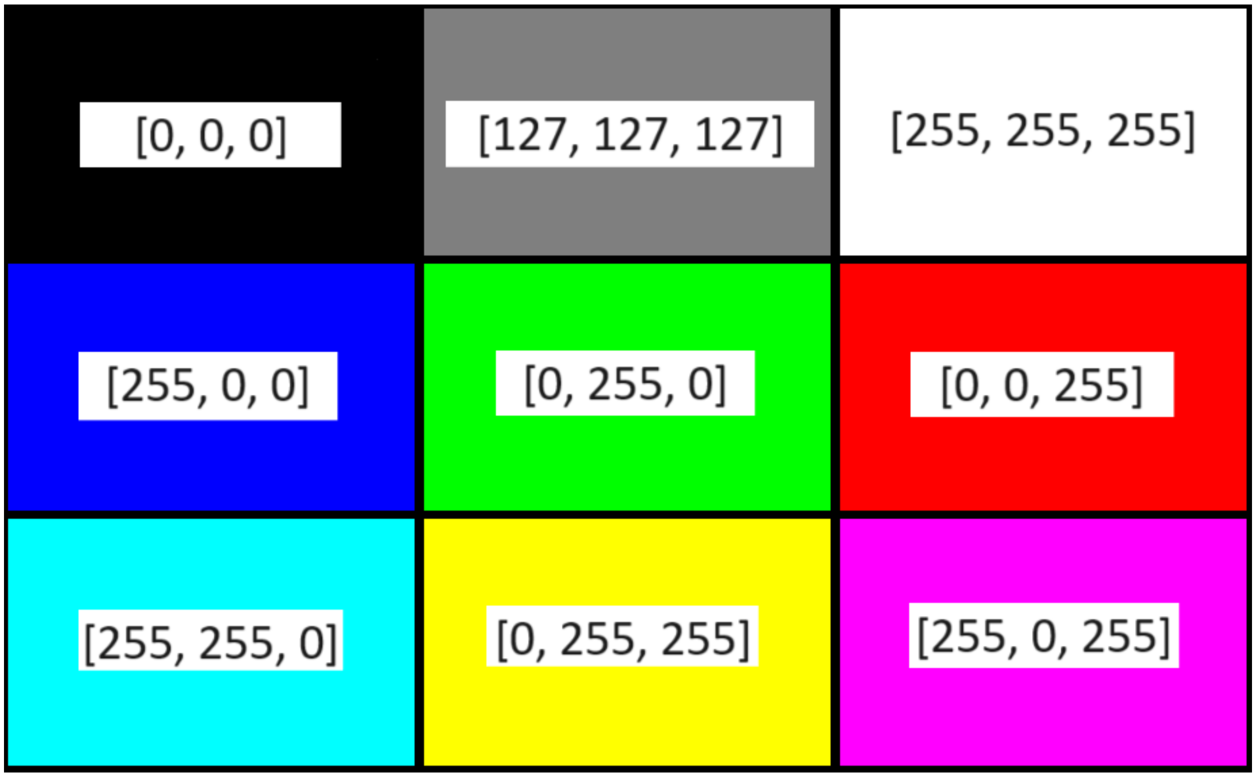
OpenCV C++ Tutorial

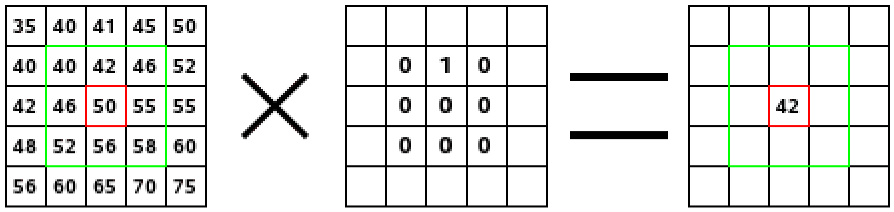
OpenCV is a powerful computer vision (CV) library for various languages including Python, Java, and C++. In this blog post I will detail how to use some simple functions to read images, capture live video, and process images and videos in C++ with the help of OpenCV. I will assume that you have already downloaded and installed OpenCV as well as a C++ compiler and some sort of editor. I am using MinGW on Windows with VSCode, and I will be using a Makefile to help compile my code.

The Mat Class

The *Mat* (matrix) class is the most fundamental part of the OpenCV library. It represents an n-dimensional array (commonly two or three dimensions) and is often used to store data relating to images. For color images, OpenCV uses the *BGR* color format (blue, green, and red) to represent the intensity of each color in a pixel. The scale on which intensity is measured is called *depth*, which can be represented by various data types (8-bit and 16-bit signed and unsigned integers, 32-bit signed integers, and 32-bit and 64-bit floating-point numbers). Mixing the values can produce various colors, as seen below with a depth of 8 bits (0 to 255, unsigned).



Grayscale images use one value rather than three to denote intensity. Grayscale images are easy to work with for image processing operations because there is only one value per pixel. One important operation is called *convolution*, which takes a larger matrix (the image) and a smaller matrix (the *kernel*, made of up of *weights*). For each element in the larger matrix, the element and its surrounding elements are multiplied element-wise with the kernel and summed. The result is the element of the new matrix created by this operation. A simple example is below:

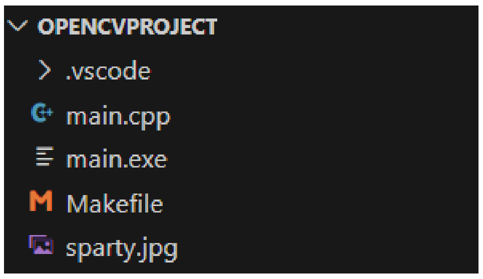


From https://docs.gimp.org/2.8/en/plug-in-convmatrix.html

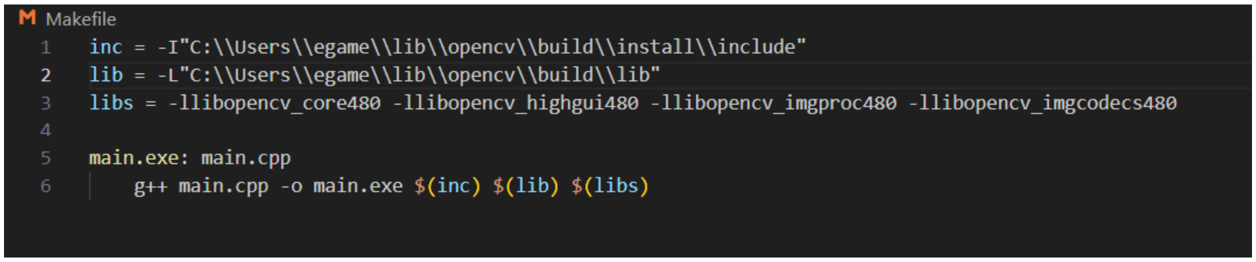
Different kernels create different effects on the image. To *blur* an image, simply use a kernel that averages the nearby pixels. For a 3x3 kernel, this would be a matrix consisting only of 1/9. Later in the tutorial, I will talk about *Sobel filters*, which highlights changes in intensity. This is useful for edge detection.

Code (Featuring the Core Module)

Let's begin coding. Below is every file in my folder, including the main C++ source file, its executable, a Makefile to help me compile the code, and an image. It doesn't really matter what you use, as long as it's in color. I downloaded a picture of a rather handsome mascot for my project. The reason I am using a Makefile is because to compile programs with OpenCV, it will be necessary to include several modules like *imgproc*, *imgcodecs*, *highgui*, and *core*. I find it easy to set some variables and occasionally change them to fit my needs. Below is my Makefile as an example; naturally, you should change the *inc* and *lib* variables to the paths associated with those folders on your machine.



```
OPENCVPROJECT
  .vscode
  main.cpp
  main.exe
  Makefile
  sparty.jpg
```



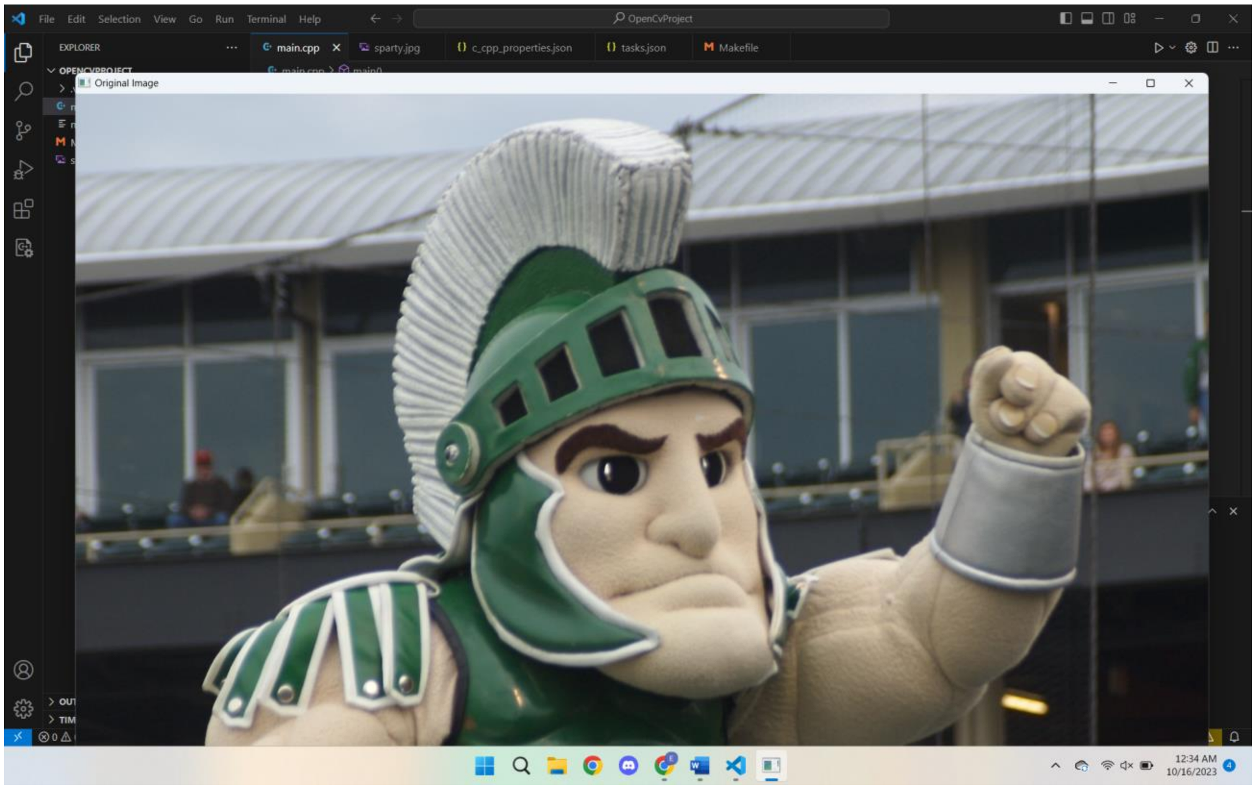
```
Makefile
inc = -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include"
lib = -L"C:\\Users\\egame\\lib\\opencv\\build\\lib"
libs = -llibopencv_core480 -llibopencv_highgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480

main.exe: main.cpp
	g++ main.cpp -o main.exe $(inc) $(lib) $(libs)
```

Whether you choose to compile with Makefiles, CMake, straight from the command line, or some other way, make sure the all the modules are included for this part. The names of the modules may differ depending on your version of OpenCV.

We will start simple to make sure OpenCV works. This code will simply create a Mat, read the image into it, and display it in a window.

```cpp
#include <iostream>
#include <opencv2/opencv.hpp>

int main() {
    cv::Mat image;
    image = cv::imread("sparty.jpg");

    if (image.empty()) {
        std::cerr << "Image not read" << std::endl;
        return 1;
    }

    cv::imshow("Original Image", image);
    cv::waitKey(0);
    return 0;
}
```

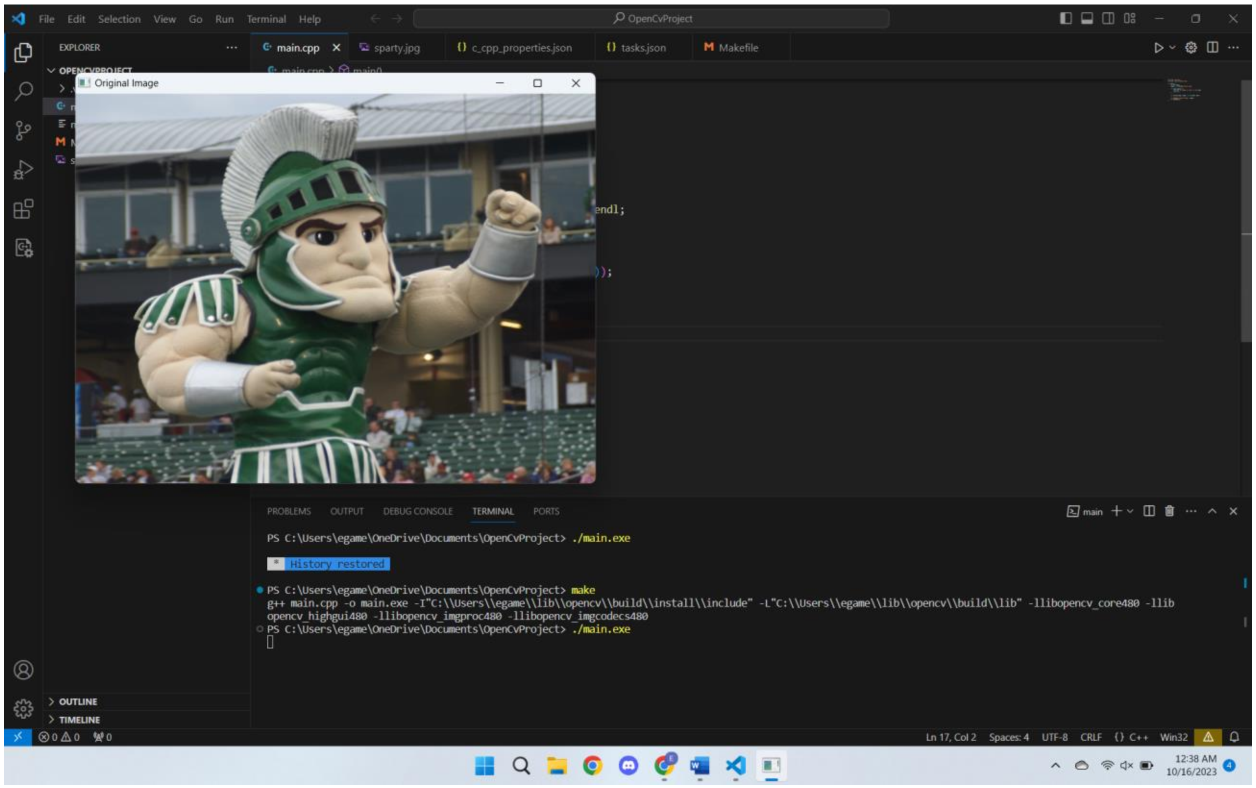
- *imread* converts the image data into the Mat data format for use in OpenCV
- Then, we check if the image is empty before trying to display it
- *imshow* opens a window containing the Mat so we can see it
- *waitKey* keeps the window open until a key is pressed. The argument is the delay in milleseconds, so with 0 the window closes immediately



The image I chose is a bit too large for my laptop display, which isn't helpful for the analysis we will do later. To remedy this, I will use the *resize* function before displaying the image.

```
    cv::resize(image, image, cv::Size(640, 480));
```

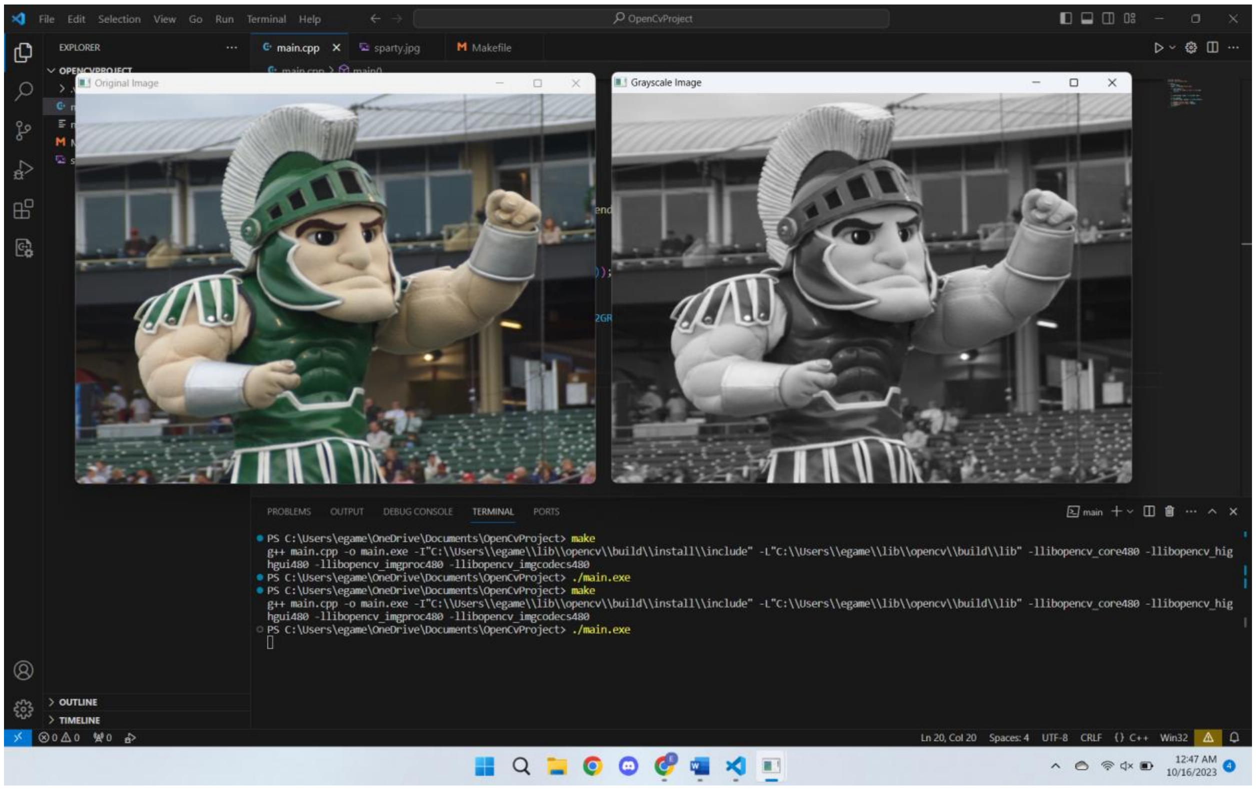
- The first argument is the *source* Mat, the Mat being read from
- The second argument is the *destination*, which can be a new Mat. I am using the same Mat for this example since I will have no need for the original Mat
- The third argument is a *Size* object, for which I am using magic numbers



```
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe

 *  History restored

PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llib
opencv_highgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
```

This looks nicer. Let's start modifying the image. To create a new image in grayscale and display both images at once, I'll replace some code at the end with:

```
    cv::Mat bwImage;
    cv::cvtColor(image, bwImage, cv::COLOR_BGR2GRAY);

    cv::imshow("Original Image", image);
    cv::imshow("Grayscale Image", bwImage);
    cv::waitKey(0);
    return 0;
```

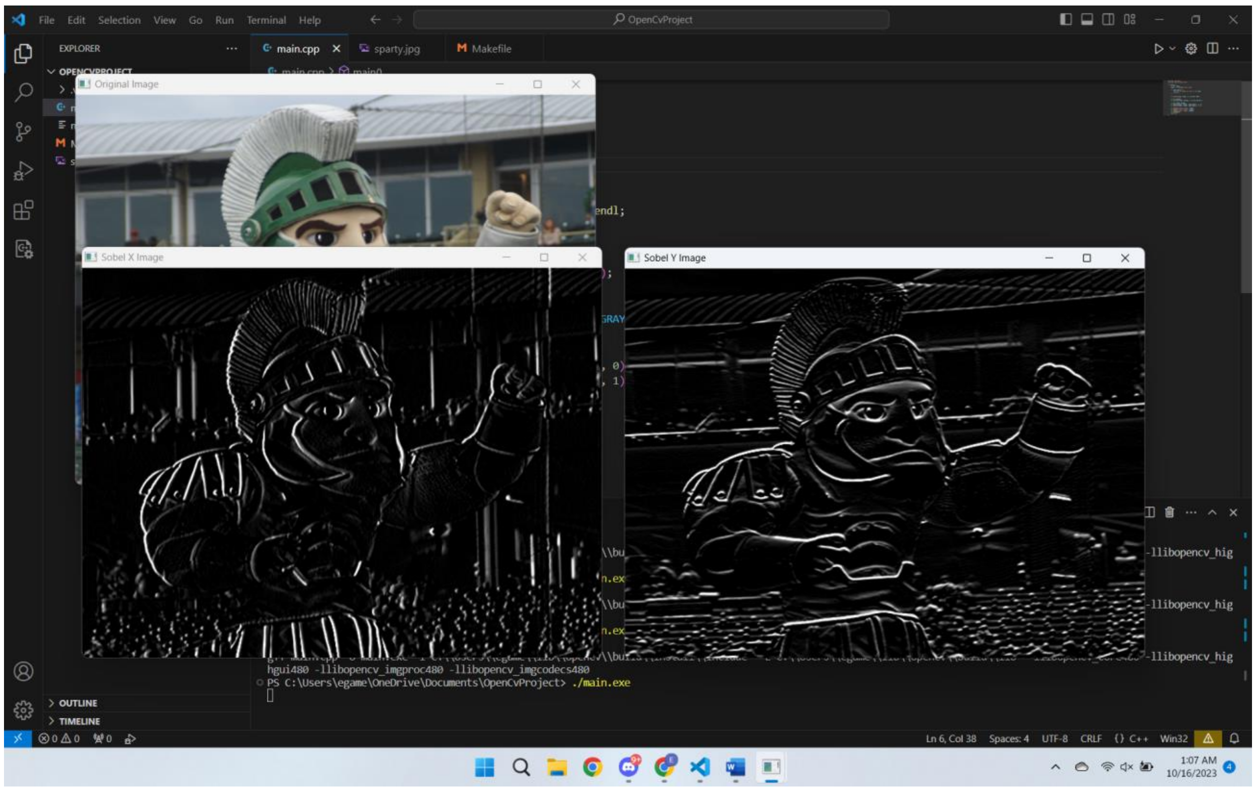


```
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
```

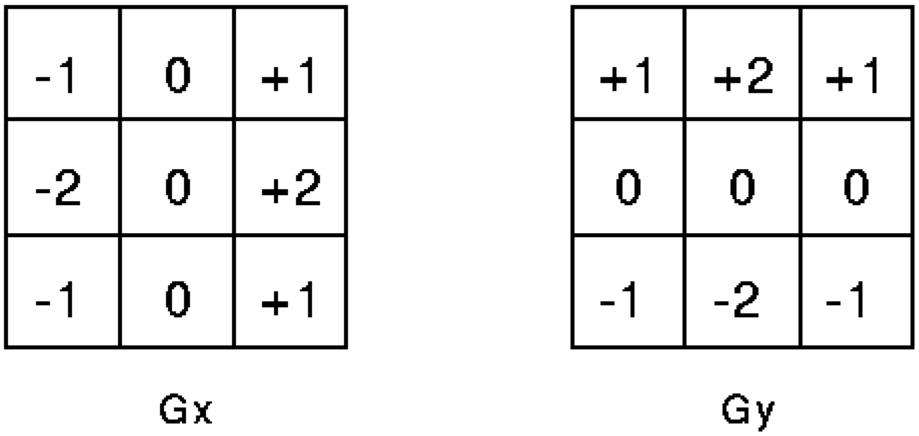
A good start. Now we will perform convolution with a Sobel filter on the grayscale image. Change the end of the main function to this:

```
    cv::Mat xImage, yImage;
    cv::Sobel(bwImage, xImage, image.depth(), 1, 0);
    cv::Sobel(bwImage, yImage, image.depth(), 0, 1);

    cv::imshow("Original Image", image);
    cv::imshow("Sobel X Image", xImage);
    cv::imshow("Sobel Y Image", yImage);
    cv::waitKey(0);
    return 0;
```

- The first argument of the Sobel function is the source, and the second is the destination
- The third argument is the depth. You can use the depth() function of a Mat for this
- The fourth and fifth arguments are dx and dy. I won't go into the calculus here, but know for now that Sobel filters should only be used in one direction (x or y) at a time



```
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
```



-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

From https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

Note the difference in the edges shown between the two images. This is because two different kernels are used in the convolution. This allows the components of the gradient magnitude and direction to be calculated. To finish this part, I will write the images to new files. To do so, add this to the end of the main function:

```
    cv::imwrite("sparty_x.png", xImage);
    cv::imwrite("sparty_y.png", yImage);
```

This concludes the beginning part of the tutorial. This is what all my code looks like:

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main() {
    cv::Mat image;
    image = cv::imread("sparty.jpg");

    if (image.empty()) {
        std::cerr << "Image not read" << std::endl;
        return 1;
    }

    cv::resize(image, image, cv::Size(640, 480));

    cv::Mat bwImage;
    cv::cvtColor(image, bwImage, cv::COLOR_BGR2GRAY);

    cv::Mat xImage, yImage;
    cv::Sobel(bwImage, xImage, image.depth(), 1, 0);
    cv::Sobel(bwImage, yImage, image.depth(), 0, 1);

    cv::imshow("Original Image", image);
    cv::imshow("Sobel X Image", xImage);
    cv::imshow("Sobel Y Image", yImage);
    cv::waitKey(0);

    cv::imwrite("sparty_x.png", xImage);
    cv::imwrite("sparty_y.png", yImage);
    return 0;
}
```

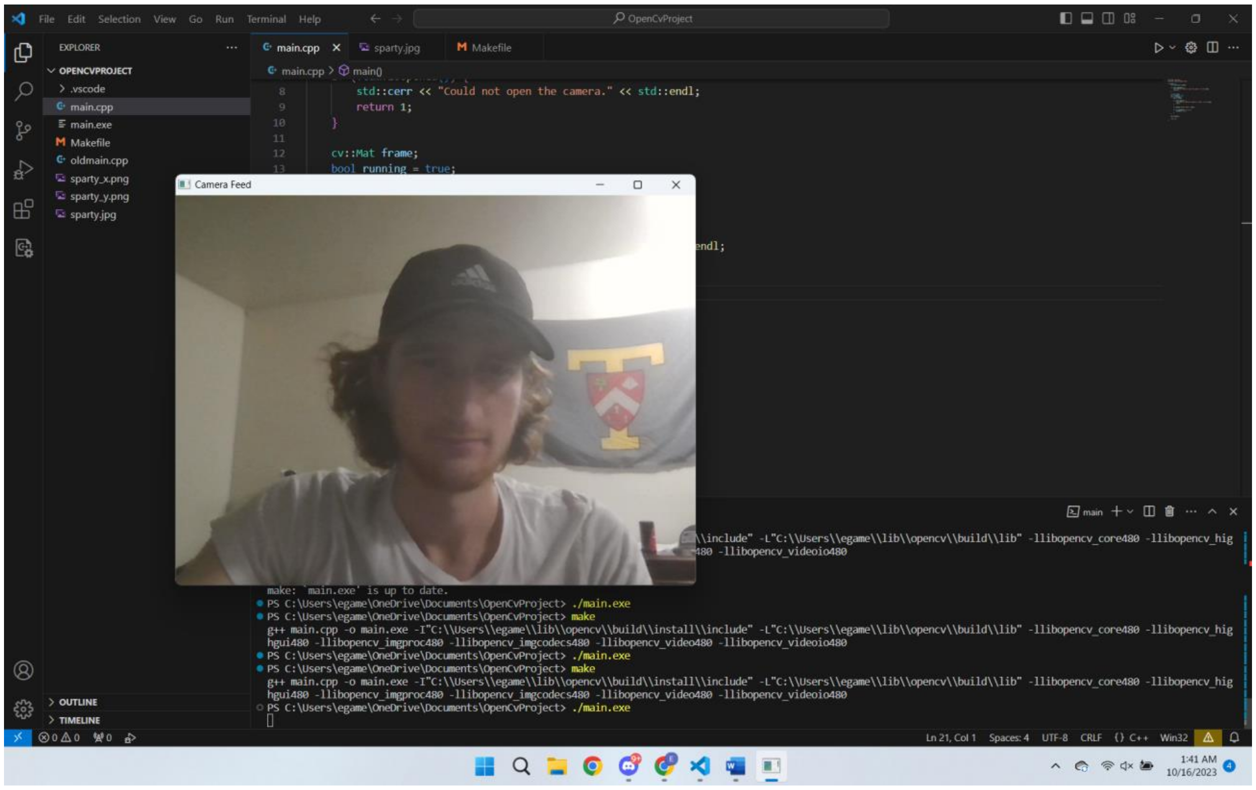
By now, you should understand Mat objects, how to read and write images, and how to perform some basic operations on Mats.

Live Video

For this part, you will need to use the *video* and *videoio* modules in addition to the ones used in the previous part. This will allow you to process and capture video. Start with this code:

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main() {
    cv::VideoCapture cam(0);

    if (!cam.isOpened()) {
        std::cerr << "Could not open the camera." << std::endl;
        return 1;
    }

    cv::Mat frame;
    bool running = true;
    while (running) {
        cam >> frame;

        if (frame.empty()) {
            std::cerr << "Could not capture a frame." << std::endl;
            break;
        }

        cv::imshow("Camera Feed", frame);

        if (cv::waitKey(1) == 'q') {
            running = false;
        }
    }

    cam.release();

    return 0;
}
```

- The *VideoCapture* constructor takes a device ID, the default laptop camera should be 0
- Use *cam >> frame* to place the current frame from the camera into the frame Mat
- Repeatedly use *imshow* to update the window
- The *waitkey* function returns an integer corresponding to the key pressed on the keyboard. Adding a delay of one millisecond allows the program to read input from the keyboard
- Release the camera at the end of the program, to be clean

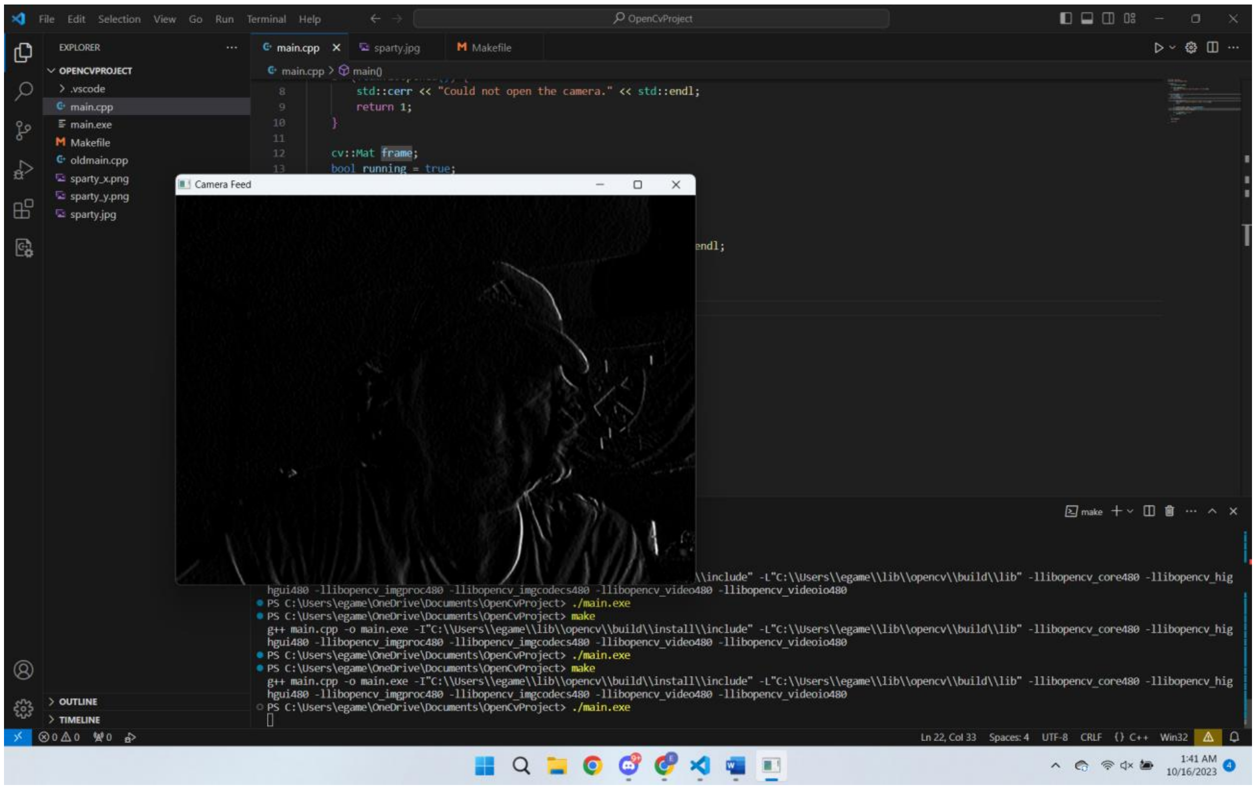


```
        std::cerr << "Could not open the camera." << std::endl;
        return 1;
    }

    cv::Mat frame;
    bool running = true;
```

```
\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
480 -llibopencv_videoio480

make: `main.exe' is up to date.
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480 -llibopencv_video480 -llibopencv_videoio480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480 -llibopencv_video480 -llibopencv_videoio480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
```

Hey, that's me. I promise the window shows video. Normal video gets boring after a while, so let's apply some filters. After capturing the frame but before placing it in the window, change the Mat to grayscale add the familiar Sobel filter with this code:

```
        cv::cvtColor(frame, frame, cv::COLOR_BGR2GRAY);
        cv::Sobel(frame, frame, frame.depth(), 1, 0);
        cv::imshow("Camera Feed", frame);
```



```cpp
        std::cerr << "Could not open the camera." << std::endl;
        return 1;
    }

    cv::Mat frame;
    bool running = true;
```

```
\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480 -llibopencv_video480 -llibopencv_videoio480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480 -llibopencv_video480 -llibopencv_videoio480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> make
g++ main.cpp -o main.exe -I"C:\\Users\\egame\\lib\\opencv\\build\\install\\include" -L"C:\\Users\\egame\\lib\\opencv\\build\\lib" -llibopencv_core480 -llibopencv_hig
hgui480 -llibopencv_imgproc480 -llibopencv_imgcodecs480 -llibopencv_video480 -llibopencv_videoio480
PS C:\Users\egame\OneDrive\Documents\OpenCvProject> ./main.exe
```

By now, you should understand how to use the VideoCapture class to use a camera for live video and how to perform operations on frames from the capture.

Conclusion

OpenCV is a big library with many functionalities. In the functions I used throughout this tutorial, there are several optional parameters which I chose not to detail for the sake of simplicity. Even then, this is only a small corner of the library; OpenCV includes modules for 3D reconstruction, feature detection, deep neural networks, object detection, iOS support, CUDA support, and much more. My goal with this tutorial was to provide the fundamentals of OpenCV and computer vision to enable further learning about the library.