OpenCV Tutorial 1: Loading and Saving an Image

Welcome to my tutorial on how to load and save an image using OpenCV.

Introduction

You may be familiar with opening an image using an app: you double-click on the file and it appears in a window. Neat!

However, what if you want to do more than just look at an image, e.g. you want to resize it, apply Gaussian blurring, extract a region of interest, change its colour space, etc. That's where OpenCV is useful.

In this tutorial I show you how to use OpenCV's C++ library to load, display, and save an image.

The tutorial is organised as follows: I first describe the requirements for completing this tutorial; then I list the tutorial's contents and describe how to build its activities and samples; next, I go through each sample's header and source files, describing each line's and snippet's purpose; I then prescribe activities for you to complete; and lastly, I conclude the tutorial.

Requirements

Windows

To build the source code listed in this tutorial, you need to have the following on your computer:

1. CMake
2. OpenCV
3. Visual Studio

If you haven't got these installed, click on each of the links to go to their respective download websites. Download and run the relevant system installer for your computer, e.g. CMake's "cmake-3.13.3-win64-x64.msi", OpenCV's 4.0.1 "Win pack", and Microsoft's latest "Windows" installers for a 64-bit version of Windows 10.

If you'd like to use an alternative Integrated Development Environment (IDE) to edit code, consider Microsoft's Visual Studio Code. You'll find it a light-weight, and flexible, alternative to Visual Studio.

Once you've got CMake, OpenCV, and an IDE installed, you're ready to get started.

How to Build the Tutorial's Samples and Activities

This tutorial contains the following files:

1. activity_1/CMakeLists.txt
2. activity_1/main.hpp
3. activity_1/main.cpp
4. activity_1/data
5. activity_2/CMakeLists.txt
6. activity_2/main.hpp

7. activity_2/main.cpp
8. sample_1/CMakeLists.txt
9. sample_1/main.hpp
10. sample_1/main.cpp
11. sample_1/data/image.png
12. sample_2/CMakeLists.txt
13. sample_2/main.hpp
14. sample_2/main.cpp
15. README.md

Sample 1 contains source code that shows how to load an image from a sub-directory. Sample 2 shows how to save an image to a sub-directory. Activity 1 and Activity 2 are projects set up for you to complete the tutorial's activities with. Sample 1's 'data' sub-directory contains an input file.

Windows

The following describe how to build the tutorial's sample and activity using either: console commands or Visual Studio IDE.

Console Commands

To build a Debug version of a sample or actvity, browse to its directory and use the following commands:

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
cmake --build . --config Debug --target install
```

To run the executable, browse to the sample's or actvity's 'bin' directory.

To build a Release version of a sample or activity, browse to its directory and use the following commands:

```
mkdir build
cd build
cmake -G "Visual Studio 15 2017 Win64" ..
cmake --build . --config Release --target install
```

To run the executable, browse to the sample's or activity's 'bin' directory.

Visual Studio IDE

To build a Debug Version of a sample or activity:

1. Open Visual Studio IDE.
2. Click on 'File > Open > CMake'.
3. Browse to the sample's or activity's directory.
4. Select 'CMakeLists.txt' and click on Open.

5. Click 'CMake > Build All'.

To run the executable:

1. Click 'CMake > Debug from Build Folder > project'.

To change the build configuration:

1. Click 'CMake > Change CMake Settings > project'
2. Select the build configuration.

To build a 64-bit, debug application, select 'x64-Debug'. To build a 64-bit, release application, select 'x64-Release'. CMake's default is a x64-bit, debug configuration.

Sample 1: Loading an Image from a Directory

Browse to the 'sample_1' directory.

Let's have a look at Sample 1's source code:

'main.hpp'

```
#ifndef __MAIN_HPP__
#define __MAIN_HPP__

#include <opencv2/opencv.hpp>

#endif //__MAIN_HPP__
```

'main.cpp'

```
#include <filesystem>
#include <iostream>

#include "main.hpp"

namespace fs = std::experimental::filesystem;

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

    fs::path executablePath{ fs::path(argv[0]) };

    fs::path dataPath{ fs::path("/data/image.png") };

    fs::path inputFile{ executablePath.parent_path().append(dataPath) };

    std::cout << inputFile.string() << std::endl;

    cv::Mat image = cv::imread(inputFile.string(), cv::IMREAD_COLOR);

    if (image.empty()) {
        std::cout << "Error: 'image' is empty" << std::endl;

        return 1;
    }

    cv::imshow("image", image);

    cv::waitKey(0);

    cv::destroyAllWindows();

    return 0;
}
```

Let's first consider 'main.hpp'. I have used a header guard to prevent including a header file more than once. Header guards are conditional directives that take the form:

```
#ifndef __NAME__
#define __NAME__
    // declarations and definitions go here.
#endif __NAME__ //__NAME__
```

When 'main.hpp' is included, the first thing it does is check if `__MAIN_H__` has been defined. If it hasn't been, it defines `__MAIN_H__` and includes a header file. If it has, the entire header is ignored. For more information about header guards, see [here].

I have included the `opencv2/opencv.hpp` header file, which provides access to a range of OpenCV's modules, e.g. core, imgproc, and highgui. For more information about the modules, see [here].

Next, let's consider 'main.cpp'. I have included the `filesystem` and `iostream` header files, which provides facilities for performing operations on file systems and their components, e.g. paths, files, and directories, and access to input and output streams, e.g. `std::cin` and `std::cout`. For simplicity, I define an alias, `fs`, for the `std::experimental::filesystem` namespace.

Let's now go through the 'main.cpp' block by block:

The line

```
int main(int argc, char* argv[])
```

defines the program's entry point and has two parameters: `int argc` and `char* argv[]`. `argc` contains an integer number of command-line arguments, and `argv` contains a string of command-line arguments.

The line

```
fs::path executablePath{ fs::path(argv[0]) };
```

defines the variable `executablePath`, which is initialised with the executable's path. For more information about `std::experimental::filesystem::path`, see [here].

The line

```
fs::path dataPath{ fs::path("/data/image.png") };
```

defines the variable `dataPath`, which is initialised with the path to the input file's location.

The line

```
fs::path inputFile{ executablePath.parent_path().append(dataPath) };
```

defines the variable `inputFile`, which is initialised with the absolute path to the input file's location. Here, `executablePath` contains the executable's name, e.g. 'C:/main.exe'. By using the `parent_path()` member function, the parent path, e.g. 'C:/', is returned. Subsequently, using the `append()` member function, `dataPath` is appended to the parent path, e.g. 'C:/data/image.png'.

The line

```
std::cout << inputFile.string() << std::endl;
```

uses the output stream, `cout`, to display `inputFile`'s path on the console.

The line

```
cv::Mat image = cv::imread(inputFile.string(), cv::IMREAD_COLOR);
```

defines the variable `image`, which is initialised using OpenCV's `imread()` function. `imread()` has two parameters: `std::string &filename` and `int flags`. `filename` is the name of the file to be loaded and `flags` is the image read mode. Here, I have used the absolute path to the input file's location and will load the image as a three channel, Blue Green Red (BGR) color image. For more information about `imread()`, see [here].

The snippet

```
if (image.empty()) {

    std::cout << "Error: 'image' is empty" << std::endl;
    return 1;
}
```

checks to see if `image`'s `empty()` member function returns `true`. If it does, `imread()` could not find the input file; an error message is displayed; and a `return` statement terminates the program. If it does not, then the program continues.

The line

```
cv::imshow("image", image);
```

uses OpenCV's `imshow()` function to display `image` in a window. `imshow()` has two parameters: `const String &winname` and `InputArray mat`. `winname` is the name of the window and `mat` is the image to be shown. For more information about `imshow()`, see [here].

The line

```
cv::waitKey(0)
```

uses OpenCV's `waitKey()` function to wait for a user to press a key. `waitKey()` has one parameter: `int delay`. `delay` is the delay in milliseconds the function waits; 0 means "forever". For more information about `waitKey()`, see [here]

The snippet

```
cv::destroyAllWindows();
```

uses OpenCV's `destroyAllWindows()` function to close all open highgui windows. `destroyAllWindows()` has no parameters. For more information about `destroyAllWindows()', see [here].

The line

```
return 0;
```

terminates the program.

Now that we've looked at the sample's source code, let's build and run its executable.

You should see the following image be displayed:

Actvity 1: Load your own Image from a Directory

Browse to the 'activity_1' directory.

Now that you know how to load an image from a directory, complete the following activities:

1. Download and save an image in Activity 1's 'data' sub-directory. Bonus points for a funny meme.
2. Use a header guard to include the OpenCV header file.
3. Use an OpenCV function to read the image.
4. Use an OpenCV function to show the image in a window.
5. Use an OpenCV function to wait for a user to press a key.

Once you've completed these, build the activity's source code and run it's executable.

Take a screen shot of the displayed window.

Sample 2: Saving an Image to a Directory

Browse to the 'sample_2' directory.

Let's have a look at Sample 2's source code:

'main.hpp'

```
#ifndef __MAIN_HPP__
#define __MAIN_HPP__

#include <opencv2/opencv.hpp>

#endif //__MAIN_HPP__
```

'main.cpp'

```
#include <filesystem>
#include <iostream>

#include "main.hpp"

namespace fs = std::experimental::filesystem;

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

    cv::Mat image = cv::Mat(cv::Size(640, 480), CV_8UC3, cv::Scalar(255, 0, 0));

    cv::imshow("image", image);
    cv::waitKey(0);

    fs::path executablePath{ fs::path(argv[0]) };

    fs::path dataPath{ fs::path("/data") };

    fs::path outputPath{ executablePath.parent_path().append(dataPath) };

    if(!fs::is_directory(outputPath)) {

        fs::create_directory(outputPath);

    }

    fs::path outputFile {outputPath.append(fs::path("/image.png"))};

    std::cout << outputFile.string() << std::endl;

    cv::imwrite(outputFile.string(), image);

    return 0;
}
```

Let's first consider 'main.hpp'. I have used a header guard to prevent including a header file more than once. Header guards are conditional directives that take the form:

```
#ifndef __NAME__
#define __NAME__
    // declarations and definitions go here.
#endif __NAME__ //__NAME__
```

When 'main.hpp' is included, the first thing it does is check if `__MAIN_H__` has been defined. If it hasn't been, it defines `__MAIN_H__` and includes a header file. If it has been, the entire header is ignored. For information about header guards, see [here].

I have included the `opencv2/opencv.hpp` header file, which provides access to a range of OpenCV's modules, e.g. core, imgproc, and highgui. For more information about the modules, see [here].

Next, let's consider 'main.cpp'. I have included the `filesystem` and `iostream` header files, which provides facilities for performing operations on file systems and their components, e.g. paths, files, and directories, and access to input and output streams, e.g. `std::cin` and `std::cout`. For simplicity, I define an alias, `fs`, for the `std::experimental::filesystem` namespace.

Let's now go through the 'main.cpp' block by block:

The line

```
int main(int argc, char* argv[])
```

```
```

defines the program's entry point and has two parameters: `int argc` and `char* argv[]`. `argc` contains an integer number of command-line arguments, and `argv` contains a string of command-line arguments.

The line

```
cv::Mat image = cv::Mat(cv::Size(640, 480), CV_8UC3, cv::Scalar(255, 0, 0));
```

defines the variable `image`, which is initialised using an overloaded member function of OpenCV's `Mat` class . There are, to date, 29 different overloaded member functions available. Here, I've defined `image` as a 640 x 480 pixel (px), 8-bit, three channel image, whose pixels' colour is set by the Blue Green Red (BGR) scalar (255, 0, 0). For more information about the `Mat` class, see [here].

The line

```
cv::imshow("image", image);
```

OpenCV's `imshow()` function displays `image` in a window. `imshow()` has two parameters: `const String &winname` and `InputArray mat`. `winname` is the name of the window and `mat` is the image to be shown. For more information about `imshow()`, see [here].

The line

```
cv::waitKey(0)
```

OpenCV's `waitKey()` function waits for a user to press a key. `waitKey()` has one parameter: `int delay`. `delay` is the delay in milliseconds the function waits; 0 means "forever". For more information about `waitKey()`, see [here]

The line

```
fs::path executablePath{ fs::path(argv[0]) };
```

defines the variable `executablePath`, which is initialised with the executable's path. For more information about the `std::experimental::filesystem::path`, see [here].

The line

```
fs::path dataPath{ fs::path("/data") };
```

define the variable `dataPath`, which is initialised with the path to the output file's directory.

The line

```
fs::path outputPath{ executablePath.parent_path().append(dataPath) };
```

defines the variable `outputPath`, which is initialised with the absolute path to the output file's directory. Here, 'executablePath' contains the executable's name, e.g. 'C:/main.exe'. By using the `parent_path()` member function, the parent path, e.g. 'C:/' is returned. Subequently, using the `append()` member function, `dataPath` is appended to the parent path, e.g. 'C:/data'.

The snippet

```
if(!fs::is_directory(outputPath)) {

        fs::create_directory(outputPath);

}
```

checks to see if `outputPath`'s directory exists. If it does, the program continuese. If it doesn't, then the directory is created.

The line

```
fs::path outputFile {outputPath.append(fs::path("/image.png"))};
```

defines the variable `outputFile`, which is initialised with the absolute path to the output file's location. Here, `outputPath` contains the output file's directory. Using the `append()` member function, 'image.png' is appended to `outputPath` to create a file location.

The line

```
std::cout << outputFile.string() << std::endl;
```

uses the output stream, `cout`, to display `outputFile`'s path on the console.

The line

```
cv::imwrite(outputFile.string(), image);
```

uses OpenCV's `imwrite()` function to write an image to a directory. `imwrite()` has three parameters: `const string &filename`, `InputArray img`, and `const std::vector<int> &params`. `filename` is the name of the file to be written, `img` is the image to be written, and `params` is the image write flags. Here, I have used the absolute path to the output file's location and the image I originally created. For more information about `imwrite()`, see [here].

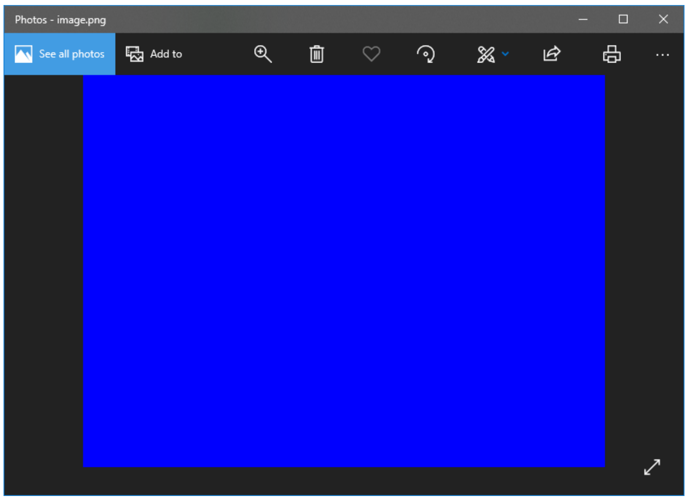
The line

```
return 0;
```

terminates the program.

Now that we've looked at the sample's source code, let's build and run its executable.

You should see the following image saved:



Activity 2: Save your own Image to a Directory

Browse to the 'activity_2' directory.

Now that you know how to save an image to a directory, complete the following activities:

1. Use a header guard to include the OpenCV header file.
2. Create a 320 x 240 px image of your own colour choice.
3. Use an OpenCV function to show the image in a window.
4. Use an OpenCV function to wait for a user to press a key.
5. Use an OpenCV function to save the image to a 'data' sub-directory.

Once you've completed these, build the activity's source code and run its executable.

Open the saved image using Windows' Photo App' and take a screen shot of the displayed image.

Conclusion

In this tutorial I've shown you how to use OpenCV's C++ library to load, display, and save an image.

You've used OpenCV's `imread()` function to load an image from a directory; `imwrite()` function to save an image to a directory; and `imshow()` to display an image in a window. You've also looked at using the `std::experimental::filesystem` to work with the system's components, e.g. paths, files, and directories.

I hope this tutorial has been helpful.

Credit

Dr Frazer K. Noble
Department of Mechanical and Electrical Engineering
School of Food and Advanced Technology
Massey University
New Zealand

Follow @drfrazernoble