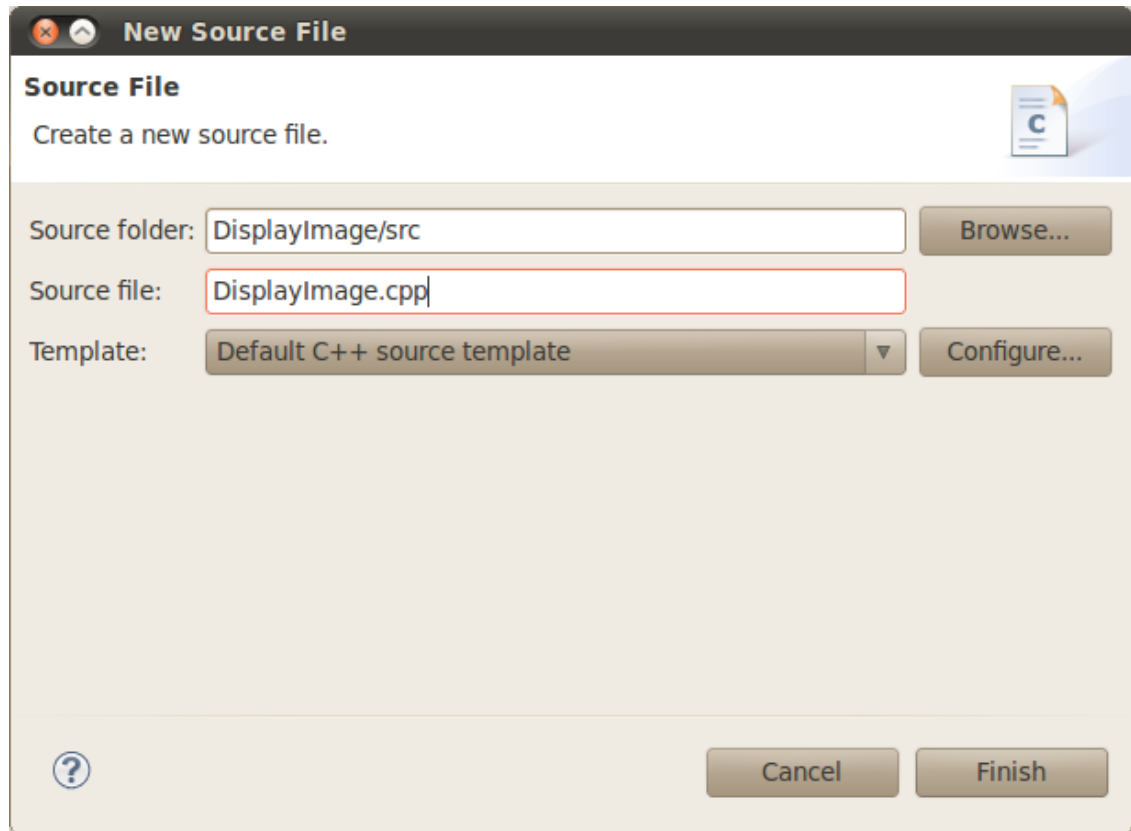
- Name your folder **src** and then hit **Finish**



- Right click on your newly created **src** folder. Choose **New source file**:



- Call it **DisplayImage.cpp**. Hit **Finish**



7. So, now you have a project with a empty .cpp file. Let's fill it with some sample code (in other words, copy and paste the snippet below):

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat image;
    image = imread( argv[1], 1 );

    if( argc != 2 || !image.data )
    {
        printf( "No image data \n" );
        return -1;
    }

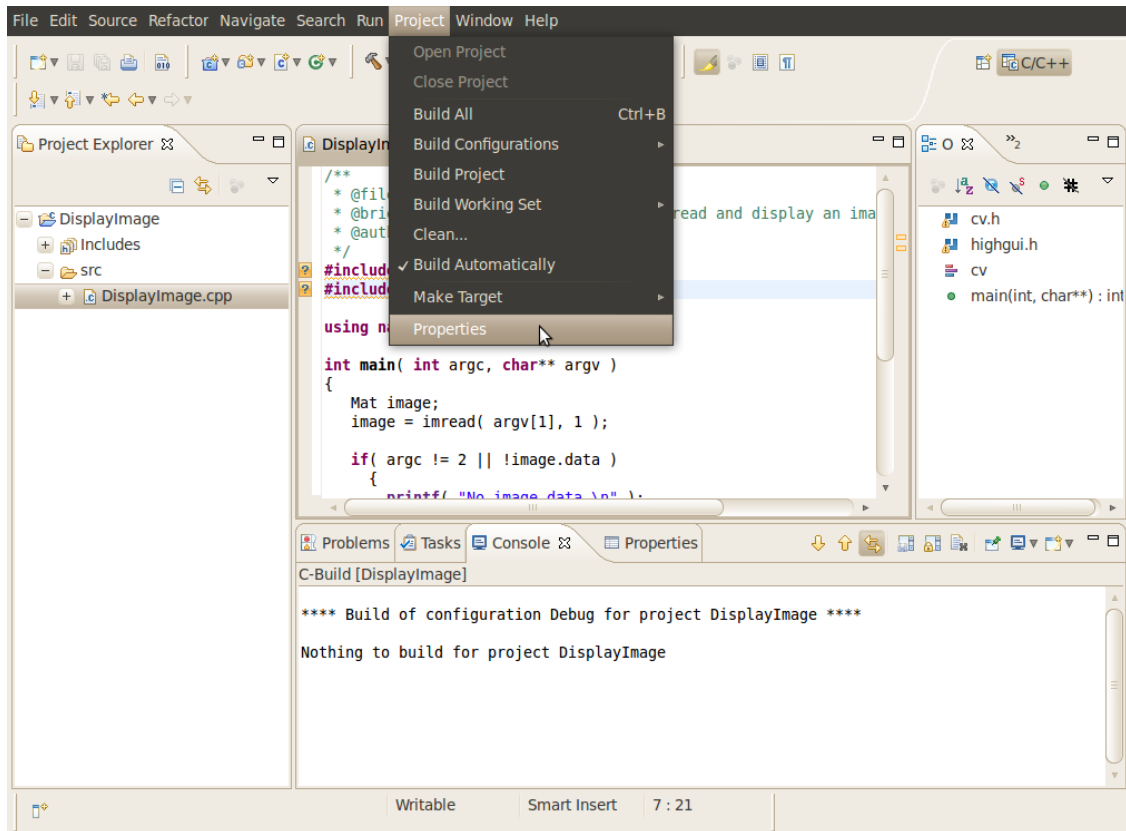
    namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
    imshow( "Display Image", image );

    waitKey(0);

    return 0;
}
```

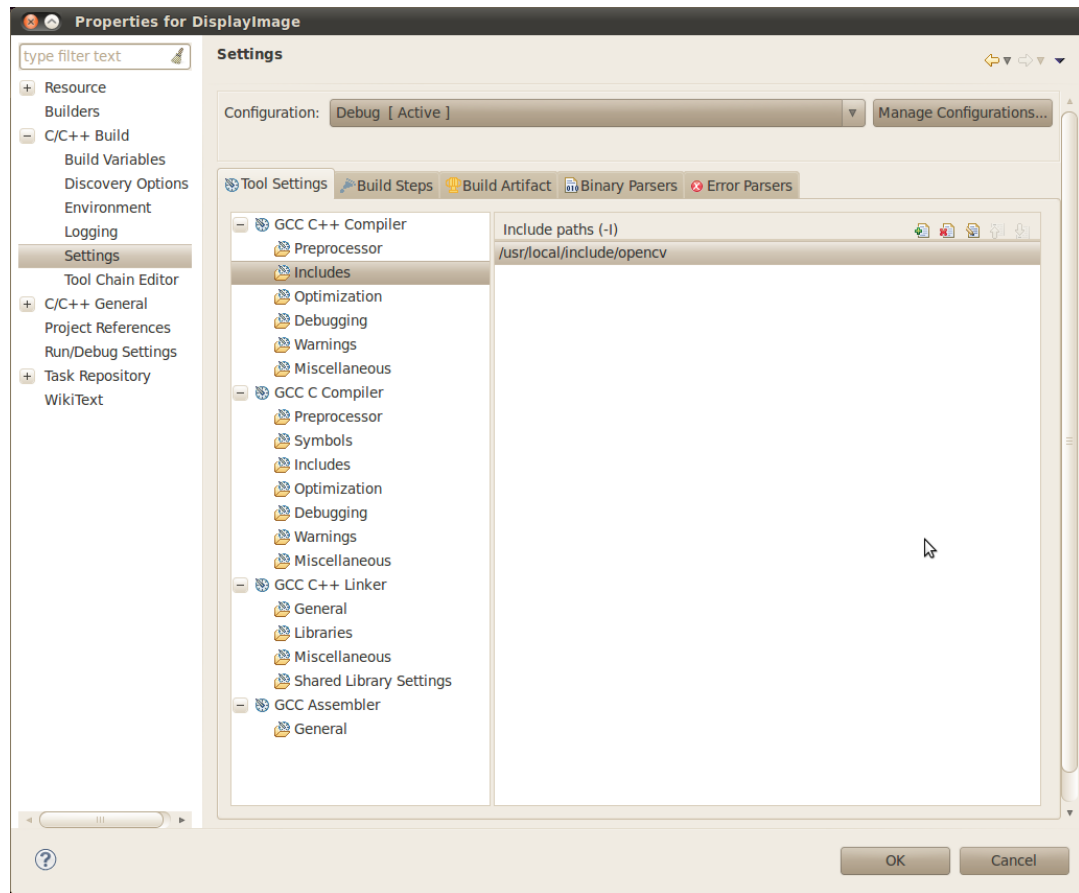
8. We are only missing one final step: To tell OpenCV where the OpenCV headers and libraries are. For this, do the following:

- Go to **Project**→**Properties**



- In **C/C++ Build**, click on **Settings**. At the right, choose the **Tool Settings** Tab. Here we will enter the headers and libraries info:
 - In **GCC C++ Compiler**, go to **Includes**. In **Include paths(-I)** you should include the path of the folder where opencv was installed. In our example, this is:

```
/usr/local/include/opencv
```



Note: If you do not know where your opencv files are, open the **Terminal** and type:

```
pkg-config --cflags opencv
```

For instance, that command gave me this output:

```
-I/usr/local/include/opencv -I/usr/local/include
```

– Now go to **GCC C++ Linker**, there you have to fill two spaces:

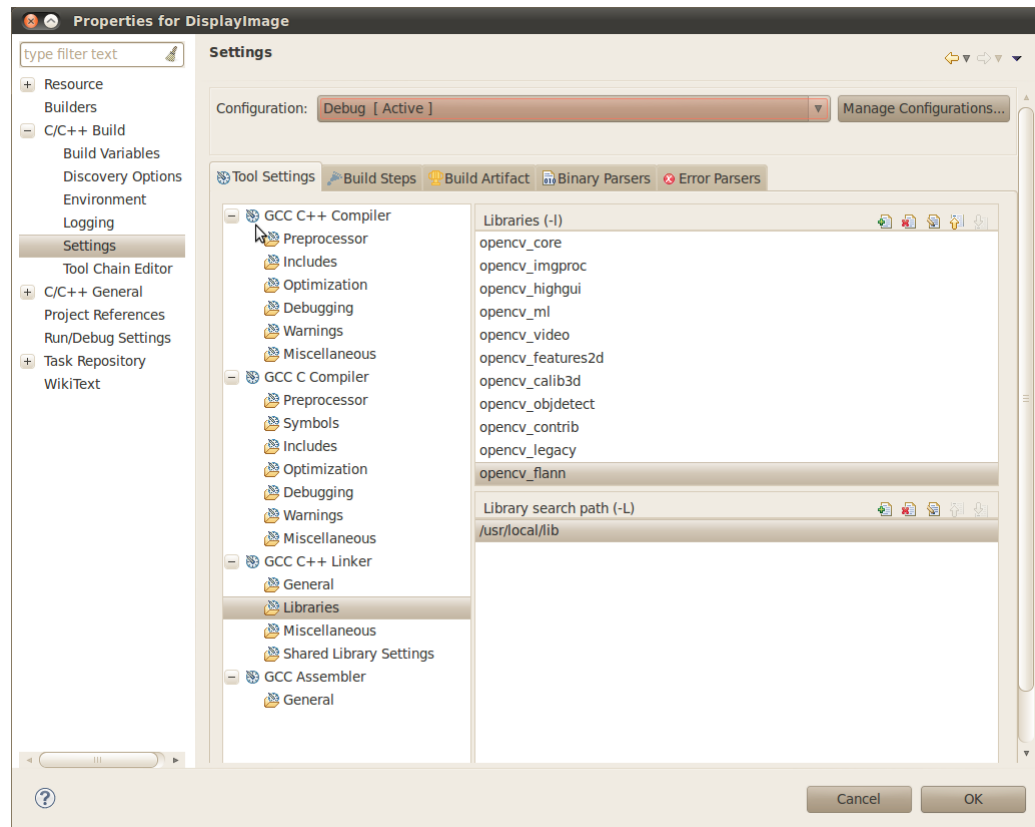
- * In **Library search path (-L)** you have to write the path to where the opencv libraries reside, in my case the path is:

```
/usr/local/lib
```

- * In **Libraries(-l)** add the OpenCV libraries that you may need. Usually just the 3 first on the list below are enough (for simple applications) . In my case, I am putting all of them since I plan to use the whole bunch:

- opencv_core
- opencv_imgproc
- opencv_highgui
- opencv_ml

- opencv_video
- opencv_features2d
- opencv_calib3d
- opencv_objdetect
- opencv_contrib
- opencv_legacy
- opencv_flann



Note: If you don't know where your libraries are (or you are just psychotic and want to make sure the path is fine), type in **Terminal**:

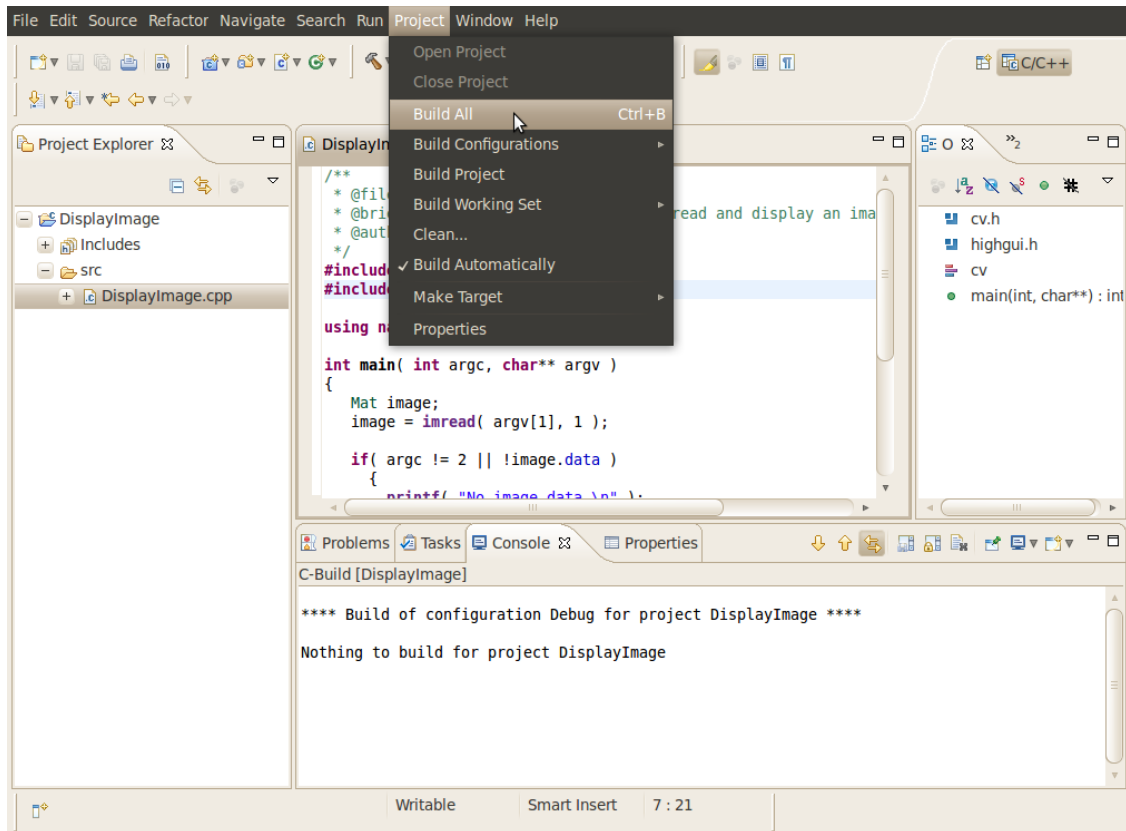
```
pkg-config --libs opencv
```

My output (in case you want to check) was:

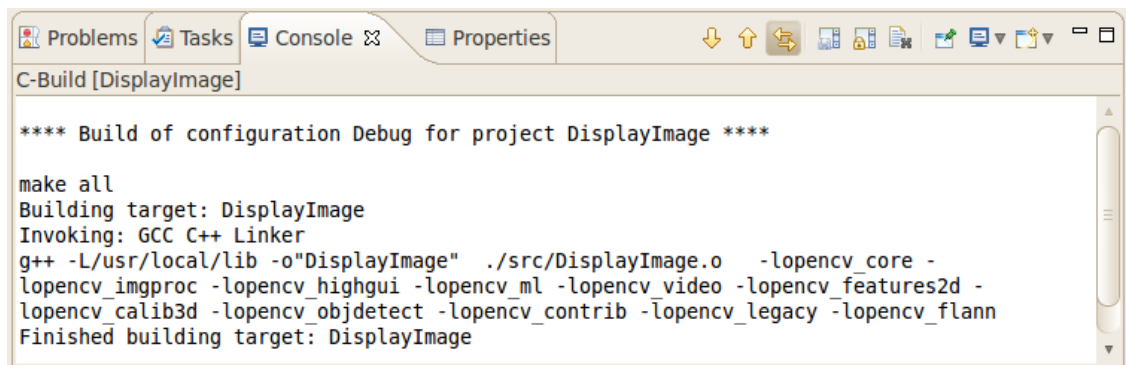
```
-L/usr/local/lib -lopencv_core -lopencv_imgproc \  
-lopencv_highgui -lopencv_ml -lopencv_video \  
-lopencv_features2d -lopencv_calib3d \  
-lopencv_objdetect -lopencv_contrib \  
-lopencv_legacy -lopencv_flann
```

Now you are done. Click **OK**

- Your project should be ready to be built. For this, go to **Project->Build all**



In the Console you should get something like



If you check in your folder, there should be an executable there.

6.3 Running the executable

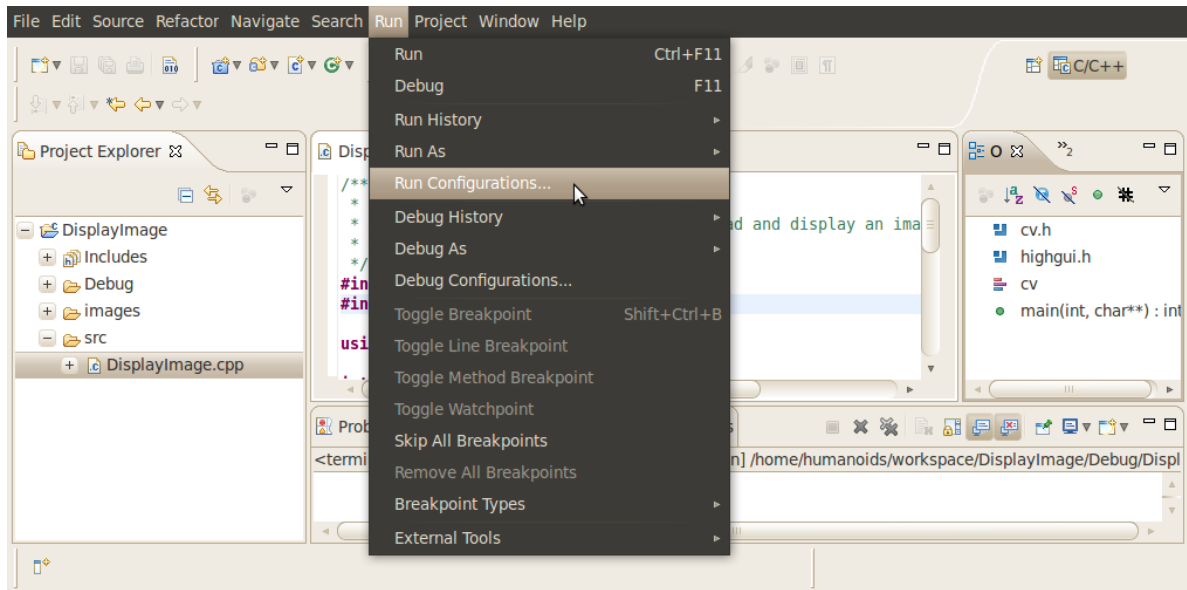
So, now we have an executable ready to run. If we were to use the Terminal, we would probably do something like:

```

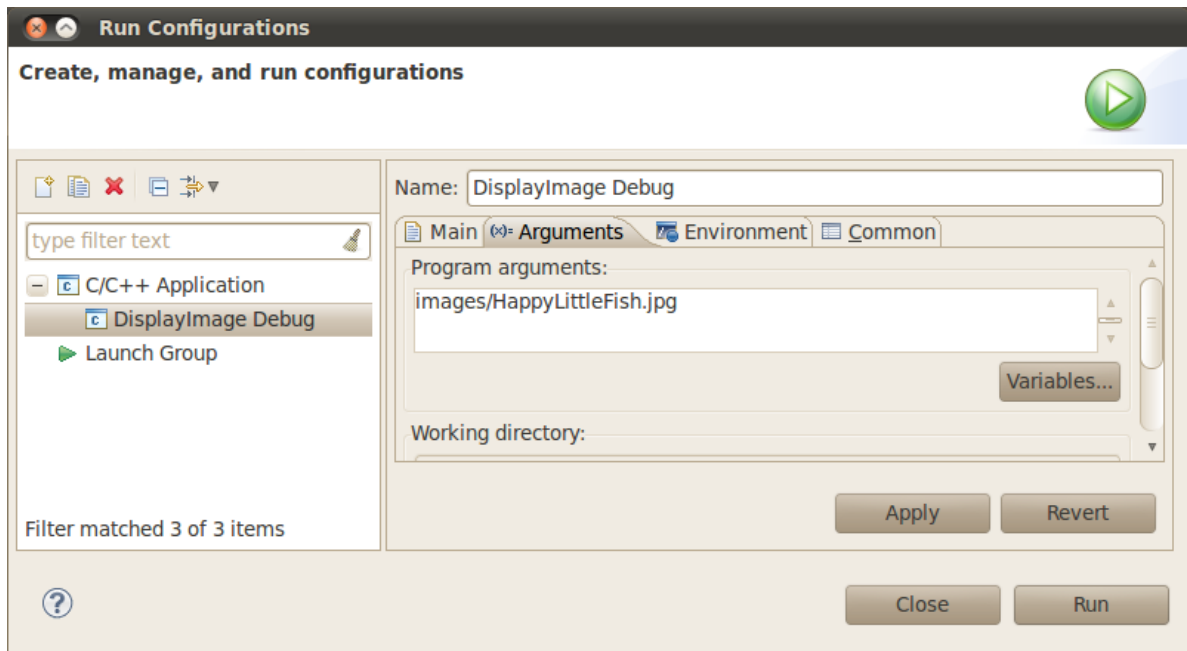
cd <DisplayImage_directory>
cd src
./DisplayImage ../images/HappyLittleFish.jpg
  
```

Assuming that the image to use as the argument would be located in `<DisplayImage_directory>/images/HappyLittleFish.jpg`. We can still do this, but let's do it from Eclipse:

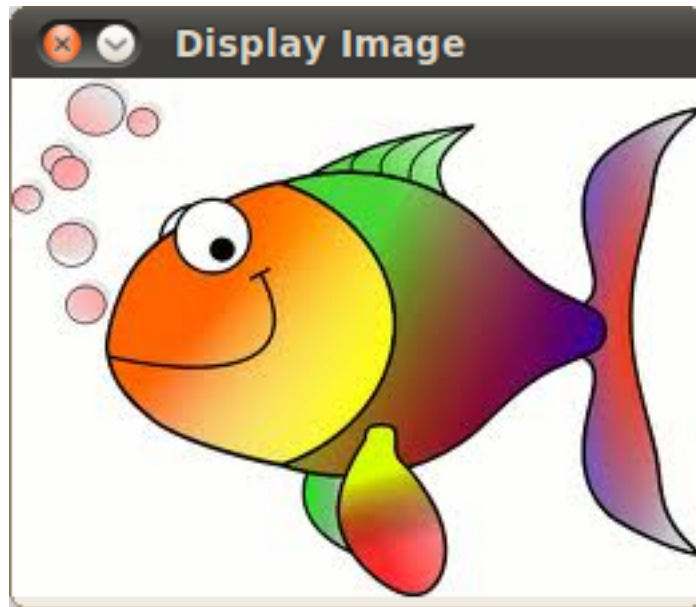
1. Go to **Run->Run Configurations**



2. Under C/C++ Application you will see the name of your executable + Debug (if not, click over C/C++ Application a couple of times). Select the name (in this case **DisplayImage Debug**).
3. Now, in the right side of the window, choose the **Arguments** Tab. Write the path of the image file we want to open (path relative to the workspace/DisplayImage folder). Let's use **HappyLittleFish.jpg**:



4. Click on the **Apply** button and then in Run. An OpenCV window should pop up with the fish image (or whatever you used).



5. Congratulations! You are ready to have fun with OpenCV using Eclipse.

DISPLAY AN IMAGE

7.1 Goal

In this tutorial you will learn how to:

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

7.2 Code

Here it is:

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

int main( int argc, char** argv )
{
    Mat image;
    image = imread( argv[1], 1 );

    if( argc != 2 || !image.data )
    {
        printf( "No image data \n" );
        return -1;
    }

    namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
    imshow( "Display Image", image );

    waitKey(0);

    return 0;
}
```

7.3 Explanation

1.

```
#include <cv.h>
#include <highgui.h>
```

```
using namespace cv;
```

These are OpenCV headers:

- *cv.h* : Main OpenCV functions
- *highgui.h* : Graphical User Interface (GUI) functions

Now, let's analyze the *main* function:

2.

```
Mat image;
```

We create a *Mat* object to store the data of the image to load.

3.

```
image = imread( argv[1], 1 );
```

Here, we called the function *imread* which basically loads the image specified by the first argument (in this case *argv[1]*). The second argument is by default.

4. After checking that the image data was loaded correctly, we want to display our image, so we create a window:

```
namedWindow( "Display Image", CV_WINDOW_AUTOSIZE );
```

namedWindow receives as arguments the window name ("Display Image") and an additional argument that defines windows properties. In this case **CV_WINDOW_AUTOSIZE** indicates that the window will adopt the size of the image to be displayed.

5. Finally, it is time to show the image, for this we use *imshow*

```
imshow( "Display Image", image )
```

6. Finally, we want our window to be displayed until the user presses a key (otherwise the program would end far too quickly):

```
waitKey(0);
```

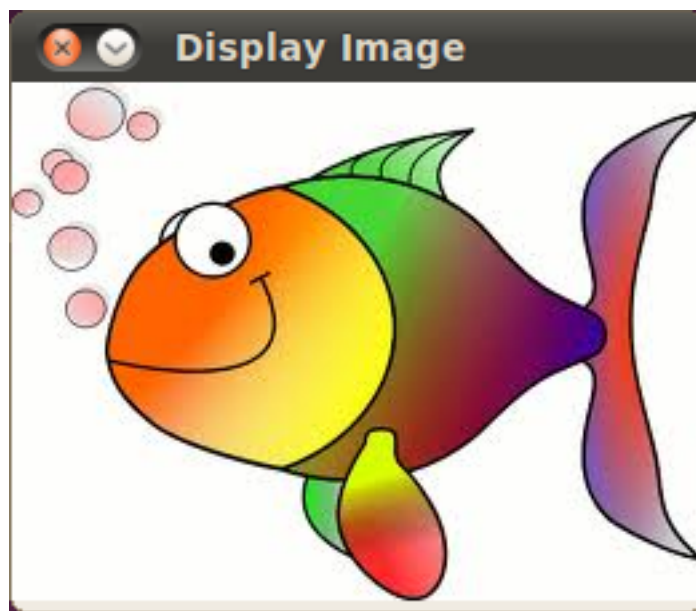
We use the *waitKey* function, which allow us to wait for a keystroke during a number of milliseconds (determined by the argument). If the argument is zero, then it will wait indefinitely.

7.4 Result

- Compile your code and then run the executable giving a image path as argument:

```
./DisplayImage HappyFish.jpg
```

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



LOAD AND SAVE AN IMAGE

Note: We assume that by now you know:

- Load an image using `imread`
 - Display an image in an OpenCV window (using `imshow`)
-

8.1 Goals

In this tutorial you will learn how to:

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

8.2 Code

Here it is:

```
1  #include <cv.h>
2  #include <highgui.h>
3
4  using namespace cv;
5
6  int main( int argc, char** argv )
7  {
8      char* imageName = argv[1];
9
10     Mat image;
11     image = imread( imageName, 1 );
12
13     if( argc != 2 || !image.data )
14     {
15         printf( " No image data \n " );
16         return -1;
17     }
18
19     Mat gray_image;
20     cvtColor( image, gray_image, CV_RGB2GRAY );
21
```

```
22  imwrite( "../images/Gray_Image.png", gray_image );
23
24  namedWindow( imageName, CV_WINDOW_AUTOSIZE );
25  namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );
26
27  imshow( imageName, image );
28  imshow( "Gray image", gray_image );
29
30  waitKey(0);
31
32  return 0;
33 }
```

8.3 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 RGB to Grayscale format. OpenCV has a really nice function to do this kind of transformations:

```
cvtColor( image, gray_image, CV_RGB2GRAY );
```

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.

And an additional parameter that indicates what kind of transformation will be performed. In this case we use `CV_RGB2GRAY` (self-explanatory).

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 analogous to `imread`: `imwrite`

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

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

4. Finally, let's check out the images. We create 02 windows and use them to show the original image as well as the new one:

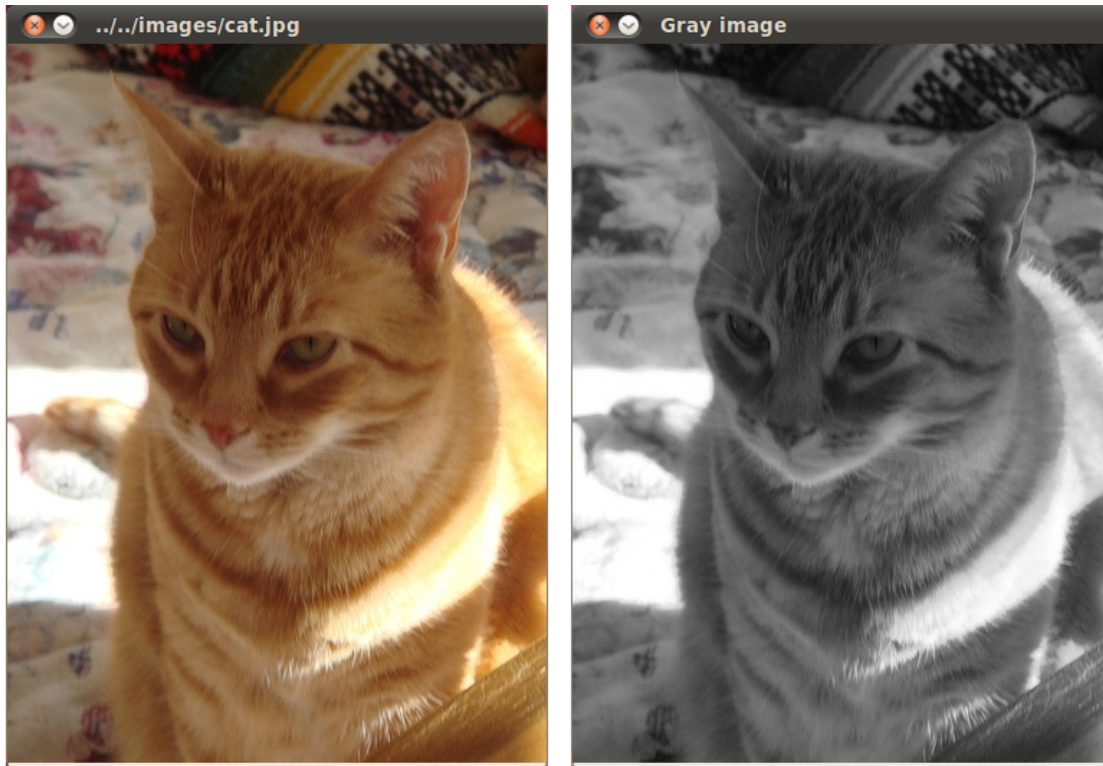
```
namedWindow( imageName, CV_WINDOW_AUTOSIZE );
namedWindow( "Gray image", CV_WINDOW_AUTOSIZE );

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

5. Add the usual `waitKey(0)` for the program to wait forever until the user presses a key.

8.4 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 .png file named *Gray_Image.png*:



Congratulations, you are done with this tutorial!

CHANGING THE CONTRAST AND BRIGHTNESS OF AN IMAGE!

9.1 Goal

In this tutorial you will learn how to:

- Access pixel values
- Initialize a matrix with zeros
- Learn what `saturate_cast` does and why it is useful
- Get some cool info about pixel transformations

9.2 Cool Theory

Note: The explanation below belongs to the book [Computer Vision: Algorithms and Applications](#) by Richard Szeliski

Image Processing

- A general image processing operator is a function that takes one or more input images and produces an output image.
- Image transforms can be seen as:
 - Point operators (pixel transforms)
 - Neighborhood (area-based) operators

Pixel Transforms

- In this kind of image processing transform, each output pixel's value depends on only the corresponding input pixel value (plus, potentially, some globally collected information or parameters).
- Examples of such operators include *brightness and contrast adjustments* as well as color correction and transformations.

Brightness and contrast adjustments

- Two commonly used point processes are *multiplication* and *addition* with a constant:

$$g(x) = \alpha f(x) + \beta$$

- The parameters $\alpha > 0$ and β are often called the *gain* and *bias* parameters; sometimes these parameters are said to control *contrast* and *brightness* respectively.
- You can think of $f(x)$ as the source image pixels and $g(x)$ as the output image pixels. Then, more conveniently we can write the expression as:

$$g(i, j) = \alpha \cdot f(i, j) + \beta$$

where i and j indicates that the pixel is located in the i -th row and j -th column.

9.3 Code

- The following code performs the operation $g(i, j) = \alpha \cdot f(i, j) + \beta$
- Here it is:

```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;

double alpha; /**< Simple contrast control */
int beta; /**< Simple brightness control */

int main( int argc, char** argv )
{
    /// Read image given by user
    Mat image = imread( argv[1] );
    Mat new_image = Mat::zeros( image.size(), image.type() );

    /// Initialize values
    std::cout<<" Basic Linear Transforms "<<std::endl;
    std::cout<<"-----"<<std::endl;
    std::cout<<"* Enter the alpha value [1.0-3.0]: ";std::cin>>alpha;
    std::cout<<"* Enter the beta value [0-100]: "; std::cin>>beta;

    /// Do the operation new_image(i,j) = alpha*image(i,j) + beta
    for( int y = 0; y < image.rows; y++ )
    {
        for( int x = 0; x < image.cols; x++ )
        {
            for( int c = 0; c < 3; c++ )
            {
                new_image.at<Vec3b>(y,x)[c] = saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] )
            }
        }
    }

    /// Create Windows
    namedWindow("Original Image", 1);
    namedWindow("New Image", 1);
```

```

/// Show stuff
imshow("Original Image", image);
imshow("New Image", new_image);

/// Wait until user press some key
waitKey();
return 0;
}

```

9.4 Explanation

1. We begin by creating parameters to save α and β to be entered by the user:

```

double alpha;
int beta;

```

2. We load an image using `imread` and save it in a Mat object:

```
Mat image = imread( argv[1] );
```

3. Now, since we will make some transformations to this image, we need a new Mat object to store it. Also, we want this to have the following features:

- Initial pixel values equal to zero
- Same size and type as the original image

```
Mat new_image = Mat::zeros( image.size(), image.type() );
```

We observe that `Mat::zeros` returns a Matlab-style zero initializer based on `image.size()` and `image.type()`

4. Now, to perform the operation $g(i,j) = \alpha \cdot f(i,j) + \beta$ we will access to each pixel in image. Since we are operating with RGB images, we will have three values per pixel (R, G and B), so we will also access them separately. Here is the piece of code:

```

for( int y = 0; y < image.rows; y++ )
{
    for( int x = 0; x < image.cols; x++ )
    {
        for( int c = 0; c < 3; c++ )
        {
            new_image.at<Vec3b>(y,x)[c] = saturate_cast<uchar>( alpha*( image.at<Vec3b>(y,x)[c] + beta ) );
        }
    }
}

```

Notice the following:

- To access each pixel in the images we are using this syntax: `image.at<Vec3b>(y,x)[c]` where `y` is the row, `x` is the column and `c` is R, G or B (0, 1 or 2).
 - Since the operation $\alpha \cdot p(i,j) + \beta$ can give values out of range or not integers (if α is float), we use `saturate_cast` to make sure the values are valid.
5. Finally, we create windows and show the images, the usual way.

```

namedWindow("Original Image", 1);
namedWindow("New Image", 1);

imshow("Original Image", image);
imshow("New Image", new_image);

waitKey(0);

```

Note: Instead of using the **for** loops to access each pixel, we could have simply used this command:

```
image.convertTo(new_image, -1, alpha, beta);
```

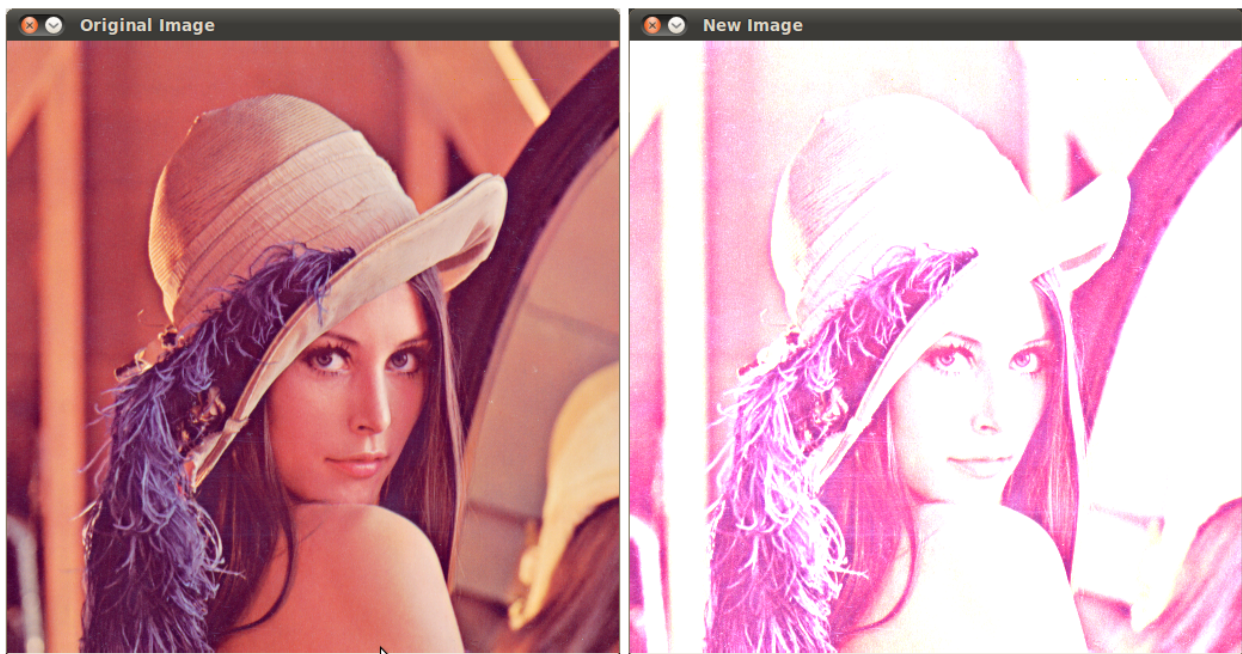
where `convertTo` would effectively perform $new_image = a * image + beta$. However, we wanted to show you how to access each pixel. In any case, both methods give the same result.

9.5 Result

- Running our code and using $\alpha = 2.2$ and $\beta = 50$

```
$ ./BasicLinearTransforms lena.png
Basic Linear Transforms
-----
* Enter the alpha value [1.0-3.0]: 2.2
* Enter the beta value [0-100]: 50
```

- We get this:



ADDING (BLENDING) TWO IMAGES USING OPENCV

10.1 Goal

In this tutorial you will learn how to:

- What is *linear blending* and why it is useful.
- Add two images using `addWeighted`

10.2 Cool Theory

Note: The explanation below belongs to the book [Computer Vision: Algorithms and Applications](#) by Richard Szeliski

From our previous tutorial, we know already a bit of *Pixel operators*. An interesting dyadic (two-input) operator is the *linear blend operator*:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

By varying α from $0 \rightarrow 1$ this operator can be used to perform a temporal *cross-dissolve* between two images or videos, as seen in slide shows and film production (cool, eh?)

10.3 Code

As usual, after the not-so-lengthy explanation, let's go to the code. Here it is:

```
#include <cv.h>
#include <highgui.h>
#include <iostream>

using namespace cv;

int main( int argc, char** argv )
{
    double alpha = 0.5; double beta; double input;

    Mat src1, src2, dst;
```

```
/// Ask the user enter alpha
std::cout<<" Simple Linear Blender "<<std::endl;
std::cout<<"-----"<<std::endl;
std::cout<<"* Enter alpha [0-1]: ";
std::cin>>input;

/// We use the alpha provided by the user iff it is between 0 and 1
if( alpha >= 0 && alpha <= 1 )
    { alpha = input; }

/// Read image ( same size, same type )
src1 = imread("../images/LinuxLogo.jpg");
src2 = imread("../images/WindowsLogo.jpg");

if( !src1.data ) { printf("Error loading src1 \n"); return -1; }
if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

/// Create Windows
namedWindow("Linear Blend", 1);

beta = ( 1.0 - alpha );
addWeighted( src1, alpha, src2, beta, 0.0, dst);

imshow( "Linear Blend", dst );

waitKey(0);
return 0;
}
```

10.4 Explanation

1. Since we are going to perform:

$$g(x) = (1 - \alpha)f_0(x) + \alpha f_1(x)$$

We need two source images ($f_0(x)$ and $f_1(x)$). So, we load them in the usual way:

```
src1 = imread("../images/LinuxLogo.jpg");
src2 = imread("../images/WindowsLogo.jpg");
```

Warning: Since we are *adding* *src1* and *src2*, they both have to be of the same size (width and height) and type.

2. Now we need to generate the $g(x)$ image. For this, the function `addWeighted` comes quite handy:

```
beta = ( 1.0 - alpha );
addWeighted( src1, alpha, src2, beta, 0.0, dst);
```

since `addWeighted` produces:

$$dst = \alpha \cdot src1 + \beta \cdot src2 + \gamma$$

In this case, γ is the argument 0.0 in the code above.

3. Create windows, show the images and wait for the user to end the program.

10.5 Result



ADDING A TRACKBAR TO OUR APPLICATIONS!

- In the previous tutorials (about *linear blending* and the *brightness and contrast adjustments*) you might have noted that we needed to give some **input** to our programs, such as α and β . We accomplished that by entering this data using the Terminal
- Well, it is time to use some fancy GUI tools. OpenCV provides some GUI utilities (*highgui.h*) for you. An example of this is a **Trackbar**



- In this tutorial we will just modify our two previous programs so that they get the input information from the trackbar.

11.1 Goals

In this tutorial you will learn how to:

- Add a Trackbar in an OpenCV window by using `createTrackbar`

11.2 Code

Let's modify the program made in the tutorial *Adding (blending) two images using OpenCV*. We will let the user enter the α value by using the Trackbar.

```
#include <cv.h>
#include <highgui.h>

using namespace cv;

/// Global Variables
const int alpha_slider_max = 100;
int alpha_slider;
double alpha;
double beta;
```

```
/// Matrices to store images
Mat src1;
Mat src2;
Mat dst;

/**
 * @function on_trackbar
 * @brief Callback for trackbar
 */
void on_trackbar( int, void* )
{
    alpha = (double) alpha_slider/alpha_slider_max ;
    beta = ( 1.0 - alpha );

    addWeighted( src1, alpha, src2, beta, 0.0, dst);

    imshow( "Linear Blend", dst );
}

int main( int argc, char** argv )
{
    /// Read image ( same size, same type )
    src1 = imread("../images/LinuxLogo.jpg");
    src2 = imread("../images/WindowsLogo.jpg");

    if( !src1.data ) { printf("Error loading src1 \n"); return -1; }
    if( !src2.data ) { printf("Error loading src2 \n"); return -1; }

    /// Initialize values
    alpha_slider = 0;

    /// Create Windows
    namedWindow("Linear Blend", 1);

    /// Create Trackbars
    char TrackbarName[50];
    sprintf( TrackbarName, "Alpha x %d", alpha_slider_max );

    createTrackbar( TrackbarName, "Linear Blend", &alpha_slider, alpha_slider_max, on_trackbar );

    /// Show some stuff
    on_trackbar( alpha_slider, 0 );

    /// Wait until user press some key
    waitKey(0);
    return 0;
}
```

11.3 Explanation

We only analyze the code that is related to Trackbar:

1. First, we load 02 images, which are going to be blended.

```
src1 = imread("../images/LinuxLogo.jpg");
src2 = imread("../images/WindowsLogo.jpg");
```

2. To create a trackbar, first we have to create the window in which it is going to be located. So:

```
namedWindow("Linear Blend", 1);
```

3. Now we can create the Trackbar:

```
createTrackbar( TrackbarName, "Linear Blend", &alpha_slider, alpha_slider_max, on_trackbar );
```

Note the following:

- Our Trackbar has a label **TrackbarName**
- The Trackbar is located in the window named **“Linear Blend”**
- The Trackbar values will be in the range from 0 to **alpha_slider_max** (the minimum limit is always **zero**).
- The numerical value of Trackbar is stored in **alpha_slider**
- Whenever the user moves the Trackbar, the callback function **on_trackbar** is called

4. Finally, we have to define the callback function **on_trackbar**

```
void on_trackbar( int, void* )
{
    alpha = (double) alpha_slider/alpha_slider_max ;
    beta = ( 1.0 - alpha );

    addWeighted( src1, alpha, src2, beta, 0.0, dst);

    imshow( "Linear Blend", dst );
}
```

Note that:

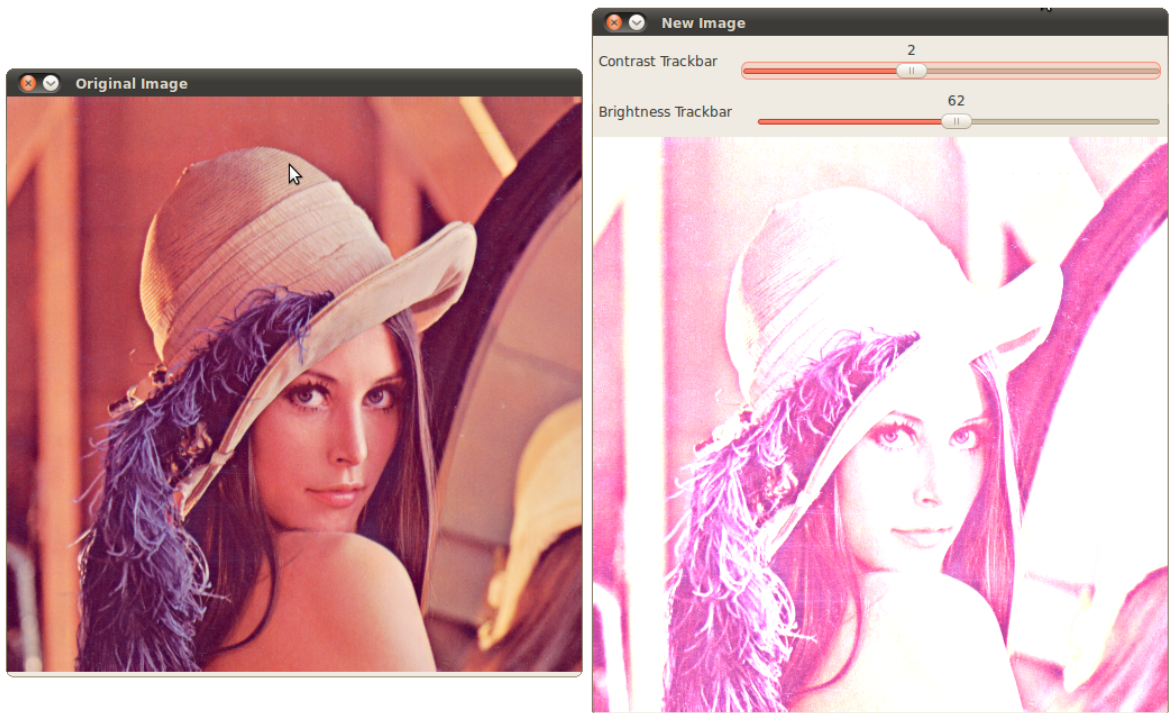
- We use the value of **alpha_slider** (integer) to get a double value for **alpha**.
- **alpha_slider** is updated each time the trackbar is displaced by the user.
- We define *src1*, *src2*, *dst*, *alpha*, *alpha_slider* and *beta* as global variables, so they can be used everywhere.

11.4 Result

- Our program produces the following output:



- As a manner of practice, you can also add 02 trackbars for the program made in *Changing the contrast and brightness of an image!*. One trackbar to set α and another for β . The output might look like:



BASIC DRAWING

12.1 Goals

In this tutorial you will learn how to:

- Use `Point` to define 2D points in an image.
- Use `Scalar` and why it is useful
- Draw a **line** by using the OpenCV function `line`
- Draw an **ellipse** by using the OpenCV function `ellipse`
- Draw a **rectangle** by using the OpenCV function `rectangle`
- Draw a **circle** by using the OpenCV function `circle`
- Draw a **filled polygon** by using the OpenCV function `fillPoly`

12.2 OpenCV Theory

For this tutorial, we will heavily use two structures: `Point` and `Scalar`:

Point

It represents a 2D point, specified by its image coordinates x and y . We can define it as:

```
Point pt;  
pt.x = 10;  
pt.y = 8;
```

or

```
Point pt = Point(10, 8);
```

Scalar

- Represents a 4-element vector. The type `Scalar` is widely used in OpenCV for passing pixel values.
- In this tutorial, we will use it extensively to represent RGB color values (3 parameters). It is not necessary to define the last argument if it is not going to be used.

- Let's see an example, if we are asked for a color argument and we give:

```
Scalar( a, b, c )
```

We would be defining a RGB color such as: *Red* = *c*, *Green* = *b* and *Blue* = *a*

12.3 Code

- This code is in your OpenCV sample folder. Otherwise you can grab it from [here](#)

12.4 Explanation

1. Since we plan to draw two examples (an atom and a rook), we have to create 02 images and two windows to display them.

```
/// Windows names
char atom_window[] = "Drawing 1: Atom";
char rook_window[] = "Drawing 2: Rook";

/// Create black empty images
Mat atom_image = Mat::zeros( w, w, CV_8UC3 );
Mat rook_image = Mat::zeros( w, w, CV_8UC3 );
```

2. We created functions to draw different geometric shapes. For instance, to draw the atom we used *MyEllipse* and *MyFilledCircle*:

```
/// 1. Draw a simple atom:

/// 1.a. Creating ellipses
MyEllipse( atom_image, 90 );
MyEllipse( atom_image, 0 );
MyEllipse( atom_image, 45 );
MyEllipse( atom_image, -45 );

/// 1.b. Creating circles
MyFilledCircle( atom_image, Point( w/2.0, w/2.0 ) );
```

3. And to draw the rook we employed *MyLine*, *rectangle* and a *MyPolygon*:

```
/// 2. Draw a rook

/// 2.a. Create a convex polygon
MyPolygon( rook_image );

/// 2.b. Creating rectangles
rectangle( rook_image,
           Point( 0, 7*w/8.0 ),
           Point( w, w ),
           Scalar( 0, 255, 255 ),
           -1,
           8 );

/// 2.c. Create a few lines
MyLine( rook_image, Point( 0, 15*w/16 ), Point( w, 15*w/16 ) );
MyLine( rook_image, Point( w/4, 7*w/8 ), Point( w/4, w ) );
```

```
MyLine( rook_image, Point( w/2, 7*w/8 ), Point( w/2, w ) );
MyLine( rook_image, Point( 3*w/4, 7*w/8 ), Point( 3*w/4, w ) );
```

4. Let's check what is inside each of these functions:

- *MyLine*

```
void MyLine( Mat img, Point start, Point end )
{
    int thickness = 2;
    int lineType = 8;
    line( img,
          start,
          end,
          Scalar( 0, 0, 0 ),
          thickness,
          lineType );
}
```

As we can see, *MyLine* just call the function `line`, which does the following:

- Draw a line from Point **start** to Point **end**
- The line is displayed in the image **img**
- The line color is defined by **Scalar(0, 0, 0)** which is the RGB value correspondent to **Black**
- The line thickness is set to **thickness** (in this case 2)
- The line is a 8-connected one (**lineType = 8**)

- *MyEllipse*

```
void MyEllipse( Mat img, double angle )
{
    int thickness = 2;
    int lineType = 8;

    ellipse( img,
             Point( w/2.0, w/2.0 ),
             Size( w/4.0, w/16.0 ),
             angle,
             0,
             360,
             Scalar( 255, 0, 0 ),
             thickness,
             lineType );
}
```

From the code above, we can observe that the function `ellipse` draws an ellipse such that:

- The ellipse is displayed in the image **img**
- The ellipse center is located in the point **(w/2.0, w/2.0)** and is enclosed in a box of size **(w/4.0, w/16.0)**
- The ellipse is rotated **angle** degrees
- The ellipse extends an arc between **0** and **360** degrees
- The color of the figure will be **Scalar(255, 255, 0)** which means blue in RGB value.
- The ellipse's **thickness** is 2.

- *MyFilledCircle*

```
void MyFilledCircle( Mat img, Point center )
{
    int thickness = -1;
    int lineType = 8;

    circle( img,
            center,
            w/32.0,
            Scalar( 0, 0, 255 ),
            thickness,
            lineType );
}
```

Similar to the ellipse function, we can observe that *circle* receives as arguments:

- The image where the circle will be displayed (**img**)
- The center of the circle denoted as the Point **center**
- The radius of the circle: **w/32.0**
- The color of the circle: **Scalar(0, 0, 255)** which means *Red* in RGB
- Since **thickness** = -1, the circle will be drawn filled.

- *MyPolygon*

```
void MyPolygon( Mat img )
{
    int lineType = 8;

    /** Create some points */
    Point rook_points[1][20];
    rook_points[0][0] = Point( w/4.0, 7*w/8.0 );
    rook_points[0][1] = Point( 3*w/4.0, 7*w/8.0 );
    rook_points[0][2] = Point( 3*w/4.0, 13*w/16.0 );
    rook_points[0][3] = Point( 11*w/16.0, 13*w/16.0 );
    rook_points[0][4] = Point( 19*w/32.0, 3*w/8.0 );
    rook_points[0][5] = Point( 3*w/4.0, 3*w/8.0 );
    rook_points[0][6] = Point( 3*w/4.0, w/8.0 );
    rook_points[0][7] = Point( 26*w/40.0, w/8.0 );
    rook_points[0][8] = Point( 26*w/40.0, w/4.0 );
    rook_points[0][9] = Point( 22*w/40.0, w/4.0 );
    rook_points[0][10] = Point( 22*w/40.0, w/8.0 );
    rook_points[0][11] = Point( 18*w/40.0, w/8.0 );
    rook_points[0][12] = Point( 18*w/40.0, w/4.0 );
    rook_points[0][13] = Point( 14*w/40.0, w/4.0 );
    rook_points[0][14] = Point( 14*w/40.0, w/8.0 );
    rook_points[0][15] = Point( w/4.0, w/8.0 );
    rook_points[0][16] = Point( w/4.0, 3*w/8.0 );
    rook_points[0][17] = Point( 13*w/32.0, 3*w/8.0 );
    rook_points[0][18] = Point( 5*w/16.0, 13*w/16.0 );
    rook_points[0][19] = Point( w/4.0, 13*w/16.0 );

    const Point* ppt[1] = { rook_points[0] };
    int npt[] = { 20 };

    fillPoly( img,
              ppt,
              npt,
              1,
```

```

        Scalar( 255, 255, 255 ),
        lineType );
}

```

To draw a filled polygon we use the function `fillPoly`. We note that:

- The polygon will be drawn on **img**
- The vertices of the polygon are the set of points in **ppt**
- The total number of vertices to be drawn are **npt**
- The number of polygons to be drawn is only **1**
- The color of the polygon is defined by **Scalar(255, 255, 255)**, which is the RGB value for *white*

- *rectangle*

```

rectangle( rook_image,
           Point( 0, 7*w/8.0 ),
           Point( w, w ),
           Scalar( 0, 255, 255 ),
           -1,
           8 );

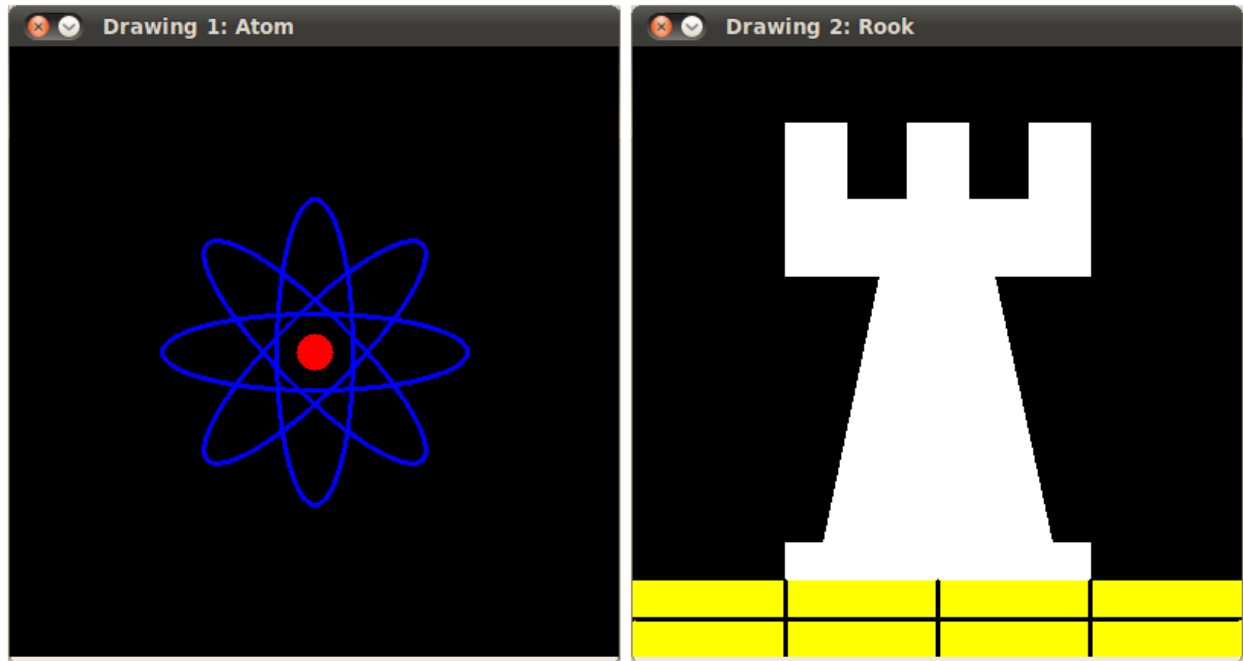
```

Finally we have the `rectangle` function (we did not create a special function for this guy). We note that:

- The rectangle will be drawn on **rook_image**
- Two opposite vertices of the rectangle are defined by **** Point(0, 7*w/8.0)**** and **Point(w, w)**
- The color of the rectangle is given by **Scalar(0, 255, 255)** which is the RGB value for *yellow*
- Since the thickness value is given by **-1**, the rectangle will be filled.

12.5 Result

Compiling and running your program should give you a result like this:



FANCY DRAWING!

13.1 Goals

In this tutorial you will learn how to:

- Use the *Random Number generator class (RNG)* and how to get a random number from a uniform distribution.
- Display Text on an OpenCV window by using the function `putText`

13.2 Code

- In the previous tutorial we drew diverse geometric figures, giving as input parameters such as coordinates (in the form of `Points`), color, thickness, etc. You might have noticed that we gave specific values for these arguments.
- In this tutorial, we intend to use *random* values for the drawing parameters. Also, we intend to populate our image with a big number of geometric figures. Since we will be initializing them in a random fashion, this process will be automatic and made by using *loops*
- This code is in your OpenCV sample folder. Otherwise you can grab it from [here](#)

13.3 Explanation

1. Let's start by checking out the *main* function. We observe that first thing we do is creating a *Random Number Generator* object (RNG):

```
RNG rng( 0xFFFFFFFF );
```

RNG implements a random number generator. In this example, *rng* is a RNG element initialized with the value `0xFFFFFFFF`

2. Then we create a matrix initialized to *zeros* (which means that it will appear as black), specifying its height, width and its type:

```
/// Initialize a matrix filled with zeros
Mat image = Mat::zeros( window_height, window_width, CV_8UC3 );

/// Show it in a window during DELAY ms
imshow( window_name, image );
```

3. Then we proceed to draw crazy stuff. After taking a look at the code, you can see that it is mainly divided in 8 sections, defined as functions:

```
/// Now, let's draw some lines
c = Drawing_Random_Lines(image, window_name, rng);
if( c != 0 ) return 0;

/// Go on drawing, this time nice rectangles
c = Drawing_Random_Rectangles(image, window_name, rng);
if( c != 0 ) return 0;

/// Draw some ellipses
c = Drawing_Random_Ellipses( image, window_name, rng );
if( c != 0 ) return 0;

/// Now some polylines
c = Drawing_Random_Polylines( image, window_name, rng );
if( c != 0 ) return 0;

/// Draw filled polygons
c = Drawing_Random_Filled_Polygons( image, window_name, rng );
if( c != 0 ) return 0;

/// Draw circles
c = Drawing_Random_Circles( image, window_name, rng );
if( c != 0 ) return 0;

/// Display text in random positions
c = Displaying_Random_Text( image, window_name, rng );
if( c != 0 ) return 0;

/// Displaying the big end!
c = Displaying_Big_End( image, window_name, rng );
```

All of these functions follow the same pattern, so we will analyze only a couple of them, since the same explanation applies for all.

4. Checking out the function **Drawing_Random_Lines**:

```
/**
 * @function Drawing_Random_Lines
 */
int Drawing_Random_Lines( Mat image, char* window_name, RNG rng )
{
    int lineType = 8;
    Point pt1, pt2;

    for( int i = 0; i < NUMBER; i++ )
    {
        pt1.x = rng.uniform( x_1, x_2 );
        pt1.y = rng.uniform( y_1, y_2 );
        pt2.x = rng.uniform( x_1, x_2 );
        pt2.y = rng.uniform( y_1, y_2 );

        line( image, pt1, pt2, randomColor(rng), rng.uniform(1, 10), 8 );
        imshow( window_name, image );
        if( waitKey( DELAY ) >= 0 )
        {
            return -1;
        }
    }

    return 0;
}
```

We can observe the following:

- The *for* loop will repeat **NUMBER** times. Since the function `line` is inside this loop, that means that **NUMBER** lines will be generated.
- The line extremes are given by *pt1* and *pt2*. For *pt1* we can see that:

```
pt1.x = rng.uniform( x_1, x_2 );
pt1.y = rng.uniform( y_1, y_2 );
```

- We know that **rng** is a *Random number generator* object. In the code above we are calling **rng.uniform(a,b)**. This generates a randomly uniformed distribution between the values **a** and **b** (inclusive in **a**, exclusive in **b**).
- From the explanation above, we deduce that the extremes *pt1* and *pt2* will be random values, so the lines positions will be quite unpredictable, giving a nice visual effect (check out the Result section below).
- As another observation, we notice that in the `line` arguments, for the *color* input we enter:

```
randomColor(rng)
```

Let's check the function implementation:

```
static Scalar randomColor( RNG& rng )
{
    int icolor = (unsigned) rng;
    return Scalar( icolor&255, (icolor>>8)&255, (icolor>>16)&255 );
}
```

As we can see, the return value is an *Scalar* with 3 randomly initialized values, which are used as the *R*, *G* and *B* parameters for the line color. Hence, the color of the lines will be random too!

5. The explanation above applies for the other functions generating circles, ellipses, polygons, etc. The parameters such as *center* and *vertices* are also generated randomly.
6. Before finishing, we also should take a look at the functions *Display_Random_Text* and *Displaying_Big_End*, since they both have a few interesting features:

7. Display_Random_Text:

```
int Displaying_Random_Text( Mat image, char* window_name, RNG rng )
{
    int lineType = 8;

    for ( int i = 1; i < NUMBER; i++ )
    {
        Point org;
        org.x = rng.uniform(x_1, x_2);
        org.y = rng.uniform(y_1, y_2);

        putText( image, "Testing text rendering", org, rng.uniform(0,8),
                rng.uniform(0,100)*0.05+0.1, randomColor(rng), rng.uniform(1, 10), lineType);

        imshow( window_name, image );
        if( waitKey(DELAY) >= 0 )
        { return -1; }
    }

    return 0;
}
```

Everything looks familiar but the expression:

```
putText( image, "Testing text rendering", org, rng.uniform(0,8),
         rng.uniform(0,100)*0.05+0.1, randomColor(rng), rng.uniform(1, 10), lineType);
```

So, what does the function `putText` do? In our example:

- Draws the text “Testing text rendering” in **image**
- The bottom-left corner of the text will be located in the Point **org**
- The font type is a random integer value in the range: $[0, 8 >$.
- The scale of the font is denoted by the expression **rng.uniform(0, 100)x0.05 + 0.1** (meaning its range is: $[0.1, 5.1 >$)
- The text color is random (denoted by **randomColor(rng)**)
- The text thickness ranges between 1 and 10, as specified by **rng.uniform(1,10)**

As a result, we will get (analogously to the other drawing functions) **NUMBER** texts over our image, in random locations.

8. Displaying_Big_End

```
int Displaying_Big_End( Mat image, char* window_name, RNG rng )
{
    Size textsize = getTextSize("OpenCV forever!", CV_FONT_HERSHEY_COMPLEX, 3, 5, 0);
    Point org((window_width - textsize.width)/2, (window_height - textsize.height)/2);
    int lineType = 8;

    Mat image2;

    for( int i = 0; i < 255; i += 2 )
    {
        image2 = image - Scalar::all(i);
        putText( image2, "OpenCV forever!", org, CV_FONT_HERSHEY_COMPLEX, 3,
                 Scalar(i, i, 255), 5, lineType );

        imshow( window_name, image2 );
        if( waitKey(DELAY) >= 0 )
        { return -1; }
    }

    return 0;
}
```

Besides the function `getTextSize` (which gets the size of the argument text), the new operation we can observe is inside the *for* loop:

```
image2 = image - Scalar::all(i)
```

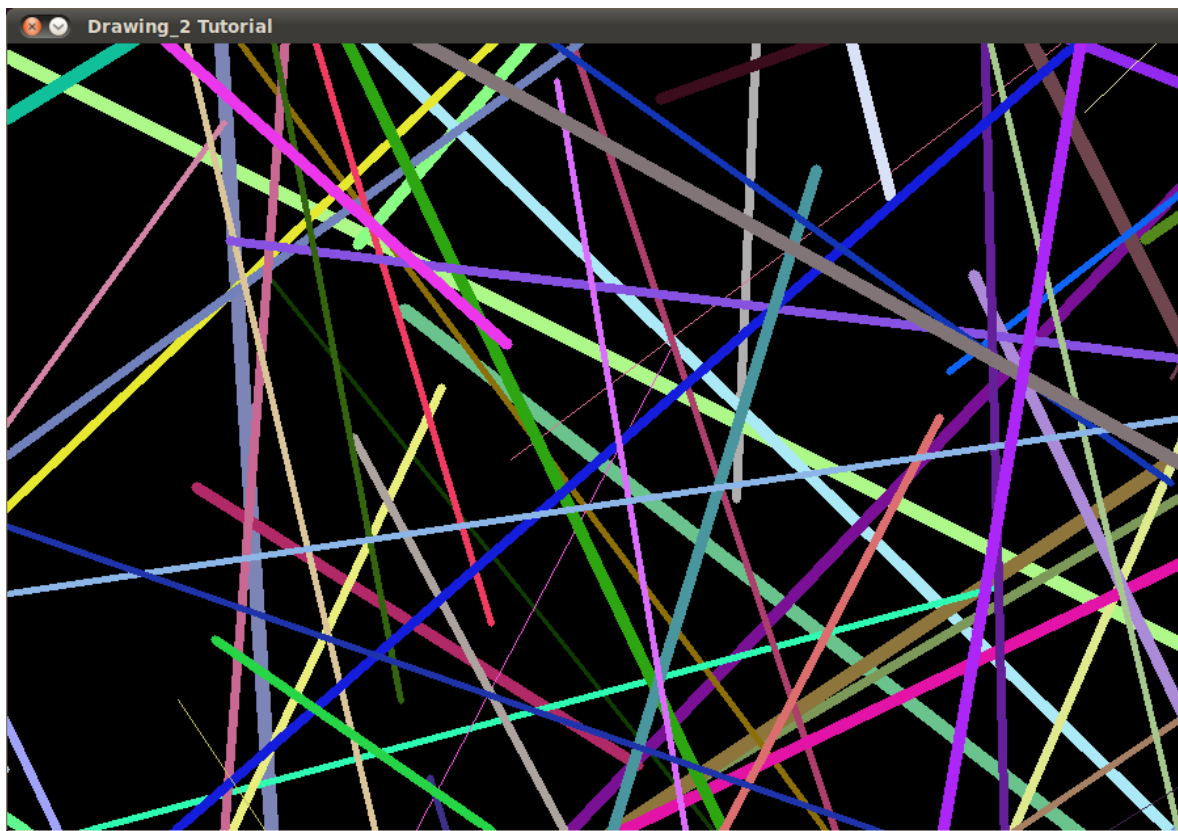
So, **image2** is the subtraction of **image** and **Scalar::all(i)**. In fact, what happens here is that every pixel of **image2** will be the result of subtracting every pixel of **image** minus the value of **i** (remember that for each pixel we are considering three values such as R, G and B, so each of them will be affected)

Also remember that the subtraction operation *always* performs internally a **saturate** operation, which means that the result obtained will always be inside the allowed range (no negative and between 0 and 255 for our example).

13.4 Result

As you just saw in the Code section, the program will sequentially execute diverse drawing functions, which will produce:

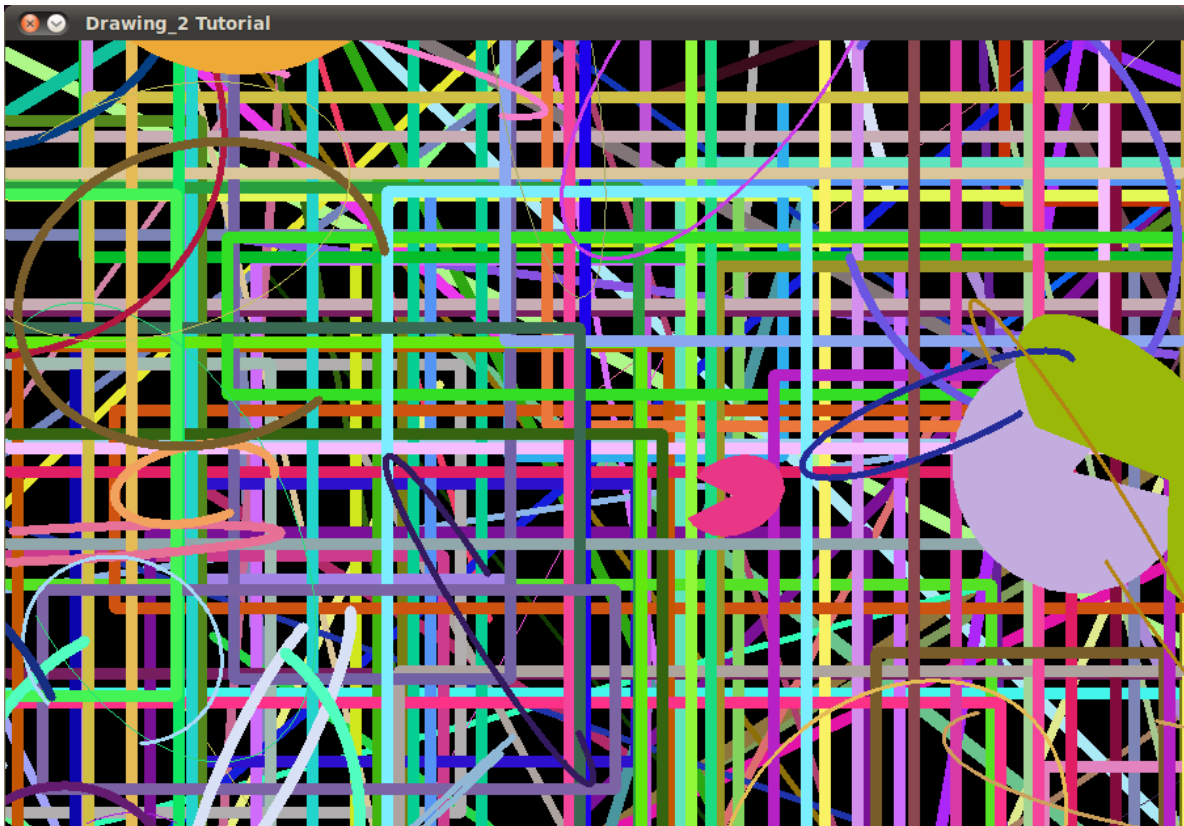
1. First a random set of *NUMBER* lines will appear on screen such as it can be seen in this screenshot:



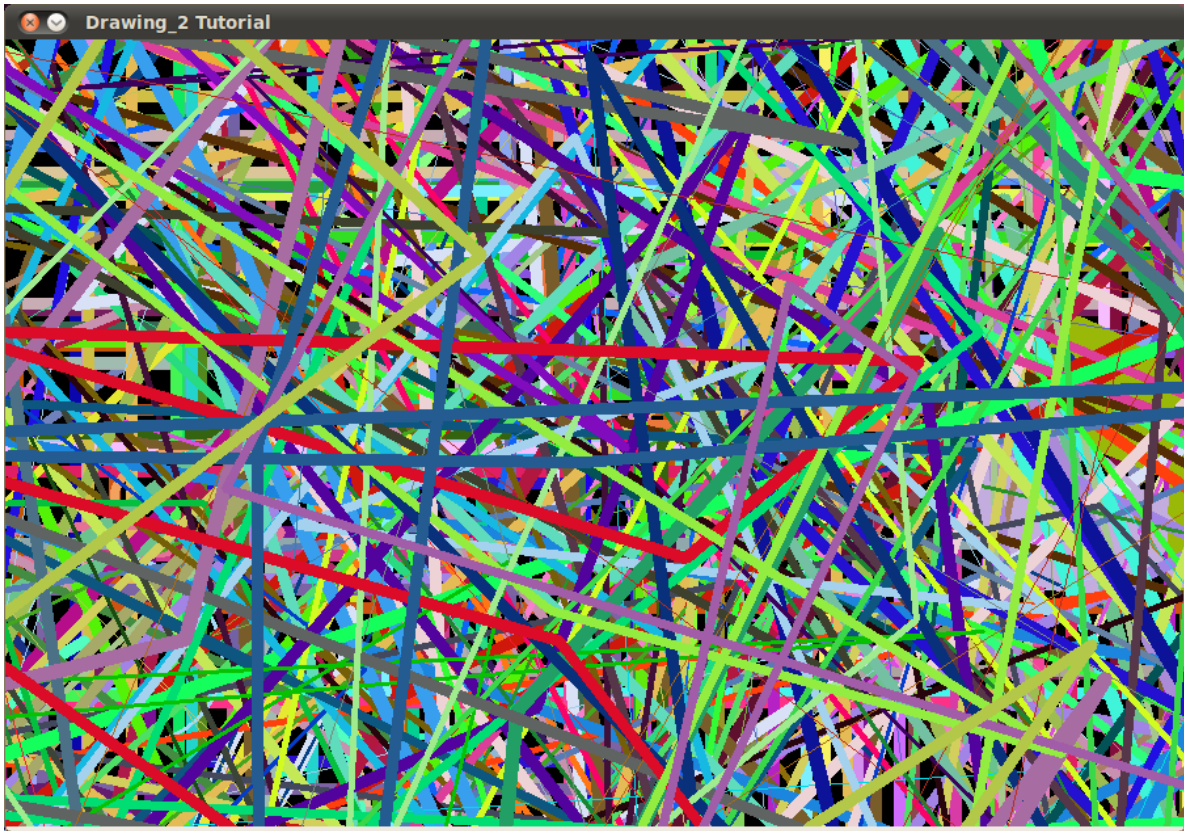
2. Then, a new set of figures, these time *rectangles* will follow:



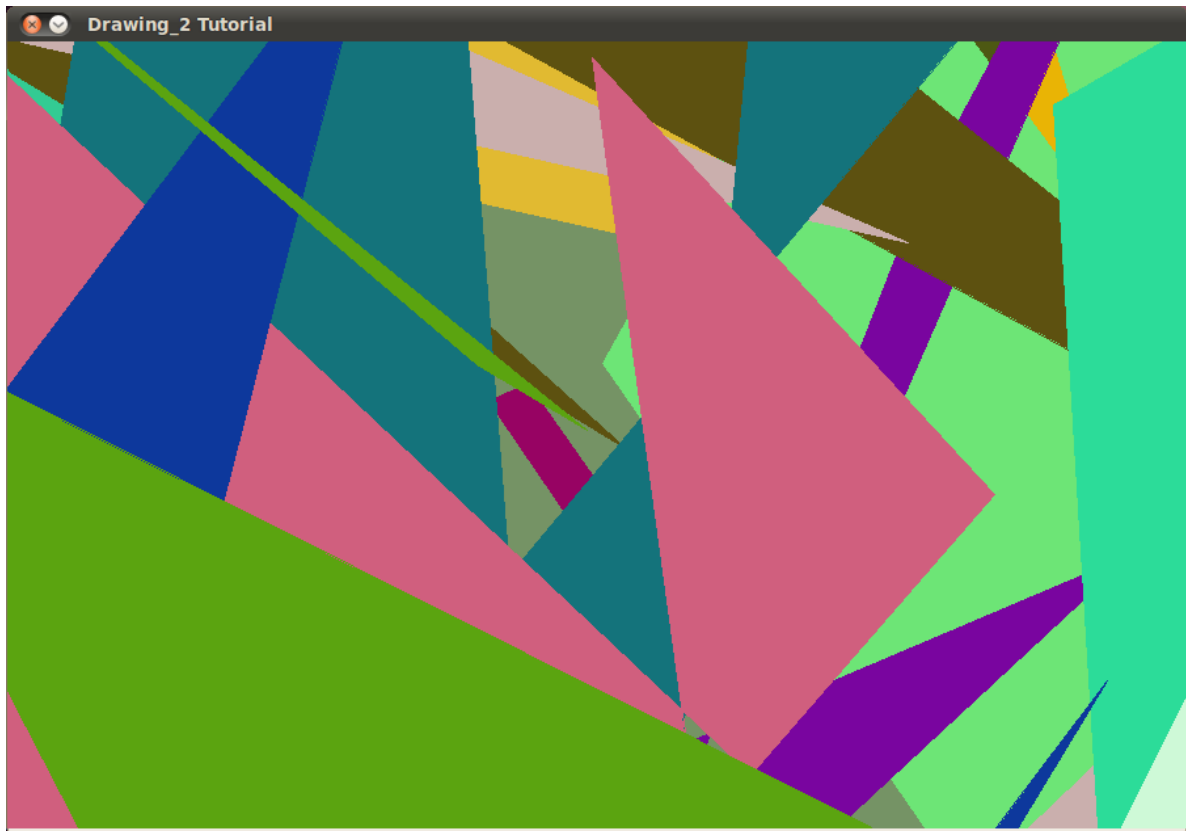
3. Now some ellipses will appear, each of them with random position, size, thickness and arc length:



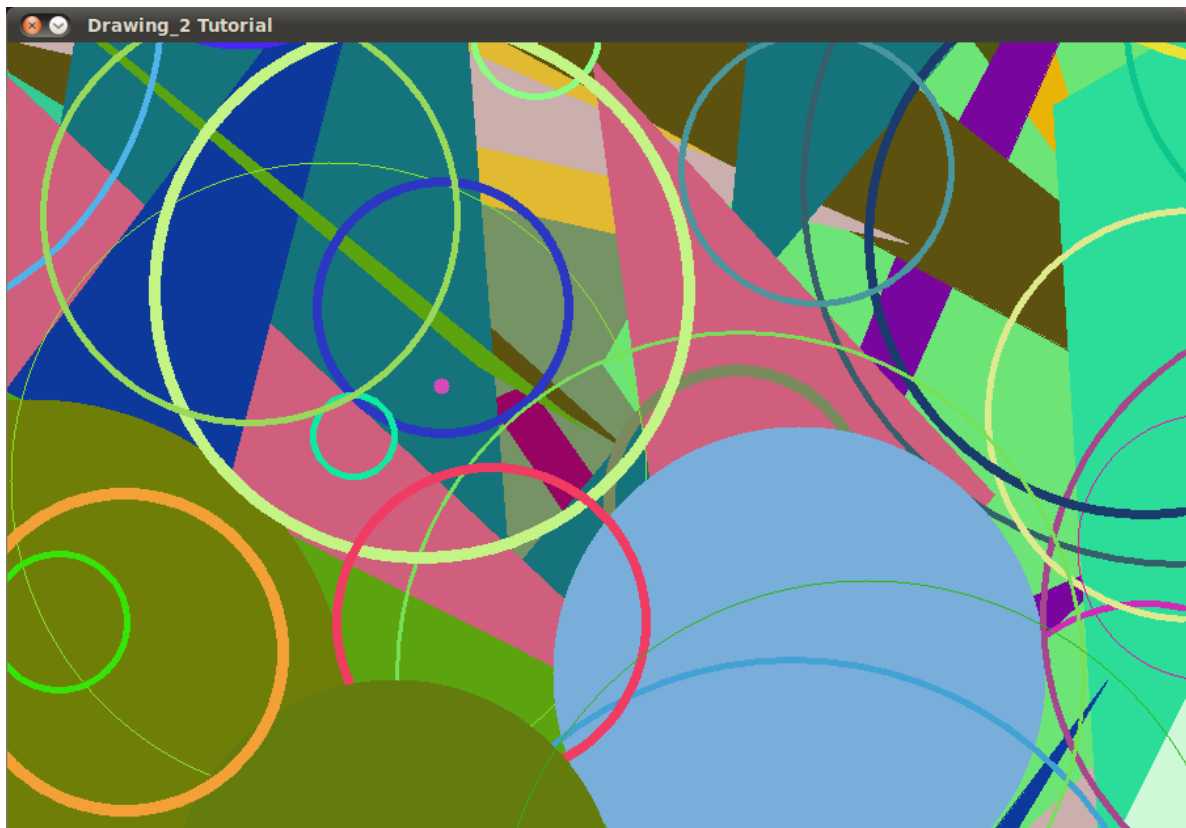
4. Now, *polylines* with 03 segments will appear on screen, again in random configurations.



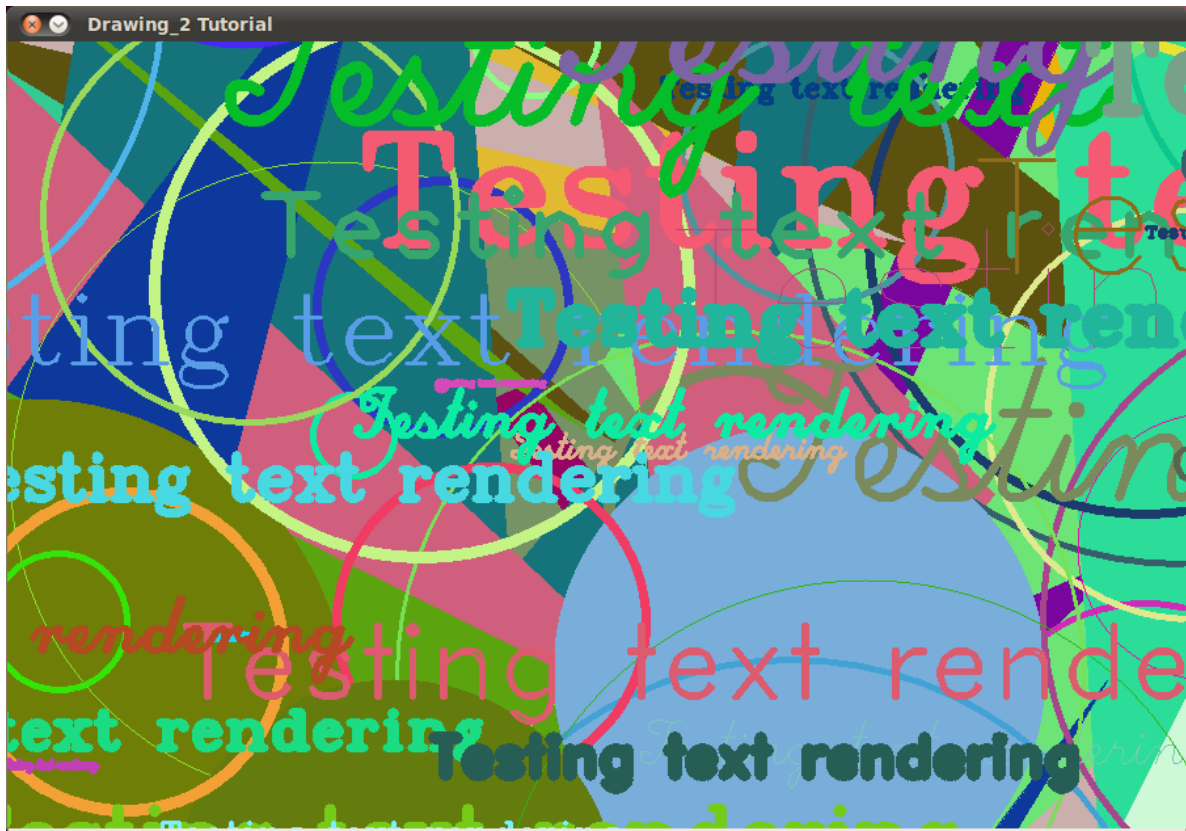
5. Filled polygons (in this example triangles) will follow:



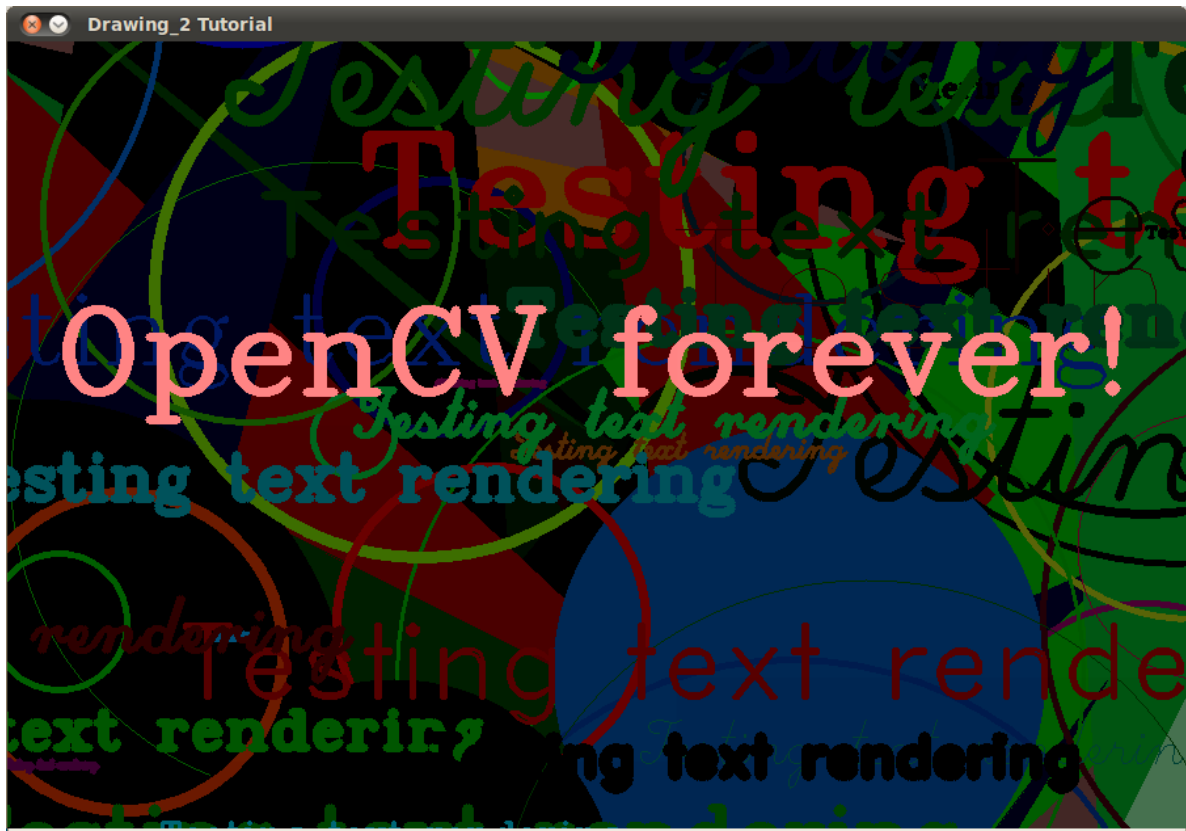
6. The last geometric figure to appear: circles!



7. Near the end, the text “*Testing Text Rendering*” will appear in a variety of fonts, sizes, colors and positions.



8. And the big end (which by the way expresses a big truth too):



SMOOTHING IMAGES

14.1 Goal

In this tutorial you will learn how to:

- Apply diverse linear filters to smooth images using OpenCV functions such as:
 - `blur`
 - `GaussianBlur`
 - `medianBlur`
 - `bilateralFilter`

14.2 Cool Theory

Note: The explanation below belongs to the book [Computer Vision: Algorithms and Applications](#) by Richard Szeliski and to *LearningOpenCV*

- *Smoothing*, also called *blurring*, is a simple and frequently used image processing operation.
- There are many reasons for smoothing. In this tutorial we will focus on smoothing in order to reduce noise (other uses will be seen in the following tutorials).
- To perform a smoothing operation we will apply a *filter* to our image. The most common type of filters are *linear*, in which an output pixel's value (i.e. $g(i, j)$) is determined as a weighted sum of input pixel values (i.e. $f(i + k, j + l)$):

$$g(i, j) = \sum_{k, l} f(i + k, j + l) h(k, l)$$

$h(k, l)$ is called the *kernel*, which is nothing more than the coefficients of the filter.

It helps to visualize a *filter* as a window of coefficients sliding across the image.

- There are many kind of filters, here we will mention the most used:

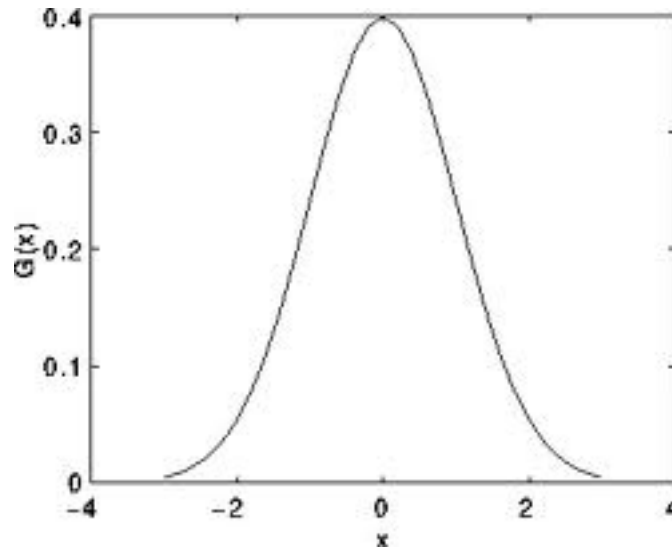
Normalized Box Filter

- This filter is the simplest of all! Each output pixel is the *mean* of its kernel neighbors (all of them contribute with equal weights)
- Just in case, the kernel is below:

$$K = \frac{1}{K_{width} \cdot K_{height}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ \cdot & \cdot & \cdot & \dots & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

Gaussian Filter

- Probably the most useful filter (although not the fastest). Gaussian filtering is done by convolving each point in the input array with a *Gaussian kernel* and then summing them all to produce the output array.
- Just to make the picture clearer, remember how a 1D Gaussian kernel look like?



Assuming that an image is 1D, you can notice that the pixel located in the middle would have the biggest weight. The weight of its neighbors decreases as the spatial distance between them and the center pixel increases.

Note:

- Remember that a 2D Gaussian can be represented as :

$$G_0(x, y) = A e^{\frac{-(x - \mu_x)^2}{2\sigma_x^2} + \frac{-(y - \mu_y)^2}{2\sigma_y^2}}$$

where μ is the mean (the peak) and σ represents the variance (per each of the variables x and y)

Median Filter

The median filter run through each element of the signal (in this case the image) and replace each pixel with the **median** of its neighboring pixels (located in a square neighborhood around the evaluated pixel).

Bilateral Filter

- So far, we have explained some filters which main goal is to *smooth* an input image. However, sometimes the filters do not only dissolve the noise, but also smooth away the *edges*. To avoid this (at certain extent at least), we can use a bilateral filter.
- In an analogous way as the Gaussian filter, the bilateral filter also considers the neighboring pixels with weights assigned to each of them. These weights have two components, the first of which is the same weighting used by the Gaussian filter. The second component takes into account the difference in intensity between the neighboring pixels and the evaluated one.
- For a more detailed explanation you can check [this link](#)

14.3 Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"

using namespace std;
using namespace cv;

/// Global Variables
int DELAY_CAPTION = 1500;
int DELAY_BLUR = 100;
int MAX_KERNEL_LENGTH = 31;

Mat src; Mat dst;
char window_name[] = "Filter Demo 1";

/// Function headers
int display_caption( char* caption );
int display_dst( int delay );

/**
 * function main
 */
int main( int argc, char** argv )
{
    namedWindow( window_name, CV_WINDOW_AUTOSIZE );

    /// Load the source image
    src = imread( "../images/lena.png", 1 );

    if( display_caption( "Original Image" ) != 0 ) { return 0; }

    dst = src.clone();
    if( display_dst( DELAY_CAPTION ) != 0 ) { return 0; }

    /// Applying Homogeneous blur
    if( display_caption( "Homogeneous Blur" ) != 0 ) { return 0; }

    for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
    { blur( src, dst, Size( i, i ), Point(-1,-1) );
      if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

```
/// Applying Gaussian blur
if( display_caption( "Gaussian Blur" ) != 0 ) { return 0; }

for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ GaussianBlur( src, dst, Size( i, i ), 0, 0 );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

/// Applying Median blur
if( display_caption( "Median Blur" ) != 0 ) { return 0; }

for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ medianBlur ( src, dst, i );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

/// Applying Bilateral Filter
if( display_caption( "Bilateral Blur" ) != 0 ) { return 0; }

for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ bilateralFilter ( src, dst, i, i*2, i/2 );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }

/// Wait until user press a key
display_caption( "End: Press a key!" );

waitKey(0);
return 0;
}

int display_caption( char* caption )
{
    dst = Mat::zeros( src.size(), src.type() );
    putText( dst, caption,
        Point( src.cols/4, src.rows/2),
        CV_FONT_HERSHEY_COMPLEX, 1, Scalar(255, 255, 255) );

    imshow( window_name, dst );
    int c = waitKey( DELAY_CAPTION );
    if( c >= 0 ) { return -1; }
    return 0;
}

int display_dst( int delay )
{
    imshow( window_name, dst );
    int c = waitKey ( delay );
    if( c >= 0 ) { return -1; }
    return 0;
}
```

14.4 Explanation

1. Let's check the OpenCV functions that involve only the smoothing procedure, since the rest is already known by now.

2. **Normalized Block Filter:**

OpenCV offers the function `blur` to perform smoothing with this filter.

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ blur( src, dst, Size( i, i ), Point(-1,-1) );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

We specify 4 arguments (more details, check the Reference):

- *src*: Source image
- *dst*: Destination image
- *Size(w,h)*: Defines the size of the kernel to be used (of width *w* pixels and height *h* pixels)
- *Point(-1, -1)*: Indicates where the anchor point (the pixel evaluated) is located with respect to the neighborhood. If there is a negative value, then the center of the kernel is considered the anchor point.

3. Gaussian Filter:

It is performed by the function `GaussianBlur` :

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ GaussianBlur( src, dst, Size( i, i ), 0, 0 );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

Here we use 4 arguments (more details, check the OpenCV reference):

- *src*: Source image
- *dst*: Destination image
- *Size(w, h)*: The size of the kernel to be used (the neighbors to be considered). *w* and *h* have to be odd and positive numbers otherwise this size will be calculated using the σ_x and σ_y arguments.
- σ_x : The standard deviation in x. Writing 0 implies that σ_x is calculated using kernel size.
- σ_y : The standard deviation in y. Writing 0 implies that σ_y is calculated using kernel size.

4. Median Filter:

This filter is provided by the `medianBlur` function:

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ medianBlur ( src, dst, i );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

We use three arguments:

- *src*: Source image
- *dst*: Destination image, must be the same type as *src*
- *i*: Size of the kernel (only one because we use a square window). Must be odd.

5. Bilateral Filter

Provided by OpenCV function `bilateralFilter`

```
for ( int i = 1; i < MAX_KERNEL_LENGTH; i = i + 2 )
{ bilateralFilter ( src, dst, i, i*2, i/2 );
  if( display_dst( DELAY_BLUR ) != 0 ) { return 0; } }
```

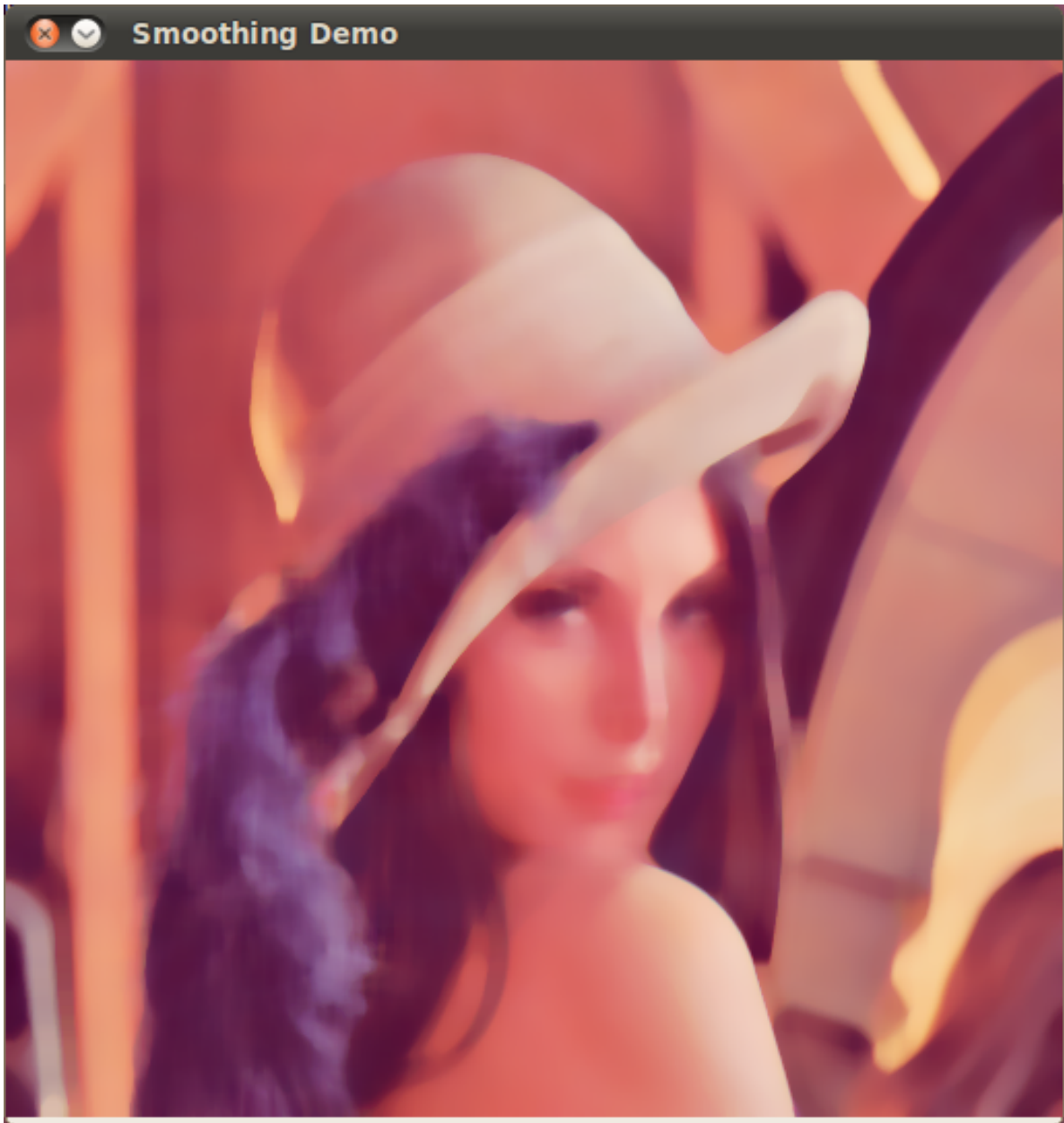
We use 5 arguments:

- *src*: Source image
- *dst*: Destination image
- *d*: The diameter of each pixel neighborhood.

- σ_{Color} : Standard deviation in the color space.
- σ_{Space} : Standard deviation in the coordinate space (in pixel terms)

14.5 Results

- The code opens an image (in this case *lena.png*) and display it under the effects of the 4 filters explained.
- Here is a snapshot of the image smoothed using *medianBlur*:



ERODING AND DILATING

15.1 Goal

In this tutorial you will learn how to:

- Apply two very common morphology operators: Dilation and Erosion. For this purpose, you will use the following OpenCV functions:
 - `erode`
 - `dilate`

15.2 Cool Theory

Note: The explanation below belongs to the book **Learning OpenCV** by Bradski and Kaehler.

Morphological Operations

- In short: A set of operations that process images based on shapes. Morphological operations apply a *structuring element* to an input image and generate an output image.
- The most basic morphological operations are two: Erosion and Dilation. They have a wide array of uses, i.e. :
 - Removing noise
 - Isolation of individual elements and joining disparate elements in an image.
 - Finding of intensity bumps or holes in an image
- We will explain dilation and erosion briefly, using the following image as an example:



Dilation

- This operations consists of convoluting an image A with some kernel (B), which can have any shape or size, usually a square or circle.
- The kernel B has a defined *anchor point*, usually being the center of the kernel.
- As the kernel B is scanned over the image, we compute the maximal pixel value overlapped by B and replace the image pixel in the anchor point position with that maximal value. As you can deduce, this maximizing operation causes bright regions within an image to “grow” (therefore the name *dilation*). Take as an example the image above. Applying dilation we can get:



The background (bright) dilates around the black regions of the letter.

Erosion

- This operation is the sister of dilation. What this does is to compute a local minimum over the area of the kernel.
- As the kernel B is scanned over the image, we compute the minimal pixel value overlapped by B and replace the image pixel under the anchor point with that minimal value.
- Analogously to the example for dilation, we can apply the erosion operator to the original image (shown above). You can see in the result below that the bright areas of the image (the background, apparently), get thinner, whereas the dark zones (the “writing”) gets bigger.



15.3 Code

This tutorial code's is shown lines below. You can also download it from [here](#)

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "highgui.h"
#include <stdlib.h>
#include <stdio.h>
```

```
using namespace cv;
```

```
/// Global variables
```

```
Mat src, erosion_dst, dilation_dst;
```

```
int erosion_elem = 0;
int erosion_size = 0;
int dilation_elem = 0;
int dilation_size = 0;
```

```

int const max_elem = 2;
int const max_kernel_size = 21;

/** Function Headers */
void Erosion( int, void* );
void Dilation( int, void* );

/** @function main */
int main( int argc, char** argv )
{
    /// Load an image
    src = imread( argv[1] );

    if( !src.data )
    { return -1; }

    /// Create windows
    namedWindow( "Erosion Demo", CV_WINDOW_AUTOSIZE );
    namedWindow( "Dilation Demo", CV_WINDOW_AUTOSIZE );
    cvMoveWindow( "Dilation Demo", src.cols, 0 );

    /// Create Erosion Trackbar
    createTrackbar( "Element:\n 0: Rect \n 1: Cross \n 2: Ellipse", "Erosion Demo",
                  &erosion_elem, max_elem,
                  Erosion );

    createTrackbar( "Kernel size:\n 2n +1", "Erosion Demo",
                  &erosion_size, max_kernel_size,
                  Erosion );

    /// Create Dilation Trackbar
    createTrackbar( "Element:\n 0: Rect \n 1: Cross \n 2: Ellipse", "Dilation Demo",
                  &dilation_elem, max_elem,
                  Dilation );

    createTrackbar( "Kernel size:\n 2n +1", "Dilation Demo",
                  &dilation_size, max_kernel_size,
                  Dilation );

    /// Default start
    Erosion( 0, 0 );
    Dilation( 0, 0 );

    waitKey(0);
    return 0;
}

/** @function Erosion */
void Erosion( int, void* )
{
    int erosion_type;
    if( erosion_elem == 0 ){ erosion_type = MORPH_RECT; }
    else if( erosion_elem == 1 ){ erosion_type = MORPH_CROSS; }
    else if( erosion_elem == 2 ) { erosion_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( erosion_type,
                                         Size( 2*erosion_size + 1, 2*erosion_size+1 ),
                                         Point( erosion_size, erosion_size ) );

```

```
/// Apply the erosion operation
erode( src, erosion_dst, element );
imshow( "Erosion Demo", erosion_dst );
}

/** @function Dilation */
void Dilation( int, void* )
{
    int dilation_type;
    if( dilation_elem == 0 ){ dilation_type = MORPH_RECT; }
    else if( dilation_elem == 1 ){ dilation_type = MORPH_CROSS; }
    else if( dilation_elem == 2) { dilation_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( dilation_type,
                                        Size( 2*dilation_size + 1, 2*dilation_size+1 ),
                                        Point( dilation_size, dilation_size ) );

    /// Apply the dilation operation
    dilate( src, dilation_dst, element );
    imshow( "Dilation Demo", dilation_dst );
}
```

15.4 Explanation

1. Most of the stuff shown is known by you (if you have any doubt, please refer to the tutorials in previous sections). Let's check the general structure of the program:

- Load an image (can be RGB or grayscale)
- Create two windows (one for dilation output, the other for erosion)
- Create a set of 02 Trackbars for each operation:
 - The first trackbar “Element” returns either **erosion_elem** or **dilation_elem**
 - The second trackbar “Kernel size” return **erosion_size** or **dilation_size** for the corresponding operation.
- Every time we move any slider, the user's function **Erosion** or **Dilation** will be called and it will update the output image based on the current trackbar values.

Let's analyze these two functions:

2. erosion:

```
/** @function Erosion */
void Erosion( int, void* )
{
    int erosion_type;
    if( erosion_elem == 0 ){ erosion_type = MORPH_RECT; }
    else if( erosion_elem == 1 ){ erosion_type = MORPH_CROSS; }
    else if( erosion_elem == 2) { erosion_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( erosion_type,
                                        Size( 2*erosion_size + 1, 2*erosion_size+1 ),
                                        Point( erosion_size, erosion_size ) );

    /// Apply the erosion operation
    erode( src, erosion_dst, element );
}
```

```
imshow( "Erosion Demo", erosion_dst );
}
```

- The function that performs the *erosion* operation is `erode`. As we can see, it receives three arguments:
 - *src*: The source image
 - *erosion_dst*: The output image
 - *element*: This is the kernel we will use to perform the operation. If we do not specify, the default is a simple 3×3 matrix. Otherwise, we can specify its shape. For this, we need to use the function `getStructuringElement`:

```
Mat element = getStructuringElement( erosion_type,
                                   Size( 2*erosion_size + 1, 2*erosion_size+1 ),
                                   Point( erosion_size, erosion_size ) );
```

We can choose any of three shapes for our kernel:

- * Rectangular box: `MORPH_RECT`
- * Cross: `MORPH_CROSS`
- * Ellipse: `MORPH_ELLIPSE`

Then, we just have to specify the size of our kernel and the *anchor point*. If not specified, it is assumed to be in the center.

- That is all. We are ready to perform the erosion of our image.

Note: Additionally, there is another parameter that allows you to perform multiple erosions (iterations) at once. We are not using it in this simple tutorial, though. You can check out the Reference for more details.

3. dilation:

The code is below. As you can see, it is completely similar to the snippet of code for **erosion**. Here we also have the option of defining our kernel, its anchor point and the size of the operator to be used.

```
/** @function Dilation */
void Dilation( int, void* )
{
    int dilation_type;
    if( dilation_elem == 0 ){ dilation_type = MORPH_RECT; }
    else if( dilation_elem == 1 ){ dilation_type = MORPH_CROSS; }
    else if( dilation_elem == 2 ) { dilation_type = MORPH_ELLIPSE; }

    Mat element = getStructuringElement( dilation_type,
                                       Size( 2*dilation_size + 1, 2*dilation_size+1 ),
                                       Point( dilation_size, dilation_size ) );

    /// Apply the dilation operation
    dilate( src, dilation_dst, element );
    imshow( "Dilation Demo", dilation_dst );
}
```

15.5 Results

- Compile the code above and execute it with an image as argument. For instance, using this image:



We get the results below. Varying the indices in the Trackbars give different output images, naturally. Try them out! You can even try to add a third Trackbar to control the number of iterations.

