CompE 571 - Embedded Operating Systems
FALL 2017

Final Project Report on:

Fire Detection via Image Processing Technique using Raspberry Pi

Submitted By :

Luis Camal
luis.e.camal@gmail.com
RED ID : 822093448

Anup N. Kirtane
akirtane@sdsu.edu
RED ID : 822057802

San Diego State University

ACCOMPLISHMENTS

HW Components

Our implementation for our embedded device consisted of a Raspberry Pi 3 and a video camera to provide live feed. Our initial camera chosen was a GoPro, but due to updates on its software the compatibility to interface it with the Raspberry Pi was limited. We then chose to opt for the Logitech c170 Webcam. Our final project successfully interfaces the Logitech c170 with the Raspberry Pi.

Software Components

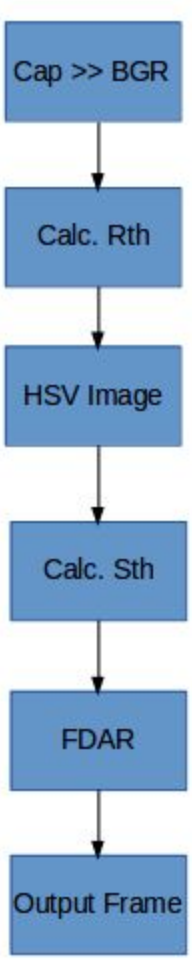

Figure 1

Since we were working on an image processing project, we wanted to take advantage of writing our code in C++, but also make use of the openCV Libraries that have been provided via open source development. Within C++ we also took advantage of libraries such as pthread, vector, unistd, and a few others.

The algorithm we chose to detect a fire makes use of a video frame's BGR (blue,green, red) color space and HSV (hue, saturation, value) color space. The fire detection algorithm must satisfy three main rules that are listed as follows:

1. $R > R_{Threshold}$
2. $R \geq G > B$
3. $S \geq ((255 - R) * S_{Threshold} / R_{Threshold})$

* three rules that must be satisfied for fire detection pixels

In our earlier report we explained the theory in depth. Here we will look into how we applied this theory via software using flow diagrams.

For our project we first implemented a default code that completed the fire detection algorithm serially. Figure 1 shows a graphic representation of the algorithm followed in in our default program. We first take an input frame from the live video and capture that frame in the BGR space, which captures pixels in a frame with three different channel colors known as blue, green, and red. The BGR image is then passed on to a function that computes a red pixel threshold. In figure 1 it is denoted as Rth. Rth is computed by taking the average of all the red pixels found in the video frame. Then a HSV image is computed using the BGR image. Once we have the BGR image and the HSV image, we can then calculate a saturation threshold denoted in our code as Sth. Once we have the red threshold and the saturation threshold we are able to compute the algorithm to detect fire pixels. The frame is then

considered to be processed and a frame with fire detection is at the output as can be seen in Figure 2.

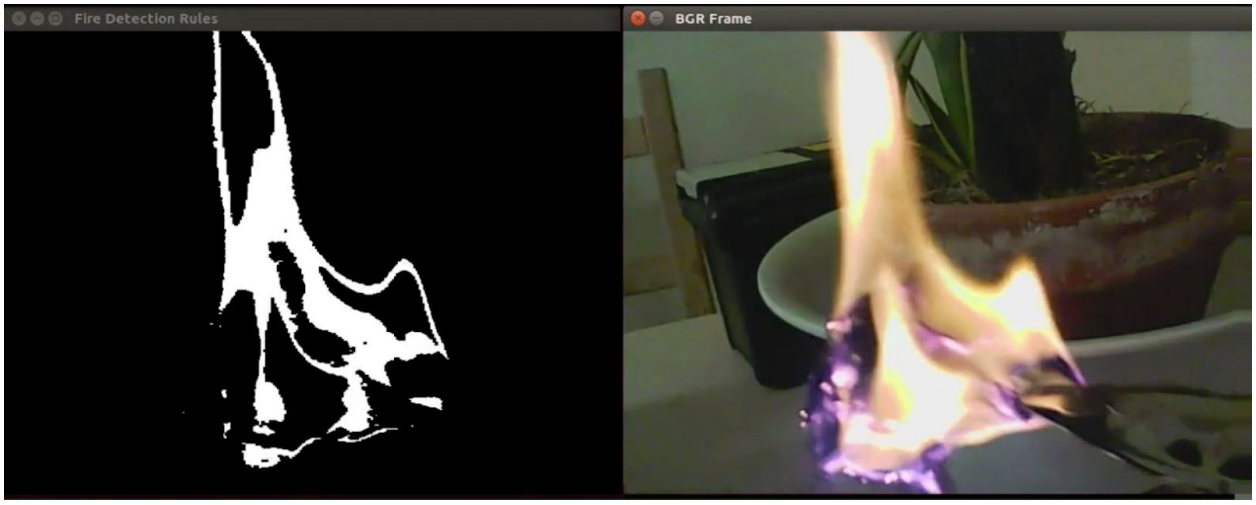

Figure 2

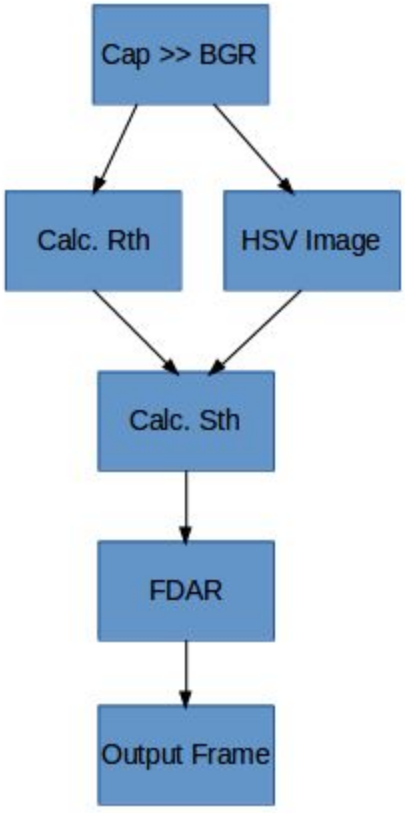

Figure 3

Running the default algorithm on the Raspberry Pi, gave us a maximum response time of about 0.6380 seconds, a minimum response time of about 0.5805 seconds, and an average response time of about 0.5944 seconds. In the live video there was quite bit of latency due to our response time being high.

Analyzing our default algorithm we noticed that two of our states do not depend on each other and they could run in parallel to each other. The states that were picked to run in parallel were the task that compute the HSV image and the task that calculates the red threshold value. A pictorial representation of this computation is demonstrated in figure 3. Looking at the state diagram we see that computing the red threshold and the HSV image both depend on the BGR image being captured. However, they don't depend on each other and thus can run in parallel. These two computations are also essential to the rest of the algorithm. Without these two tasks completed it is not possible to move to the next state and calculate the saturation threshold, because the saturation threshold is computed having hold of the red threshold and the HSV image. The calculation of Sth, the saturation threshold, is dependent on the red threshold and HSV image because to

calculate Sth we take only the saturation values, which are found in the HSV image, for the same index red pixel of whose values should be greater than the Rth, the red threshold. After Sth is computed the rest of the algorithm must run serially because each next task depends on the previous. Note that the last important part of the algorithm is to detect fire pixels using the three rules listed previously. In the state diagram we denote this as FDAR, which means fire detect all rules. Lastly the processed image is displayed at the output as shown in figure 2. Figure 2 shows the output frame on the left, and the input BGR frame on the right.

We then thought of comparing the default algorithm with modified implementations of the default algorithm to see if there would be any improvements or advantages to the response time of the system. To accomplish this we revisited topics we covered in class about threads and processes. We created two modifications of our algorithm in which one made use of threads and the other made use of fork(). We hoped that by utilizing these tools we could improve the response time of our system. Next we will examine the difference between the default implementation of the algorithm in comparison to the thread implementation and the fork implementation of the algorithm.

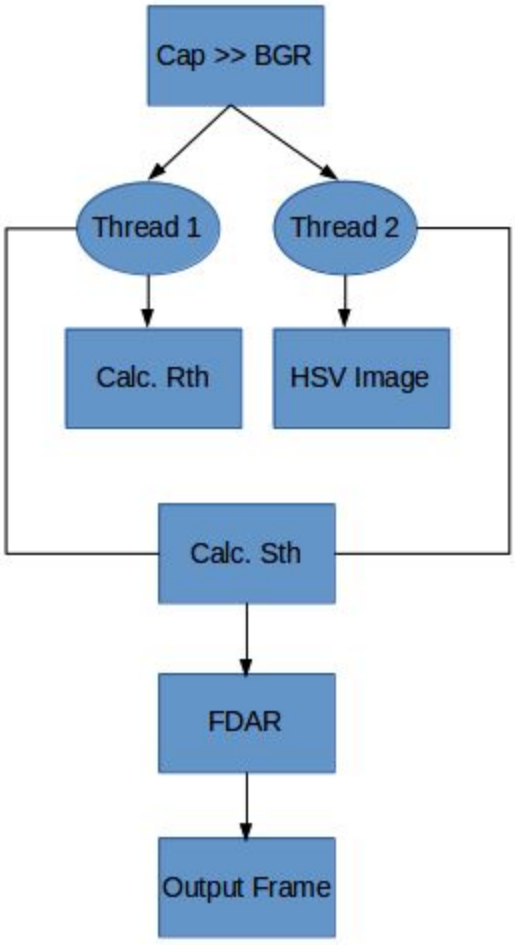

Figure 4

The main changes done in our thread based framework revolved on executing the Rth computation and the HSV computation in parallel. To better understand what is happening in the thread based program, let us use figure 4.

The main difference in the thread framework is what happens after the BGR frame is captured. Once obtaining this BGR image two threads are instantiated by calling the thread constructor. The threads run simultaneously and run in a multicore environment. After running these tasks on threads, they could run independent of the main thread, or independent of what is happening in the main function. But our state diagram in figure 3 shows that to successfully compute the fire detection, these task are essential to other tasks in the state diagram because those tasks depend on Rth and the HSV image. In our implementation we used a thread function that helps us come back to the main thread or program to finish the fire detection algorithm. This function is known as the join() function in multithreading. As can be seen in figure 4, the threads run but after execution they come back to the main thread to later compute the rest of the fire detection

algorithm serially. It is important to recognize that we are able to resume computation in the main program because of the use of the join function.

Let us now analyze the framework of the fork implementation of the algorithm and refer to figure 5. It is important to know how the fork() function works. Calling the fork() creates another process separately from the current process from which it was called by or created by. The process that calls the fork() function is called the parent process, and the new process created is called the child process. When a child process is created, it can run in parallel to the parent process. However, using the pid of the parent process and child process, we can differentiate between parent and child processes. If we do not want to run the parent process in parallel with the child process we must call the wait() function for the parent process.

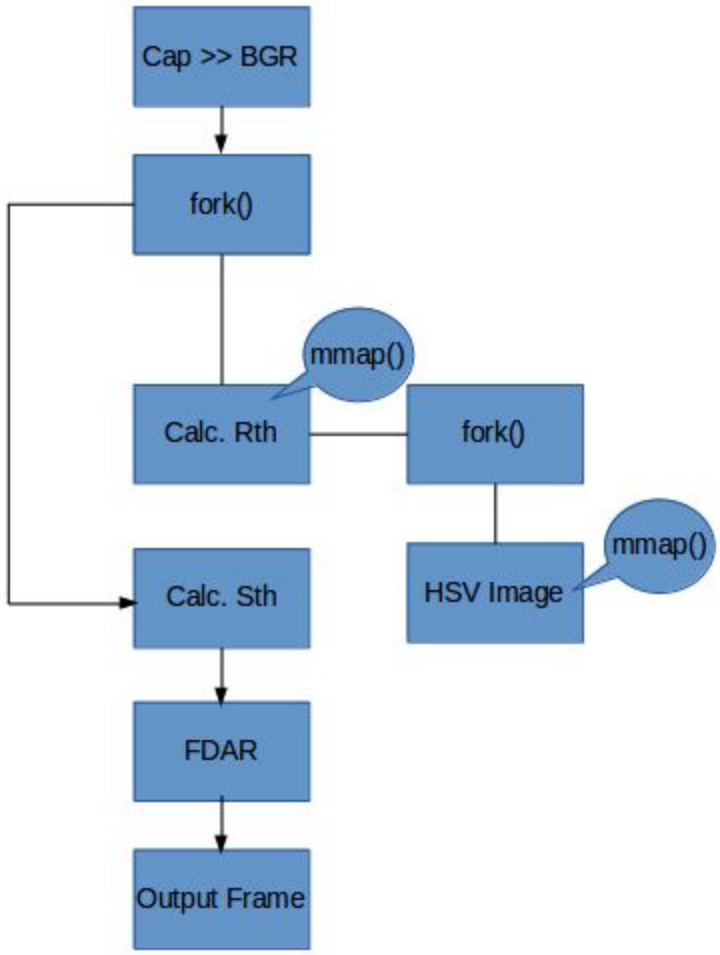

Figure 5

For the implementation of our code we want to execute two processes in parallel. That is compute the Rth value and the HSV image. Therefore, after capturing the BGR image in our main process, that is the main function, we call fork(). As seen in figure five, after fork() is called in the main process, a child process begins that creates a process that computes the Rth. However, there is a wait() declared for the code block that is in the parent process. The Sth computation and the FDAR computation cannot begin until the child process is done. Now looking at the child process that calls on the computation for Rth, another fork() is called. This fork creates yet another child, which we will call child two, and whose parent is the previous child, which we will now call child one. Therefore, child one calls the Rth computation and child two call on the HSV image computation. The second fork() call differs from the first because in the second one no wait function is called on the parent process, that is child one process. Hence, this gives us the ability to run both the child one, the Rth computation, and child two, the HSV image

computation, in parallel. After the children are done running. The processes are finished and they return to the main process or the parent process. Once we are at the parent process. We are able to continue the implementation of fire detection serially as before.

EXPERIMENTAL DESIGN

Metrics of success

The main goal of our project is to detect a fire mass. This is done by capturing an image frame for the region of interest and then using Image processing to validate a real fire. Our project implementation is successfully able to to detect the fire by taking advantage of an image's color space. It is important to first break that image into pixels and gain specific information from those pixels. We also wanted to overcome the problem of false pixels, and we achieved this by using an iterative logic where we calculated the difference between each real-time processed frame and a previously processed frame. Furthermore, as promised in our progress report we wanted to optimize the response time factor of the algorithm using the concepts taught in class about multitasking. Thus, we implemented our fire detection algorithm in three different process structures whose results were compared to the default algorithm. We were able to see some solid and evident results which actually justified the theory behind the concepts used.

Following are the details of metrics about our implementation results:

1. *Calculating Red pixel threshold* - The 1st condition of validation of fire pixels mentioned previously, states that 'R > Rth', where Rth is the the red pixel threshold. We used a BGR image frame and iterated through all the color pixels to determine the Rth value. The Rth values is the average of all the red pixels calculated in frame. Since a fire mass contains a concentrated Red color, we can use this calculation to determine a fire mass.
2. *Converting the image frame from BGR color space to HSV colorspace and calculating Saturation threshold* - The 2nd condition of validation of fire pixels says that R > G > B to threshold out the unwanted pixels. We obtained this by converting the color space from B-G-R (Blue-Green-Red) to H-S-V (Hue-Saturation-Value) and then setting the hue value in the range of 0 - 30 degree. After that, we calculated Sth, the saturation threshold, value by iterating through the HSV image frame and obtaining an average over all saturation pixels whose red pixel value was greater than the red threshold value.
3. *Using multitasking* - We had scope of optimization in our algorithm by using a multitasking approach. We implemented our fire detection algorithm with three different process structures and then compared their time responses. We were able to observe some solid parameters justifying the theoretical concept behind those implementations.
4. *Time response* - We saw that among all, multi-threading approach gives us a improvement in time response and fork() implementation takes more time than the default, i.e. serial processing implementation. Table 1 gives us an idea about time responses with different implementations.

Results

Fire Detection Algorithm Response Time in us

	Minimum Response Time	Maximum Response Time	Average Response Time	Percent Improvement to Avg. Time Response
Default	580512	638023	594434	N/A
Thread	554834	623107	584371	+ 1.69%
Fork	976424	1054870	993188	- 67.08%

Table 1

1. Our project is able to successfully detect the fire mass using the given approach. Also, to counter the drawback of false pixels / false image we implemented a logical algorithm in our program where we are taking into consideration the dynamicity of fire i.e. as the fire flame moves there is a difference between the pixel values for different instances i.e. frames, so we are comparing each processed frame with the previously processed one and validating that it's a real fire by taking the difference between them. Figure 6 shows our progress in eliminating false fire pixels.

BGR Frame

Difference Frame with elimination of false pixels

Processed Frame without elimnation of false pixels

Figure 6

2. Furthermore, to optimize our algorithm's performance in terms of response time we successfully implemented our algorithm with three different process structures, they are:

a) Serial Processing : As shown in the time response table in success of metrics, we saw that serial processing takes around 0.59 seconds to process the fire detection algorithm which is actually appreciable considering it to be a real time operating system. This is because there are no dependencies and shared memory/data involved in such type of process structure execution. The functions are been called serially one after the another and this time value will depend from processor to processor.

b) Multi-threading : From the table again, we can see that using multi-threading approach takes around 0.58 seconds to process the fire detection algorithm which is actually an improvement than previous implementation. With the multithreading there are few factors to be taken into consideration such as which task can run as a single independent thread, taking care of the shared memory/resource when threads are running parallely, thread creation overhead, joining back the threads with the 'main' thread. In our case, we had two threads, one calculating Rth and the other one converting a BGR image to HSV image. Theoretically, creating threads has overhead. Even after creating such overheads, the multi-threading implementation was able to execute faster than the serial processing which actually justifies the fact that multi-threading is a better approach in respect to response time when there are multiple calculations in a program.

c) fork() implementation : We can see that using fork() implementation had a huge toll on the implementation of our algorithm. In comparison to creating a thread, the fork() functions creates different processes. And as taught in class when we create different processes we know that there is going to be quite a bit of overhead. The system has to go through context switches. In our program we call the fork() function twice, and three total processes are created. Also in the use of our fork() function, we learned that variables that want to be passed down and modified between parent and child processes must be mapped directly to memory. For that we made use of the mmap() function. As seen in figure 5, mmap() is used for the tasks that obtain the Rth and the HSV image. We had to map the Rth and the HSV image to memory so that both children and parent processes could access the results given by those tasks. However, mapping directly to memory is also overhead and creates delay in creating such memory blocks. Allocating the memory is only one job that the children processes were involved in. But the parent process to use this memory it had to go seek it in memory. We also learned that obtaining data from memory has a great time cost. Due to all the added overhead it makes sense that our response time to our algorithm would worsen.

POSSIBLE EXTENSIONS TO THE PROJECT

1. So, our implementation had a drawback of detecting false pixels leading to false alarming. To counter this we implemented an iterative logic where we compared a previously processed frame with a real time processed frame and then segmenting out the static pixels. But, we didn't find that implementation to be full proof or effective

enough. There were still some bugs and loopholes which our algorithm needed to handle intelligently. What if there is an object coming under the fire color spectrum which is moving, even this case can lead to false alarming. As a solution, there should be a computation in our algorithm where it can detect the expanse or spread of the fire mass.

2. A major issue in our system implementation on the Raspberry Pi was laggy output because of the low processing power of it's gpu. Our thought on curbing this lag was that we can downgrade the image resolution inside our program code and then process it. Another idea was that we can perhaps sample at a different interval of time between frames.

MISSING MILESTONES

1. As mentioned in our project proposal we were planning to implement some part of our code in the kernel module and check whether it gives us better results than the user space implementation in terms of response time. Since, we were using OpenCV libraries in our program, we had a doubt over a few OpenCV instructions and data structures. Compatibility between the kernel implementation and our algorithm seemed out of portion since the instructions and data structures have been pre-defined under the OpenCV header files. Moreover, we had less time to think about its implementation since we were concentrating more on the optimization of our algorithm using the multitasking techniques.
2. When we implemented a base version of our fire detection algorithm we were facing a drawback of false pixels which can lead to false alarming. To counter this bug we wanted to implement some solid edge detection algorithm within our program. We had proposed to implement the Sobel Edge detection in our program to calculate the pixel gradient in every frame for our region of interest. The purpose of the Sobel was to take the dynamicity of fire into consideration. However, this would have been ideal if we were focusing on image processing techniques, but our goal was to learn and apply techniques within embedded operating systems. Because our focus was not in image processing we took a simpler technique that creates a frame by differencing the current fired detected processed frame with the previous fire detected fire frame. By applying this technique, we eliminate false fire pixels that have no dynamic characteristic of fire.

Talking about negative results, we can say that currently our algorithm is able to detect a fire mass accurately, but we still have concern over an issue. We considered a scenario, where there is a waving flag, and whose color is lying in the color spectrum of fire. This can lead to false alarming. How can we improve this? Well, our thought on this problem was to use a 'Sampling mechanism' that is to obtain a frame at every particular interval of time (it can be in milliseconds or seconds). Sampling will help us know about the severity of fire, thus making it's validation more deterministic. The trade off here is that some fire can spread quickly and some fire can spread at a slower rate. Perhaps, an average or optimal sampling interval time must be taken so that the system does not warn with delay.

INDIVIDUAL ROLES

Luis Camal - Camera interfacing, HSV modelling & calculation, implementing pixel difference iterative logic, fork() implementation in program.

Anup Kirtane - Raspberry Pi configuration, BGR modelling & calculations, BGR model algorithm implementation, Multi-threading implementation in program.

Collaborative work - Study about OpenCV tutorials and understanding it's image processing libraries, Colorspace conversion of image (BGR - HSV), Fire detection algorithm implementation using serialized processing, Testing algorithms in real time and time response calculations.

REFERENCES

1. Learning properties about BGR and HSV colorspaces:
   http://www.learnopencv.com/color-spaces-in-opencv-cpp-python/
2. Setting H-S-V values after converting image from B-G-R colorspace:
   https://stackoverflow.com/questions/35846581/opencv-c-set-hsv-values-of-a-pixel?rq=1
3. Remote access to Raspberry Pi session using VNC viewer:
   https://www.realvnc.com/en/connect/download/viewer/windows/
4. Installing OpenCV on raspberry pi:
   http://pklab.net/?id=392&lang=EN
5. Study about Multithreading implementation in C++
   https://www.tutorialcup.com/cplusplus/multithreading.htm
6. Paper on "An Intelligent Real-Time Fire-Detection Method Based on Video Processing":
   http://ieeexplore.ieee.org/document/1297544/
7. Paper on "Fire Detection Algorithm Using Image Processing Techniques" by Kumarguru Poobalan and Siau-Chuin Liew:
   fire detection algorithm using image processing techniques
8. Study on mmap implementation:
   http://zedboard.com/content/mmap-access-memory