OpenCV Tutorial 2: Working with an Image's Pixels

Welcome to my tutorial on how to access an image's pixels using OpenCV.

Introduction

You may be familiar with loading and saving an image with OpenCV: use OpenCV's functions `imread()` to load an image from a directory and `imwrite()` to save an image to a directory.

However, what if you want to access one of the image's pixels, e.g. you want to see if a pixel at point [10, 1] is red, or green, or blue. OpenCV's `Mat` class has a number of member functions that can help do that.

In this tutorial I show you how to use OpenCV's C++ library to access an image's pixels.

This 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 activity and sample; next, I go through the 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

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 Sample and Activity

Contents

This tutorial contains the following files:

1. activity_1/CMakeLists.txt
2. activity_1/main.hpp
3. activity_1/main.cpp
4. sample_1/CMakeLists.txt
5. sample_1/data/image.png
6. sample_1/main.hpp

7. sample_1/main.cpp
8. README.md

Sample 1 contains source code that shows how to get and set an image's pixel's value. Activity 1 is a project 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 console 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 console 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: Accessing an Image's Pixels

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.jpg") };

    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::Vec3b intensity = image.at<cv::Vec3b>(cv::Point(100, 100));

    uchar blue = intensity.val[0];
    uchar green = intensity.val[1];
    uchar red = intensity.val[2];

    std::cout << intensity << ", " << static_cast<int>(blue) << ", " <<
static_cast<int>(green) << ", " << static_cast<int>(red) << std::endl;

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

    cv::waitKey(0);

    image.at<cv::Vec3b>(cv::Point(100, 100)) = cv::Vec3b(100, 0, 0);

    intensity = image.at<cv::Vec3b>(cv::Point(100, 100));

    blue = intensity.val[0];
    green = intensity.val[1];
    red = intensity.val[2];

    std::cout << intensity << ", " << static_cast<int>(blue) << ", " <<
static_cast<int>(green) << ", " << static_cast<int>(red) << std::endl;

    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) colour 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::Vec3b intensity = image.at<cv::Vec3b>(cv::Point(100, 100));
```

defines the variable `intensity`, which is initialised using `image`'s overloaded `at()` member function. There are, to date, 12 different overloaded member functions available. Here, I've accessed `image`'s pixels at point [100, 100]. As `image` is a color image, `at()` returns a vector of three values: one for each colour channel. For more information about `at()`, see [here].

The snippet

```
uchar blue = intensity.val[0];
uchar green = intensity.val[1];
uchar red = intensity.val[2];
```

defines the variables `blue`, `green`, and `red` and initialises each with the corresponding colour channel's value from `intensity`. Here, as each pixel is ordered Blue-Green-Red, I've assined index [0] to `blue`, index [1] to `green`, and index [2] to `red`.

The line

```

```

```
std::cout << intensity << ", " << static_cast<int>(blue) << ", " <<
static_cast<int>(green) << ", " << static_cast<int>(red) << std::endl;
```

uses the output stream, `cout`, to display `intensity`'s, `blue`'s, `green`'s, and `red`'s values on the console.

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 line

```
image.at<cv::Vec3b>(cv::Point(100, 100)) = cv::Vec3b(100, 0, 0);
```

assigns the value [100, 0, 0] to `image`'s pixel at point [100, 100]. Here, we're changing the pixels values to a different colour. We can validate this change by using `image`'s overloaded `at()` member function.

The line

```
intensity = image.at<cv::Vec3b>(cv::Point(100, 100));
```

accesses `image`'s pixels at point [100, 100] and assigns the results to `intensity`.

The snippet

```
blue = intensity.val[0];
green = intensity.val[1];
red = intensity.val[2];
```

assigns `blue`, `green`, and `red` with the corresponding colour channel's value from `intensity`.

The line

```
std::cout << intensity << ", " << static_cast<int>(blue) << ", " <<
static_cast<int>(green) << ", " << static_cast<int>(red) << std::endl;
```

uses the output stream, `cout`, to display `intensity`'s, `blue`'s, `green`'s, and `red`'s values on the console.

The line

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

uses OpenCV's `imshow()` function to display `image` in a window.

The line

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

uses OpenCV's `waitKey()` function to wait for a user to press a key.

The line

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

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

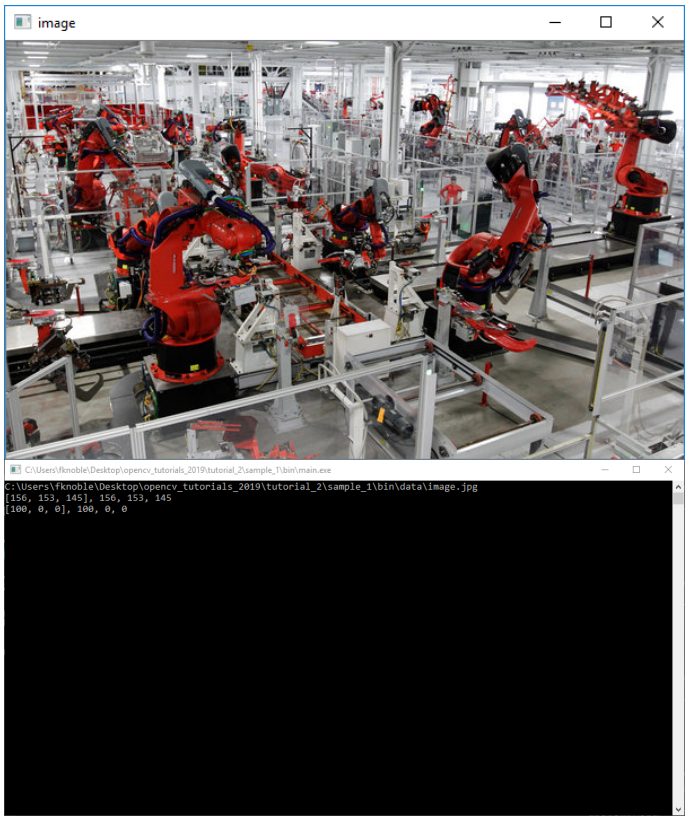
The line

```
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 and text displayed in a window and on the console, respectively:



Activity 1: Accessing the Pixels of Your Own Image

Browse to the 'activity_1' directory.

Now that you know how to access an image's pixels, complete the following activities:

1. Use a header guard to include the OpenCV header.
2. Create a 320 x 240 px of type CV_8UC1 with a value of your own choice.
3. Use an OpenCV function to show the image in a window.
4. Use an OpenCV function to access the image's pixel at point [100, 25].
5. Use an output stream to display the pixel's value on the console.

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

Take a screen shot of the displayed window and console output.

Conclusion

In this tutorial I have shown you how to use OpenCV's C++ library to access an image's pixels.

You've used OpenCV's `Mat` class's overloaded `at()` membership function to access an image's pixels at a specific point.

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