Winter Workshop I.P. Phase - 1

I.P. Workshop Documentation

Eat ! Sleep ! Code ! Repeat !

Introduction

This is a documentation of the recently conducted Winter Workshop on Image Processing, and a compilation of all the codes we learned to use during the same . It was great fun working with C++ on a new platform (for me !) i.e. Visual Studio.

Kudos to my mentors for conducting such a great workshop ! A BIG THANK YOU to Kuntal,Aditya,Vaibhav and Vaisal .

And of course it'd never have been as fun as it was without my batchies !

Day-wise Documentation

Day 1

We **installed the Visual Studio software** .

After linking out Projects with the correct folders we ran a sample Program to test the Software.

Learnt the basics, like what an image is (a **collection of pixels** ?!) , how we use the **3 channel system of colours** (Red,Green and Blue) and **single channel** (grayscale) system .

We initialize them as follows -

CV_8UC3 for 3 channel

CV_8UC1 for single channel

Also learnt **Initialisation of an image** -

Mat (image type) img (image name) (img.rows,img.cols,channel,255)

Intensity of colour varies from 0 to 255 for the 3 colours in 3 channel system .

In the single channel system 0 indicates black and 255 stands for white.

To **Traverse the image** pixel by pixel we use the following code -

```
for (int i = 0; i < img.rows; i++)

{

for (int j = 0; j < img.cols; j++)

{
```

```
img.at<Vec3b>(i,j)[0];       - For blue

img.at<Vec3b>(i,j)[1];       -For green

img.at<Vec3b>(i,j)[2];       -For red

img.at<uchar>(i,j);          -For grayscale

}

}
```

Day 2

Learnt how to **accept an existing image** on Visual Studio .

Mat img=imread(“filename”,1);

Learnt how to **convert a coloured image to Grayscale image** -

Grayscale by average method :

```
for (int i = 0; i < img.rows; i++)
{
for (int j = 0; j < img.cols; j++)
{
r=img.at<Vec3b>(i,j)[0];
g=img.at<Vec3b>(i,j)[1];
b=img.at<Vec3b>(i,j)[2];
imGray.at<uchar>(i,j)=(r+g+b)/3;

}
}
```

Grayscale by Human eye ratio method :

```
for (int i = 0; i < img.rows; i++)

{

for (int j = 0; j < img.cols; j++)

{

r=img.at<Vec3b>(i,j)[0];

g=img.at<Vec3b>(i,j)[1];

b=img.at<Vec3b>(i,j)[2];

imGray.at<uchar>(i,j)=(r*0.299 + g*587 + b*0.299);


}

}
```

Images can also be represented in a "**Binary**" form i.e., only black and white .

We can convert a grayscale image to a binary image by defining a limiting value called threshold .

For the given conversion we give the pixels lying above the threshold value value of white colour ,i.e. 255 and those below the threshold are given value of black equal to 0.

The threshold value can be defined as a fixed value by the Programmer or we can create a trackbar to change the threshold value manually .

To create a trackbar -

```
int t = 0;
namedWindow("Window", CV_WINDOW_AUTOSIZE);
imshow("Window", img1);
createTrackbar("threshold", "Window", &t, 255);
```

Where t is the value of threshold which can be toggled by user between 0 to 255.

Then, if t is the value of threshold -

```
for (int i = 0; i < img.rows; i++)
{
for (int j = 0; j < img.cols; j++)
{
if((int)img.at<uchar>(i,j)>t)
{
img.at<uchar>(i,j)=255;
}
else
{
img.at<uchar>(i,j)=0;
```

}

}

}

Examples -

Grayscale image and corresponding Binary image .

Binary image using Histogram -

If we compute the number of pixels of same intensity in an image and store it in an array having parameters 0 to 255 ,we can keep the value of threshold to be the value of intensity when number of pixels just crosses half the total number of pixels in the image .

***Extras** -

Also to escape the output window we use - waitKey(0)

To print an image -

Declaring window : namedWindow(“name of window”,CV_WINDOW_AUTOSIZE);

Showing window : imshow(“name of window”,image name);

We also used the **CallBack function** to determine details of the pixel at a given point of an image . By giving an event for clicking the left button we print the asked details -

```
if (event == EVENT_LBUTTONDOWN)
{
Data;
}
```

The function itself -

```
setMouseCallback("Window", CallBackFunc, NULL);
        cvWaitKey(0);
```

Day 3

Started by doing a program to **Blur the image -**

In this program we select a kernel of pixels around the pixel being considered . After considering the **validity of the pixels** using this isValid function :

```
int isValid(int i, int j, Mat img)

{

if ((i<0) || (i >= img.rows) || (j<0) || (j >= img.cols))

 return 0;

return 1;

}
```

We traverse the image and for every square kernel we take the average of intensities of all constituent pixels and store it in all the pixels of the kernel . We can also take the median of intensities of all the pixels in the kernel and do the same instead .

For mean :

```
for (int i = 0; i < img.rows; i++)

{

        for (int j = 0; j < img.cols; j++)

        {


                for (int k = (i - ((size - 1) / 2)); k <= (i + ((size - 1) / 2)); k++)

                {

                        for (int l = (j - ((size - 1) / 2)); l <= (j + ((size - 1) / 2)); l++)
```

```
                {
                        if (isValid(k, l))
                        {
                                r = r + img.at<Vec3b>(k, l)[2];
                                g = g + img.at<Vec3b>(k, l)[1];
                                b = b + img.at<Vec3b>(k, l)[0];
                                cm++;
                        }
                }
        }
        temp.at<Vec3b>(i, j)[2] = (r / cm);
        temp.at<Vec3b>(i, j)[1] = (g / cm);
        temp.at<Vec3b>(i, j)[0] = (b / cm);
        r = 0; g = 0; b = 0; cm = 0;
    }

}
```

Where we store the sum of respective coloured entities in r,g and b and take the sum total of valid kernels for average in cm. The i and j loops traverse the image and k anl loop s traverse the kernel of pixel under consideration.

For median :

In median we have to create arrays to store the intensities of pixels and sort it out and choose the middle value .

```
for (int i = 0; i < img.rows; i++)
{
	for (int j = 0; j < img.cols; j++)
	{

		for (int k = (i - ((size - 1) / 2)); k <= (i + ((size - 1) / 2)); k++)
		{
			for (int l = (j - ((size - 1) / 2)); l <= (j + ((size - 1) / 2)); l++)
			{
				if (isValid(k, l))
				{
					r2[c] = img.at<Vec3b>(k, l)[2];
					g2[c] = img.at<Vec3b>(k, l)[1];
					b2[c] = img.at<Vec3b>(k, l)[0];
					c++;
				}
			}
```

```
                }


                temp2.at<Vec3b>(i, j)[2] = median(r2, c);

                temp2.at<Vec3b>(i, j)[1] = median(g2, c);

                temp2.at<Vec3b>(i, j)[0] = median(b2, c);

                for (int i = 0; i < kernel; i++)

                {

                        r2[i] = 0; g2[i] = 0; b2[i] = 0;

                }

                c = 0;

        }

}
```

median is the function which returns the middle value or the median value .

Gaussian Filter -

In this,we make the kernel around the pixel such that

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

Where , the central element is the pixel under consideration and the kernel surrounds it.

Thus by using the earlier algorithm to access kernel we multiply with the given factors to put a Gaussian filter on the images .

Original Image and the one with Gaussian filter

Sobel filter -

Similar to Gaussian filter except that instead of a 3 channel system we apply this filter to a grayscale image .

The kernel matrix in question is :

Gx,

-2	-1	-2

0	0	0
2	1	2

Gy,

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

Thus, the filter is applied along lines parallel to x and y axis respectively .

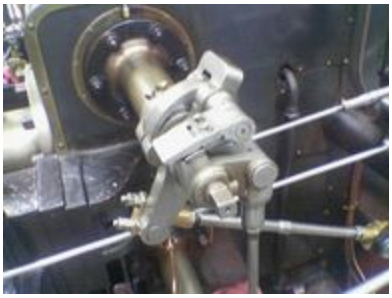
We also have a filter, which utilises both Gx and Gy .

It uses the kernel values of both as :

$$Gxy = (Gx^2 + Gy^2)^{1/2}$$

Thus , image resolved along a slant line.

Examples -



<-Original

Sobel filter image ->

Edge detection -

A program was done to detect edges of objects in an image using similar thought process.

First we convert the image to a binary image . In this, we take kernel into account and after comparing the threshold value to the difference between the maximum and minimum intensity values we print the according values to form only edges of objects.

We also used **Canny Function** for edge detection -

We use it as : Canny(input image , image output, limit min,limit max, kernel dimensions);

IT is a simple and effective way of giving edges to an image by changing pixels taken into account utilising different threshold values and trackbars.

***Extras** -

To save an image : imwrite("filename",image);

To clone an image

Erosion and Dilation -

These are basic noise reduction tools which help in removing spots and unnecessary lines from image . These functions can be carried on Binary images .

While in erosion : Image with black background and white image is sharpened
Image with white background and black image is blurred

In dilation : Image with black background and white image is blurred
Image with white background and black image is sharpened

Basically in erosion even if one black pixel exists in kernel the central pixel is converted to black . Vice versa for dilation.

In case of a binary image with more than 50% black we perform erosion and then dilation for best results and in case of predominantly white image we first apply dilation and then erosion for noise reduction .

Code for erosion -

```
for (int i = 0; i < filename.rows; i++)

{
```

```
for (int j = 0; j < filename.cols; j++)
{
        b = 0;
        for (int k = i - (size - 1) / 2; k <= i + (size - 1) / 2; k++)
        {
                for (int l = j - (size - 1) / 2; l <= j + (size - 1) / 2; l++)
                {
                        if (isValid(k, l, imtemp))
                        {
                                if ((int)imtemp.at<uchar>(i, j) == 0) b++;
                        }
                }
        }
        if (b > 0)
        {
                for (int k = i - (size - 1) / 2; k <= i + (size - 1) / 2; k++)
                {
                        for (int l = j - (size - 1) / 2; l <= j + (size - 1) / 2; l++)
                        {
                                if (isValid(k, l, filename))
                                {
```

```
                    filename.at<uchar>(k, l) = 0;

                }

            }}}}}
```

Day 4

Shushman's Algorithm -

We use this algorithm to detect objects in an image . Firstly we learnt what a blob is, a closed collection of pixels having the same intensity .

We basically traversed the image in binary form and take a kernel into account consisting of the top left corner cells of the kernel .

□	□
□	M

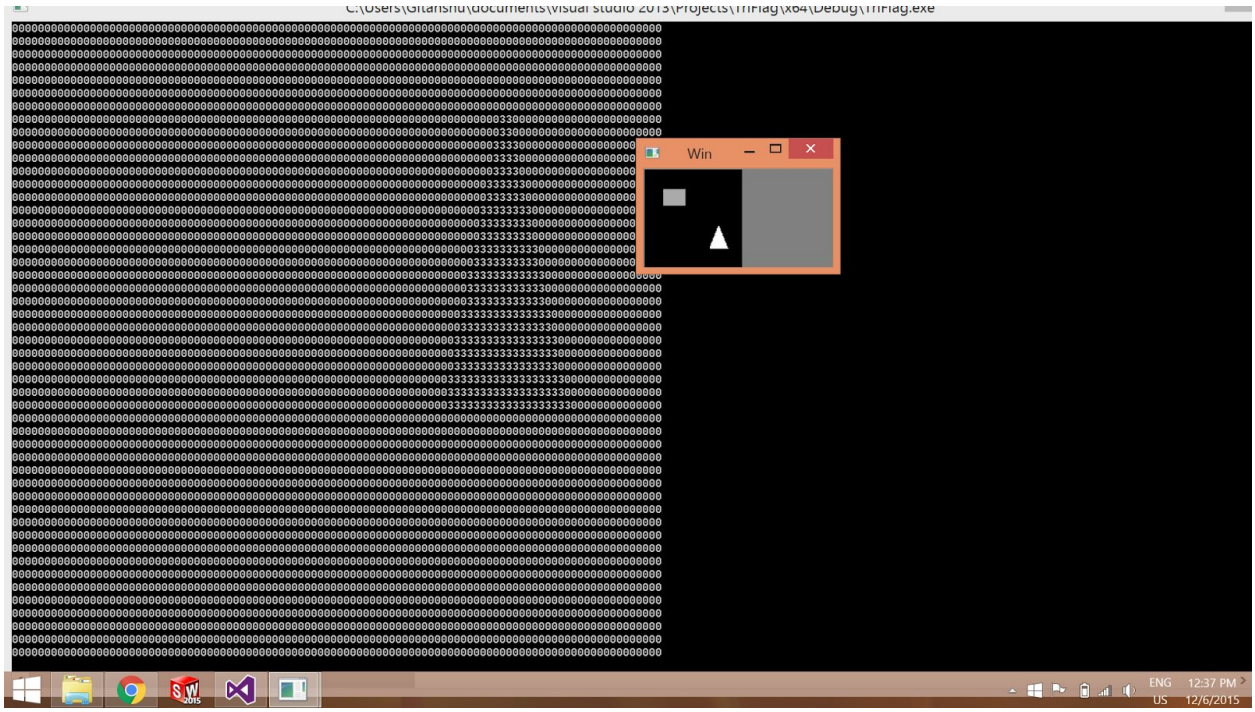
We declare a helper array and an array containing the nodes . Initially all the elements of the helper array have a large value infinity .

After running the kernel program as seen earlier we will mark the pixels in the array. For all white pixels in image we will mark them in numerically increasing order in helper array starting from 1, however, it will compare the closest three elements as seen in the kernel and assign it the lowest value .

For node array we basically have a truth table for our helper array . If two markings are connected their corresponding value in 2D node array is 1 . For each unconnected cell the value increases . These interconnected markings are being called by a recursive function.

We can make changes to helper array with help of node array and return values within an object to base value . Then we print the helper array .

Example -



Blob detection can also be done using **Stacks and Queues** using **BFS(Breadth First Search) and DFS(Depth First Search)** method .

BFS method uses a queue to find the pixels corresponding to the central pixel in a blob. In DFS method we use a recursive function . We compare the binary image to be checked with an all black picture containing the same number of pixels and mark every pixel not visited by comparing it to the other image . Initially all are marked as unvisited . As the pixel is visited it is made a white pixel .

```
void visit(int i, int j, int cnt,Mat img,Mat img2)
{

 img2.at<uchar>(i, j) = (255 / cnt);


for (int k = i - 1; k <= i + 1; k++)
{
for (int l = j - 1; l <= j + 1; l++)
{
if (img2.at<uchar>(k, l) == 0 && img.at<uchar>(k, l) == 255)
{
visit(k, l, cnt, img, img2);
}
}
}
}
```

cnt is count till we reach a point where we don’t find no unmarked pixel .
img in input image and img2 is output image .

Day 5

Learnt how to **read a video from our webcam** using -
videoCapture example(0);
Mat temp;
example.read(temp);

Fooled around with video and applied filters et. all on the video similar to what we did with images .

Then we moved on to **how read a pre existing video in Visual Studio** using -
videoCapture vid("filename");

We can total number of **frames** and current frames using -
vid.get(CV_CAP_PROP_POS_FRAMES); - For total frames
vid.get(CV_CAP_PROP_FRAME_COUNT); -For current frames

To access each frame in a video we can do the same in a loop -
Mat frame;
vid >> frame

Line detection -
For detecting lines in an image first we apply Canny function on it . Then we plot a graph between r and Θ using the following relation -

$$r = xcos(t) + ysin(t)$$

To plot graph -

```
Mat graph(200, 360, CV_8UC1);
	for (int i = 0; i < graph.rows; i++)
	{
		for (int j = 0; j < graph.cols; j++)
		{
			graph.at<uchar>(i, j) = 0;
		}
```

```
}

Mat line(img.rows, img.cols, CV_8UC1);



for (int i = 0; i < img2.rows; i++)
{
	for (int j = 0; j < img2.cols; j++)
	{
		if ((int)img2.at<uchar>(i, j) == 255)
		{
			for (int t = 0; t < graph.cols; t++)
			{
				r = (int)((i*cos(t*pi / 180)) + (int)(j*sin(t*pi / 180)));
				if (r>0)
				{
					if (graph.at<uchar>(r, t) < 256)
					{
						graph.at<uchar>(r, t) += 50;
					}
				}
			}
		}

	}
}
```

Having plotted the graph we take into consideration points where the intensity is more than a certain threshold and trace them back as pixels of image .On an entirely black image having same number of pixels as original image, we make these pixels white and print the image .

Code is as follows -

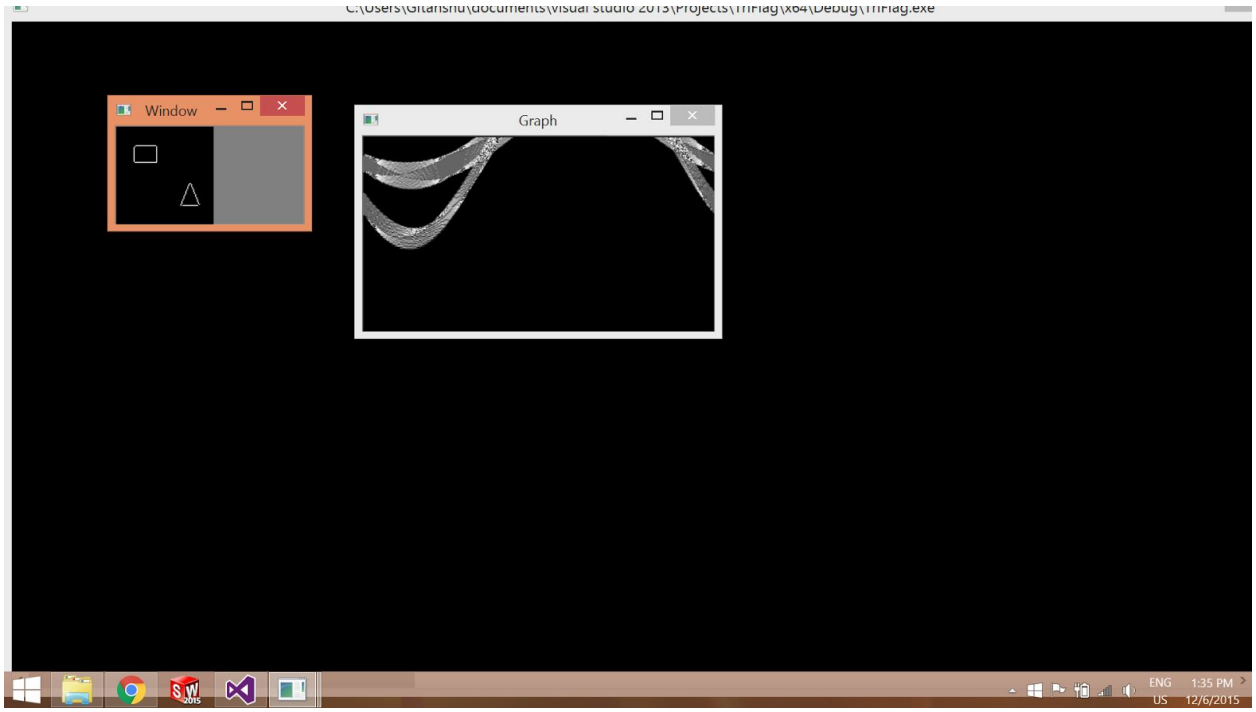
```
for (int i = 0; i < img2.rows; i++)
	{
		for (int j = 0; j < img2.cols; j++)
		{
			line.at<uchar>(i, j) = 0;
		}
```

```
        }
```

To make a black image .

```
                        for (int r = 0; r < graph.cols ;r++)
{
 for (int t = 0; t<2 * pi;t++)
        {

        if (graph.at<uchar>(t, r)>30)
        {
        for (int i = 0; i < img2.rows; i++)
                {
                for (int j = 0; j < img2.cols; j++)
                        {
                        if (r == i*cos(t*pi / 180) + j*sin(t*pi / 180) && img2.at<char>(i, j) ==
255)
                        {

                        line.at<uchar>(i, j) = 255;

                        }
                }

                }
                }
                }
}
```

Checking graph and back conversion .



Contours -

Also learnt detection of number of corners in an image with objects .Firstly we apply Canny to the read greyscale image . And define contour (an outline) -

vector <vector<Point>> contour>

This stores the number of lines and data of each pixel in it . Then we define hierarchy -

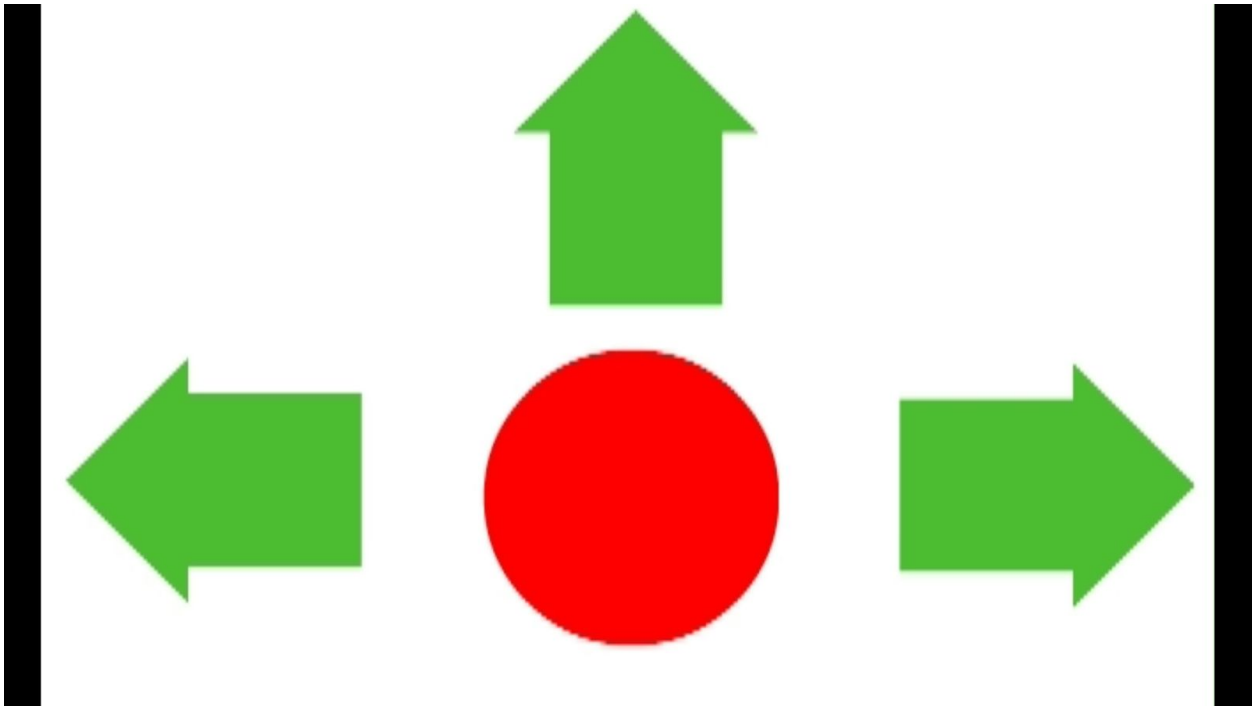
Vector<Vec4i> hierarchy

This stores information about the relation between each line and its surrounding line . Then we use contour function -

findCountour(img,contour,hierarchy,CV_RETR_TREE,CV_CHAIN_APRROX_SIMPLE,Point(0,0))
For(i → 0 →contour.size())
Define vector<Point> d;
Then to find corners we use -
approxPolyDP(contour[i],d,size of kernel,1 for closed objects or 0 for both open and closed objects)

And we use Print d.size() to get the no. of corners .

Day 6



Finally started AND finished work on our Problem Statement !

Firstly we did programs to find **longest lines** and to print **curves of objects** in an image .

They were basically derivatives of the line finding code done earlier and fairly elementary in nature .

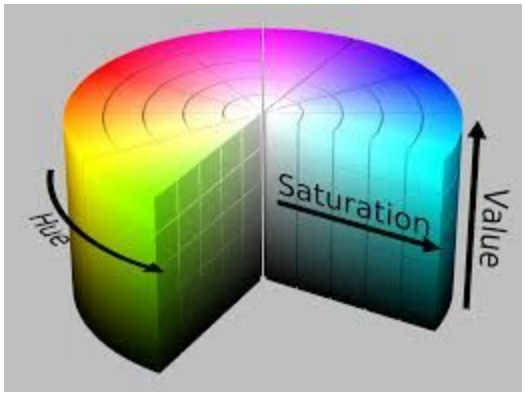
HSV (Hue,Saturation & Value) -

It was an important concept as we learned about another system of colours . This system defines a colour based on these three parameters . Considering a cylinder , then the depth defines value, the radius defines saturation and the angle of arc defines hue.

It is easier to use HSV rather than RGB system as it is processed faster . To convert RGB to HSV system we use -

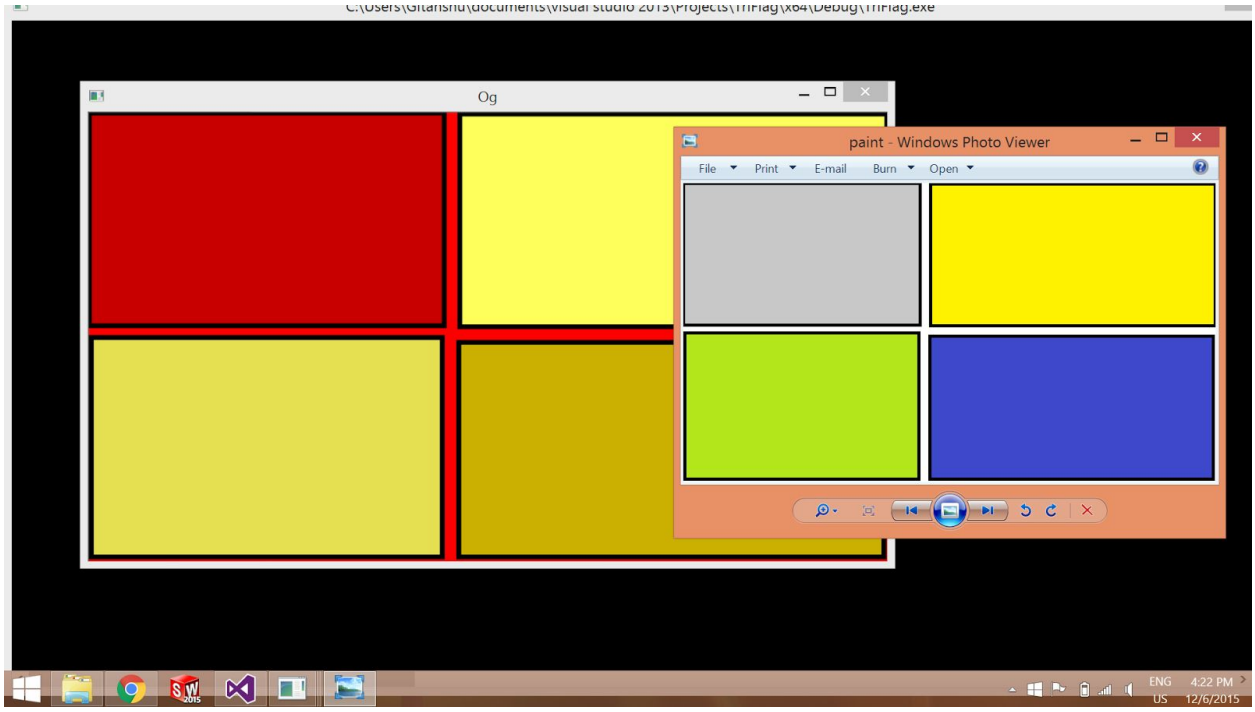
```
Mat hsvimg(img.rows, img.cols, CV_8UC3);
```

We have to define both minimum and maximum values for the threshold of H,S and V .



Diagrammatic Representation

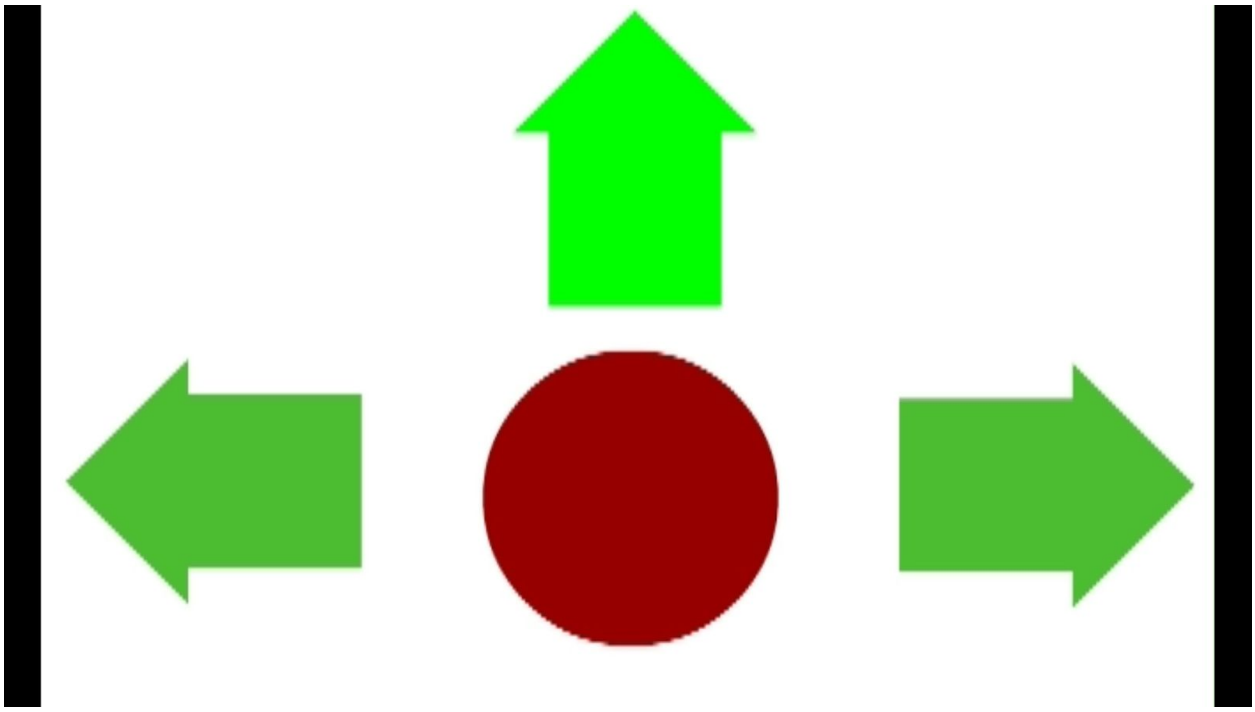
Image alongside HSV representation -



Problem Statements

PS I

To read a video which instructs the bot to move in 3 directions or stop based on a lighted arrow or circle .



Moving forward frame of video .

The code used is as follows .

Basically I stored the RGB values of dark red and bright red as well as bright green . Then comparing these values as I traversed each frame of the image I initialized the leftmost and rightmost j points of dark red and thus defined a j . This j is valuable in comparing in case of bright green .

If colour detected lies to the left of jleft it means the bot should turn left and if j lies right of jright then the bot will turn right . I also initialized an imin which is the topmost most or first i value of deep red in frame image . In case of bright green being detected and i being less than imin the Bot will move forward .

PS I solution -

```
#include "stdafx.h"
```

```
#include <stdio.h>

#include <iostream>

#include <opencv2\highgui\highgui.hpp>

#include <opencv2\imgproc\imgproc.hpp>

#include <opencv2\core\core.hpp>

#include<conio.h>


using namespace cv;

using namespace std;

/*

#define constant 10

Mat frame;


int main()

{

VideoCapture vid("PS1.avi");

cout << vid.get(CV_CAP_PROP_FRAME_COUNT) << endl;


namedWindow("Output", CV_WINDOW_AUTOSIZE);

for (; vid.get(CV_CAP_PROP_FRAME_COUNT) > vid.get(CV_CAP_PROP_POS_FRAMES);)

{
```

```
vid.read(frame);

int counterRed = 0;

int imin = frame.rows - 2, jleft = frame.cols - 2, jright = 0;

int rb0 = 0, rb1 = 0, rb2 = 254, rd2 = 152;

int gb0 = 0, gb1 = 255, gb2 = 0, gd0 = 51, gd1 = 189, gd2 = 77;


for (int i = 0; i < frame.rows; i++)

{

   for (int j = 0; j < frame.cols; j++)

    {

     if ((frame.at<Vec3b>(i, j)[0] == rb0) && (frame.at<Vec3b>(i, j)[1] == rb1) &&
(frame.at<Vec3b>(i, j)[2] == rb2))

         {

          counterRed++;

         if (counterRed <= 1)

            {

          printf("t"); //Stop

          cout << vid.get(CV_CAP_PROP_POS_FRAMES) << endl;

         }

         break;

        }
```

```
      }
    }
for (int i = 0; i < frame.rows; i++)
{
  for (int j = 0; j < frame.cols; j++)
   {
    if (((frame.at<Vec3b>(i, j)[0] == rb0) && (frame.at<Vec3b>(i, j)[1] == rb1)) &&
((frame.at<Vec3b>(i, j)[2] == rb2) || (frame.at<Vec3b>(i, j)[2] == rd2)))
     {
      if (imin > i)
                  imin = i;
      if (jleft > j)
         jleft = j;
      if (jright < j)
      jright = j;
      }
    }
}
int counterGreen = 0;
for (int i = 0; i < frame.rows; i++)
 {
```

```
  for (int j = 0; j < frame.cols; j++)
    {
     if ((frame.at<Vec3b>(i, j)[0] == gb0) && (frame.at<Vec3b>(i, j)[1] == gb1) &&
(frame.at<Vec3b>(i, j)[2] == gb2))
        {
         if ((i>imin) && (j>jright + constant))
           {
            counterGreen++;
             if (counterGreen <= 1)
              {
                printf("d"); //Right
                 cout << vid.get(CV_CAP_PROP_POS_FRAMES) << endl;
              }
             }
        if ((i>imin) && (j<jleft - constant))
            {
             counterGreen++;
            if (counterGreen <= 1)
                {
                  printf("a"); //Left
                   cout << vid.get(CV_CAP_PROP_POS_FRAMES) << endl;
```

```
                }
            }
         if (i<imin - constant)
        {
          counterGreen++;
        if (counterGreen <= 1)
           {
             printf("w");  //Proceed forward
             cout << vid.get(CV_CAP_PROP_POS_FRAMES) << endl;
              }
          }
        }
      }
  }
    if (vid.get(CV_CAP_PROP_POS_FRAMES) == (vid.get(CV_CAP_PROP_FRAME_COUNT) - 1))
           printf("Stop");


imshow("Output", frame);
```

```
  int iKey = waitKey(50);
  if (iKey == 27) break;

}
int iKey = waitKey(50);
waitKey(0);
return 0;
}

*/
```

PS Ⅱ -