Image Watch: viewing in-memory images in the Visual Studio debugger

Image Watch is a plug-in for Microsoft Visual Studio that lets you to visualize in-memory images (cv::Mat or IplImage_ objects, for example) while debugging an application. This can be helpful for tracking down bugs, or for simply understanding what a given piece of code is doing.

Prerequisites

This tutorial assumes that you have the following available:

1. Visual Studio 2012 Professional (or better) with Update 1 installed. Update 1 can be downloaded here.
2. An OpenCV installation on your Windows machine (Tutorial: Installation in Windows).
3. Ability to create and build OpenCV projects in Visual Studio (Tutorial: How to build applications with OpenCV inside the Microsoft Visual Studio).

Installation

Download the Image Watch installer. The installer comes in a single file with extension .vsix (Visual Studio Extension). To launch it, simply double-click on the .vsix file in Windows Explorer. When the installer has finished, make sure to restart Visual Studio to complete the installation.

Example

Image Watch works with any existing project that uses OpenCV image objects (for example, cv::Mat). In this example, we use a minimal test program that loads an image from a file and runs an edge detector. To build the program, create a console application project in Visual Studio, name it "image-watch-demo", and insert the source code below.

```
// Test application for the Visual Studio Image Watch Debugger extension

#include <iostream>                        // std::cout
#include <opencv2/core/core.hpp>           // cv::Mat
#include <opencv2/highgui/highgui.hpp>     // cv::imread()
#include <opencv2/imgproc/imgproc.hpp>     // cv::Canny()

using namespace std;
using namespace cv;

void help()
{
    cout
        << "----------------------------------------------------" << endl
        << "This is a test program for the Image Watch Debugger " << endl
        << "plug-in for Visual Studio. The program loads an     " << endl
        << "image from a file and runs the Canny edge detector. " << endl
        << "No output is displayed or written to disk."
        << endl
        << "Usage:"                                         << endl
        << "image-watch-demo inputimage"                    << endl
        << "-----------------------------------------------------" << endl
        << endl;
}

int main(int argc, char *argv[])
{
    help();

    if (argc != 2)
    {
        cout << "Wrong number of parameters" << endl;
        return -1;
    }

    cout << "Loading input image: " << argv[1] << endl;
    Mat input;
    input = imread(argv[1], CV_LOAD_IMAGE_COLOR);

    cout << "Detecting edges in input image" << endl;
    Mat edges;
    Canny(input, edges, 10, 100);

    return 0;
}
```

Make sure your active solution configuration (*Build ‣ Configuration Manager*) is set to a debug build (usually called "Debug"). This should disable compiler optimizations so that viewing variables in the debugger can work reliably.

Build your solution (*Build ‣ Build Solution*, or press F7).

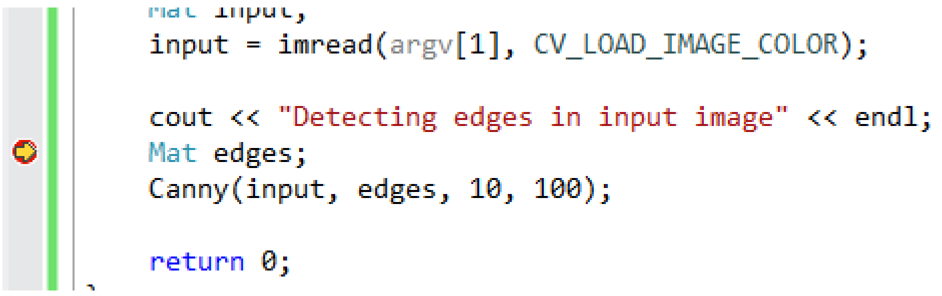
Before continuing, do not forget to add the command line argument of your input image to your project (*Right click on project ‣ Properties ‣ Configuration Properties ‣ Debugging* and then set the field Command Arguments with the location of the image).

Now set a breakpoint on the source line that says

```
Mat edges;
```

To set the breakpoint, right-click on the source line and select *Breakpoints ‣ Insert Breakpoint* from the context menu.

Launch the program in the debugger (*Debug ‣ Start Debugging*, or hit F5). When the breakpoint is hit, the program is paused and Visual Studio displays a yellow instruction pointer at the breakpoint:

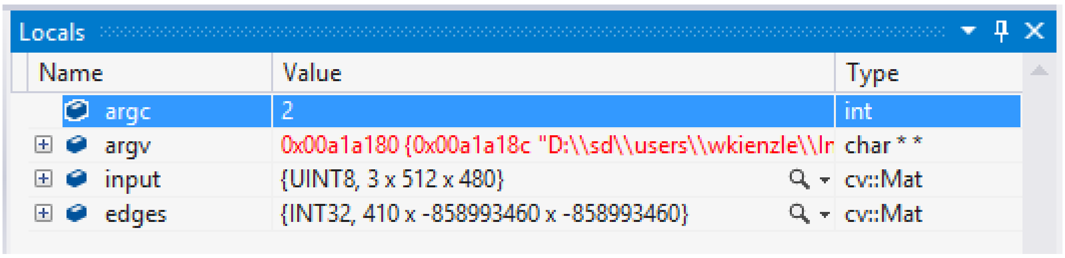


```
    Mat input;
    input = imread(argv[1], CV_LOAD_IMAGE_COLOR);

    cout << "Detecting edges in input image" << endl;
    Mat edges;
    Canny(input, edges, 10, 100);

    return 0;
```

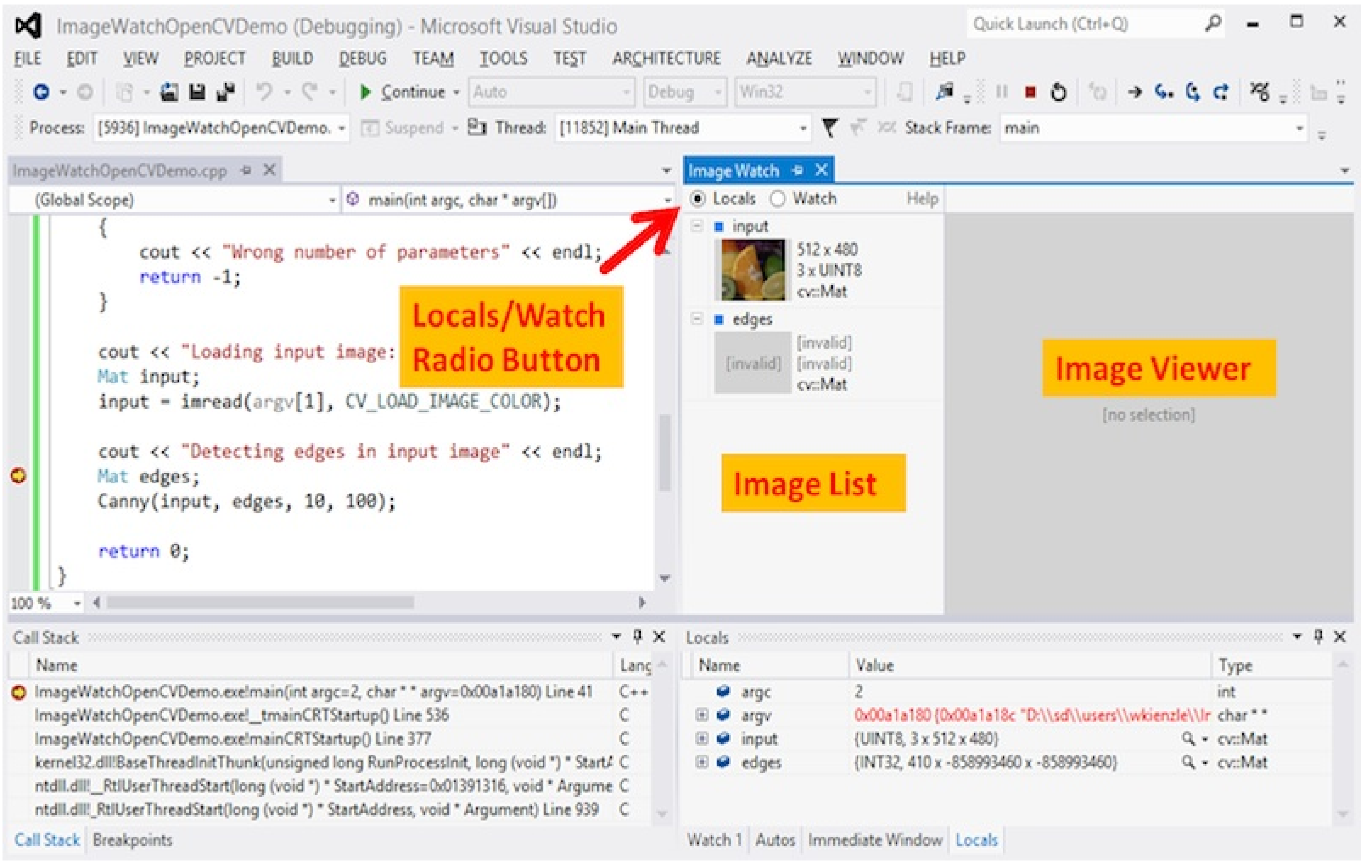
Now you can inspect the state of you program. For example, you can bring up the Locals window (*Debug ‣ Windows ‣ Locals*), which will show the names and values of the variables in the current scope:



Locals

Name	Value	Type
argc	2	int
argv	0x00a1a180 {0x00a1a18c "D:\\sd\\users\\wkienzle\\Ir	char * *
input	{UINT8, 3 x 512 x 480}	cv::Mat
edges	{INT32, 410 x -858993460 x -858993460}	cv::Mat

Note that the built-in Locals window will display text only. This is where the Image Watch plug-in comes in. Image Watch is like another Locals window, but with an image viewer built into it. To bring up Image Watch, select *View ‣ Other Windows ‣ Image Watch*. Like Visual Studio’ s Locals window, Image Watch can dock to the Visual Studio IDE. Also, Visual Studio will remember whether you had Image Watch open, and where it was located between debugging sessions. This means you only have to do this once – the next time you start debugging, Image Watch will be back where you left it. Here’ s what the docked Image Watch window looks like at our breakpoint:


ImageWatchOpenCVDemo (Debugging) - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Continue
Process: [5936] ImageWatchOpenCVDemo.
Thread: [11852] Main Thread
Stack Frame: main
ImageWatchOpenCVDemo.cpp
(Global Scope)
main(int argc, char * argv[])
Image Watch
Locals
Watch
Help
input
512 x 480
3 x UINT8
cv::Mat
edges
[invalid]
[invalid]
[invalid]
cv::Mat
[no selection]
Locals/Watch Radio Button
Image Viewer
Image List

```
    {
        cout << "Wrong number of parameters" << endl;
        return -1;
    }

    cout << "Loading input image:
    Mat input;
    input = imread(argv[1], CV_LOAD_IMAGE_COLOR);

    cout << "Detecting edges in input image" << endl;
    Mat edges;
    Canny(input, edges, 10, 100);

    return 0;
}
```

Call Stack

Name	Lang
ImageWatchOpenCVDemo.exe!main(int argc=2, char * * argv=0x00a1a180) Line 41	C++
ImageWatchOpenCVDemo.exe!__tmainCRTStartup() Line 536	C
ImageWatchOpenCVDemo.exe!mainCRTStartup() Line 377	C
kernel32.dll!BaseThreadInitThunk(unsigned long RunProcessInit, long (void *) * Start	C
ntdll.dll!__RtlUserThreadStart(long (void *) * StartAddress=0x01391316, void * Argume	C
ntdll.dll!_RtlUserThreadStart(long (void *) * StartAddress, void * Argument) Line 939	C

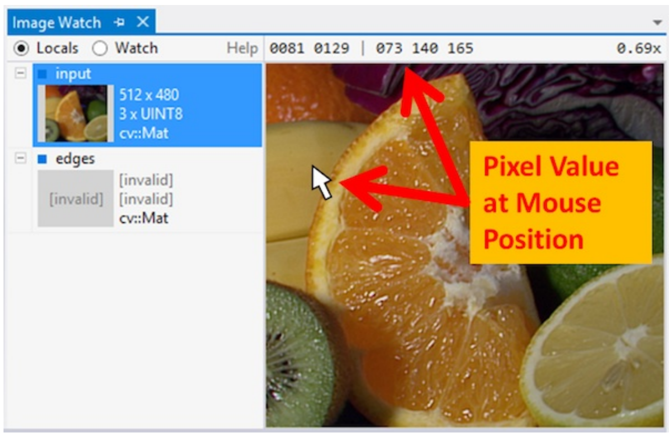
Locals

Name	Value	Type
argc	2	int
argv	0x00a1a180 {0x00a1a18c "D:\\sd\\users\\wkienzle\\Ir	char * *
input	{UINT8, 3 x 512 x 480}	cv::Mat
edges	{INT32, 410 x -858993460 x -858993460}	cv::Mat

Call Stack Breakpoints Watch 1 Autos Immediate Window Locals

The radio button at the top left (Locals/Watch) selects what is shown in the Image List below: Locals lists all OpenCV image objects in the current scope (this list is automatically populated). Watch shows image expressions that have been pinned for continuous inspection (not described here, see Image Watch documentation for details). The image list shows basic information such as width, height, number of channels, and, if available, a thumbnail. In our example, the image list contains our two local image variables, input and edges.

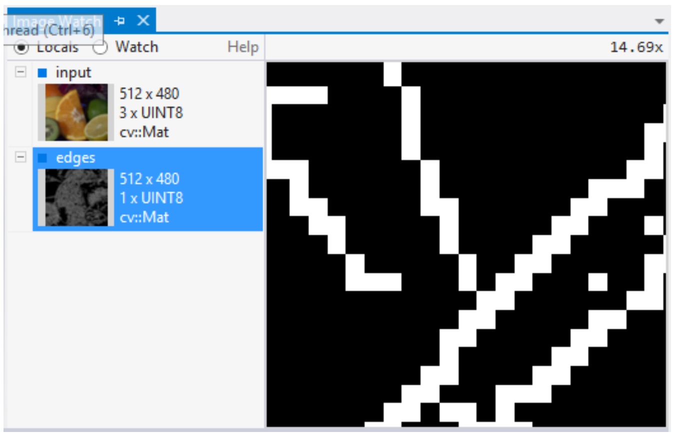
If an image has a thumbnail, left-clicking on that image will select it for detailed viewing in the Image Viewer on the right. The viewer lets you pan (drag mouse) and zoom (mouse wheel). It also displays the pixel coordinate and value at the current mouse position.



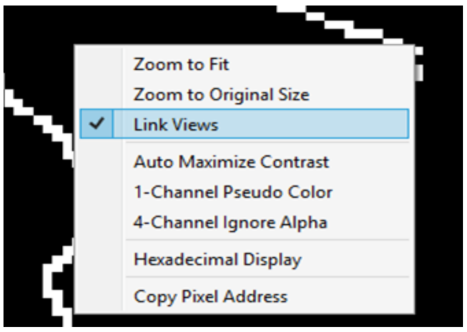
Note that the second image in the list, edges, is shown as "invalid". This indicates that some data members of this image object have corrupt or invalid values (for example, a negative image width). This is expected at this point in the program, since the C++ constructor for edges has not run yet, and so its members have undefined values (in debug mode they are usually filled with "0xCD" bytes).

From here you can single-step through your code (*Debug->Step Over*, or press F10) and watch the pixels change: if you step once, over the Mat edges; statement, the edges image will change from "invalid" to "empty", which means that it is now in a valid state (default constructed), even though it has not been initialized yet (using cv::Mat::create(), for example). If you make one more step over the cv::Canny() call, you will see a thumbnail of the edge image appear in the image list.

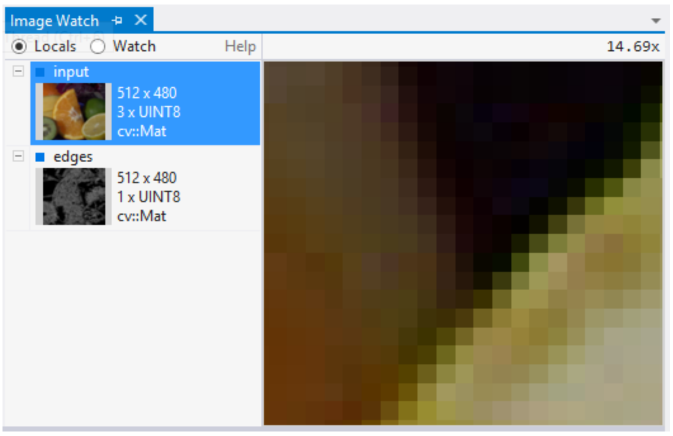
Now assume you want to do a visual sanity check of the cv::Canny() implementation. Bring the edges image into the viewer by selecting it in the Image List and zoom into a region with a clearly defined edge:



Right-click on the Image Viewer to bring up the view context menu and enable ***Link Views*** (a check box next to the menu item indicates whether the option is enabled).



The ***Link Views*** feature keeps the view region fixed when flipping between images of the same size. To see how this works, select the input image from the image list - you should now see the corresponding zoomed-in region in the input image:



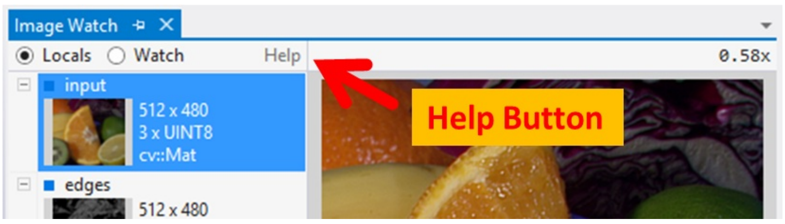
You may also switch back and forth between viewing input and edges with your up/down cursor keys. That way you can easily verify that the detected edges line up nicely with the data in the input image.

More ...

Image watch has a number of more advanced features, such as

1. pinning images to a Watch list for inspection across scopes or between debugging sessions
2. clamping, thresholding, or diff'ing images directly inside the Watch window
3. comparing an in-memory image against a reference image from a file

Please refer to the online Image Watch Documentation for details - you also can get to the documentation page by clicking on the Help link in the Image Watch window:



Help and Feedback

You did not find what you were looking for?

- Ask a question on the Q&A forum.
- If you think something is missing or wrong in the documentation, please file a bug report.