Machine Vision and Behavioral Computing

AINT308

(S) DEAN HUNT

Table of Contents

AINT308

Abstract

Presented in this paper are various tasks that utilise stereo vision to perceive images and environments using simple Cross-correlation & Servocontrol, Homography and Itti & Koch's bottom-up Saccadic model. All tasks use Open CV which is an open source computer vision library. During the undertaking of these tasks the Plymouth OWL Robot was used for assessment 2i and the Owl-less Repo was used for assessments 2ii and 2iii.

Assessment 2i:

Introduction

To focus on an object, cameras in a stereo configuration need to align so that the object is in the centre of each 'eye' or lens. To track a target a technique known as cross correlation is used which is: *"A measure of similarity of two series as a function of the displacement of one relative to the other"* [1] The two images taken by each camera will be compared and using the Open CV function matchTemplate. The equation for tracking used is:

[2]

TM_CCOEFF_NORMED

$$R(x,y)=\frac{\sum_{x',y'}(T'(x',y')\cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'}T'(x',y')^2\cdot\sum_{x',y'}I'(x+x',y+y')^2}}$$

To achieve this the servos are moved and corrected so that each camera verges onto the chosen target, this is known as vengeance. From this you can extrapolate depth and position providing that the cameras and servos are calibrated correctly. One issue is that in order to do this some constants such as the interpupillary distance (the distance between the lenses of the cameras) will be unique for each OWL robot and will need adjusting to calculate depth effectively. A paper written in 1999 states:

> *"Using binocular systems to extract the depth of the objects in a scene demands to know the geometry of projection axes according to the nature of the scene. This assumption is imposed by the need of establishing the interest objects in the scene and the evaluation of the overlapping necessary for the stereo correspondence. The vergence of the focal axis constitutes the basic tool to obtain good results in the stereo correspondence algorithms because the disparity of the interest objects is reduced. In this way, objects located from different distances can be used. "* [3]

This make it very clear that in order to use vengeance and other techniques to view objects in a scene use of geometry will need to be used.

Solution

In order to track a target code in *figure 1* was used. This code is used to control the servos to keep the target in the centre of the x and y positions of the right camera. The same code is used for the left camera.

```
//match template within left eye
        OwlCorrel OWL_L;
        OWL_L = Owl_matchTemplate(Left, OWLtempl);

        //match template within right eye
        OwlCorrel OWL_R;
        OWL_R = Owl_matchTemplate(Right, OWLtempl);

//Control the right eye to track the target
        //Update x-axis value based on target pos
//Calculate number of pwm steps per pixel
        double RxScaleV = RxRangeV/static_cast<double>(640);
//Compare to centre of image
        double rXoff= (OWL_R.Match.x + OWLtempl.cols/2 -320)/RxScaleV ;
 //Update Servo position
        Rx=static_cast<int>(Rx+rXoff*KPx);

        //Update y-axis value based on target pos
//Calculate number of pwm steps per pixel
        double RyScaleV = RyRangeV/static_cast<double>(480);
//Compare to centre of image
        double rYoff= ((OWL_R.Match.y + OWLtempl.rows/2 - 240)/RyScaleV) ;
//Update Servo position
    Ry=static_cast<int>(Ry-rYoff*KPy);
```

Figure 1 – Code for right eye tracking

Once the target is being successfully tracked, we can calculate the distance. To achieve this use of trigonometry to determine distance will be employed. Figure 2 shows various distances and angles used to calculate the distance from the target to the centre of the OWL.

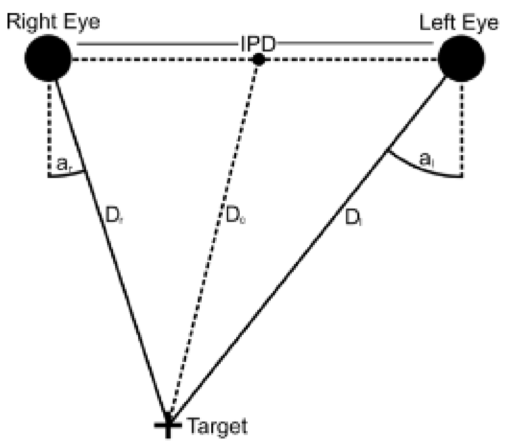


Figure 2 – Distance from angles of eyes

The known variables are a_r a_l and the IPD. To get D_c two triangles are drawn and all angles from both triangles are calculated. This is achievable because all angles inside a triangle add up to 180 degrees and once this is known we can use the sine rule to get the distance. Below are the equations used:

$RightInternalAngle = 90+a_r$ [Eqn.1]

$LeftInternalAngle = 90-a_l$ [Eqn.2]

$TargetAngle = 180 - (90-a_r) - (90+a_l)$ therefor

$TargetAngle = a_l - a_r$ [Eqn.3]

Using the Sin Rule:

$$\frac{\sin(a_l + a_r)}{IPD} = \frac{\sin(90 - a_r)}{D_l} \qquad \frac{IPD\cos(a_r)}{\sin(a_l + a_r)} = D_l$$

Once we know the Distance of each camera (D_l) and the IPD and the angle between them ($90 - a_l$) we can use these values and the cosine rule to find the central triangle length (D_c)

$$D_c{}^2 = D_l{}^2 + \left(\frac{IPD}{2}\right)^2 - 2 \cdot D_l \cdot \frac{IPD}{2} \cdot cos(90 - a_l)$$

$$D_c = \sqrt{D_l{}^2 + \frac{IPD^2}{4} - D_l \cdot IPD \cdot sin(a_l)}$$

The code in figure 3 implements this theory by calculating each PWM step in degrees then applying the algorithm stated above.

```
#ifndef VERG_H
#define VERG_H

#include <math.h>

#define Pi 3.14159265359

#define PWMstepInDeg 0.113

// The inter-pupillary distance (IPD)
#define IPD  0.065 // SI units Meters.

//Convert degrees into rads
double degToRad(double degrees){
    double rad = 0.0;
    rad = (degrees/360)*2*Pi;
    return rad;
}
// Convert rads into degrees
double radToDeg (double radians){
    double deg = 0.0;
    deg = (radians/(2*Pi))*360;
    return deg;
}
double PWM_to_Degree(int PWM){
   double deg = PWM*PWMstepInDeg;
   return deg;
}
//Calculate distance to object
double distanceToObject(int angleLeftEye, int angleRightEye){
    double eyeLeftinRad = degToRad(PWM_to_Degree(angleLeftEye));
    double eyeRightinRad = degToRad(PWM_to_Degree(angleRightEye));
    double num = IPD*(sin(Pi-eyeRightinRad));
    double dem = sin(eyeLeftinRad+eyeRightinRad);
    double Dl = num/dem;
    double Dc = sqrt((Dl*Dl)+((IPD*IPD)/4)-(Dl*IPD*sin(eyeLeftinRad)));
    return Dc;
}


#endif // VERG_H
```

Figure 3 - Code to calculate distance

```
double dis = distanceToObject(abs(Lx-LxC),abs(Rx-RxC));
        printf("distance to target %f\n\r", dis);
```

Figure 4 – Function with arguments passed in

Results

To test if the distanceToObject function works correctly a ruler was placed and the distance of a target was measured from 10cm up to 1m. This was completed three times and an error was calculated which can be used as a correction term. Figure 5 shows the values and graph produced from this experiment. It is obvious that after about 80cm the OWL robot gives incorrect readings and can be discarded. This shows that 80cm is the machine limits.

Actual (Meters)	Measued1	Measured 2	Measured 3	Error 1	Error 2	Error 3	Average Error
0.1	0.162	0.143	0.164	0.062	0.043	0.064	0.056333333
0.2	0.281	0.264	0.29	0.081	0.064	0.09	0.078333333
0.3	0.3	0.324	0.318	0	0.024	0.018	0.014
0.4	0.39	0.417	0.4	-0.01	0.017	0	0.002333333
0.5	0.53	0.562	0.549	0.03	0.062	0.049	0.047
0.6	0.67	0.725	0.684	0.07	0.125	0.084	0.093
0.7	0.75	0.782	0.851	0.05	0.082	0.151	0.094333333
0.8	0.8	0.861	0.822	0	0.061	0.022	0.027666667
0.9	0.45	0.32	1.4	-0.45	-0.58	0.5	-0.176666667
1	0.12	2	2.1	-0.88	1	1.1	0.406666667

Figure 5 - Measurements of distance

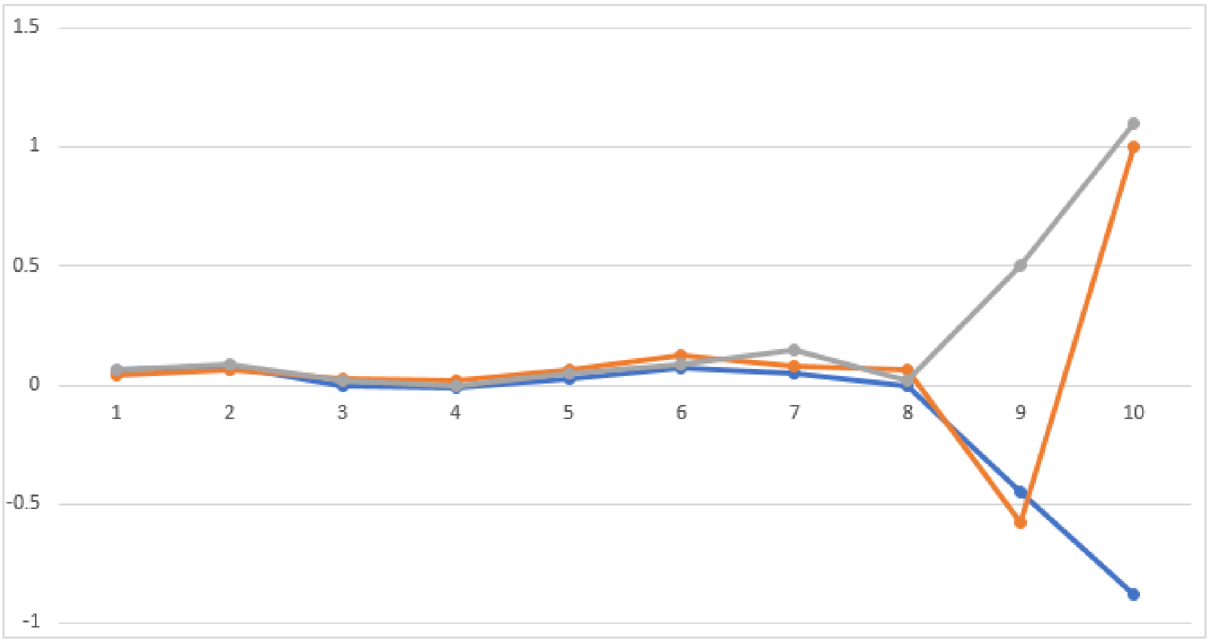


Figure 6 - Graph of calculated distance

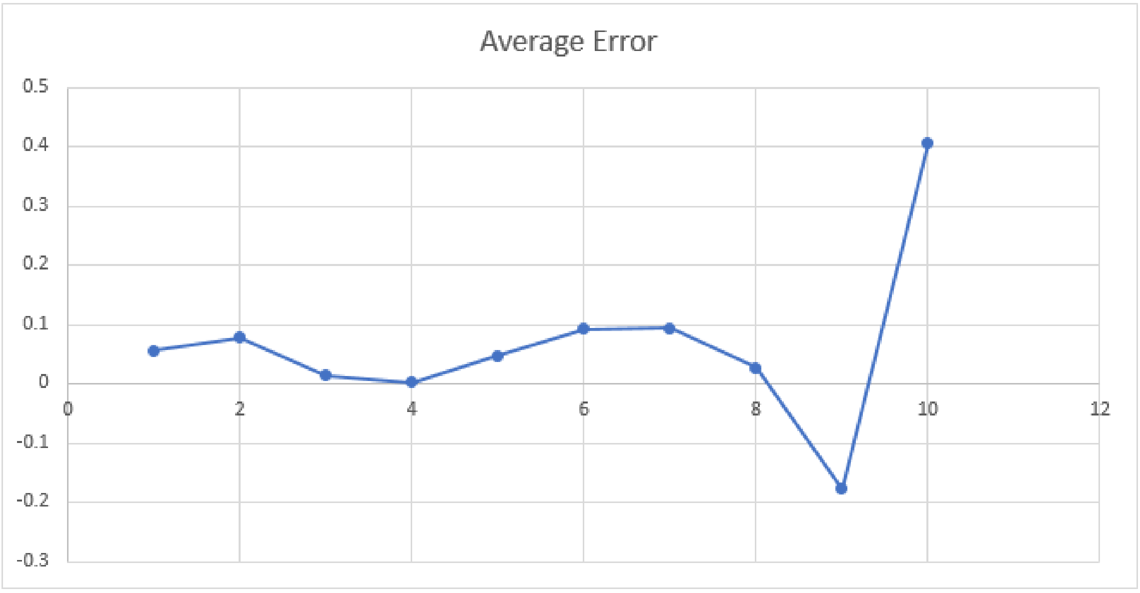


Figure 7- Graph of average errors

Here are the

Average Error
0.056333333
0.078333333
0.014
0.002333333
0.047
0.093
0.094333333
0.027666667
-0.176666667
0.406666667

The average of the first 8 errors for this OWL is **- 0.051625** . Using this a simple error adjustment shows closer measured values are to the real values.

Adjusted Measure 1	Adjusted Measure 2	Adjusted Measure 3
0.110375	0.212375	0.238375
0.229375	0.272375	0.266375
0.248375	0.365375	0.348375
0.338375	0.510375	0.497375
0.478375	0.673375	0.632375
0.618375	0.730375	0.799375
0.698375	0.809375	0.770375
0.748375	0.268375	1.348375
0.398375	1.948375	2.048375
0.068375	-0.051625	-0.051625

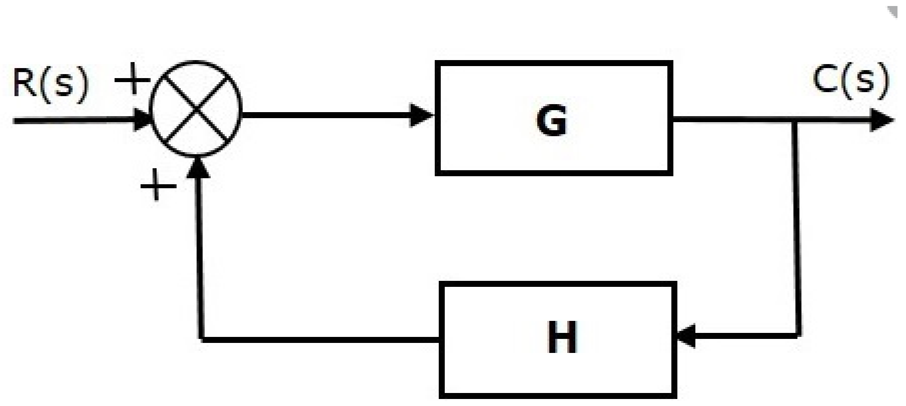


Figure 8 - Negative feedback loop

Figure 8 shows how you would error correct the measurements using a simple feedback loop, where H = - 0.051625 and G = the measured value.

Assessment 2ii:

Introduction

In order to use the camera's for stereo vision they must first be calibrated. Once a 3D camera model has been obtained it is then possible to move onto calculating values and information from the real world. Values such as depth are not possible to achieve without stereo cameras. One technique is to use a disparity map to find corresponding matches between pixels. To do this the image is converted to greyscale and the brightness of each pixel relates to difference in position at which the left and right eyes see a target. The closer a target, the higher this difference in perspectives, and the brighter the value. This is different from the depth map which is produced for the second part which the pixels become darker the closer the object and is linear.

In figure 8 we can see the parameters needed to calculate disparity.

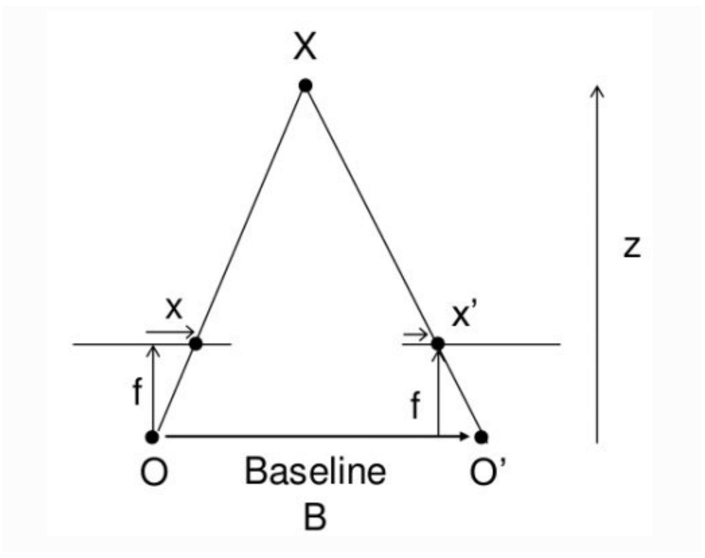


Figure 9 – Disparity calculations

Solution

A chessboard pattern is used to calibrate the pair of cameras to enable them to be used in stereo. Using OpenCV and the file stereo_calib.cpp we can pass a chessboard pattern in and a pair of jpegs of a stereo image. To do this a stereo pair of images where used to establish a common three-dimensional baseline, you can see in fig 10 that the green lines line up with parts of the chess board pattern despite the images having other differences due to the distance between cameras.

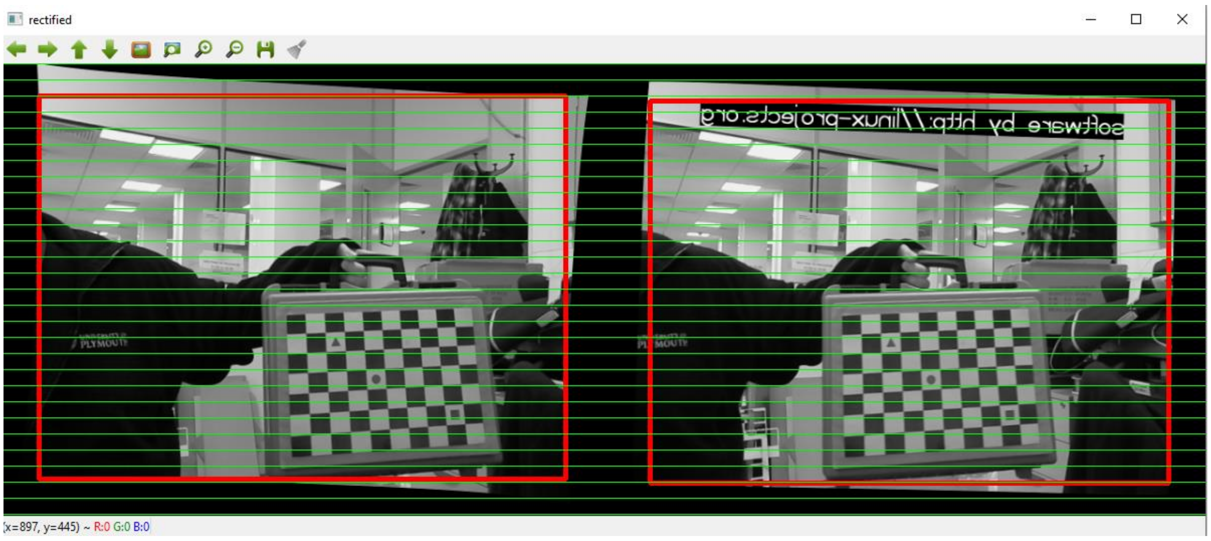
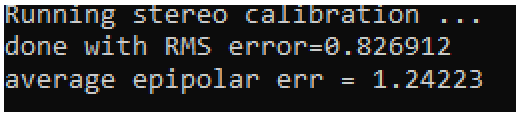

rectified
software by http://linux-projects.org
(x=897, y=445) ~ R:0 G:0 B:0

Figure 10 - rectified images



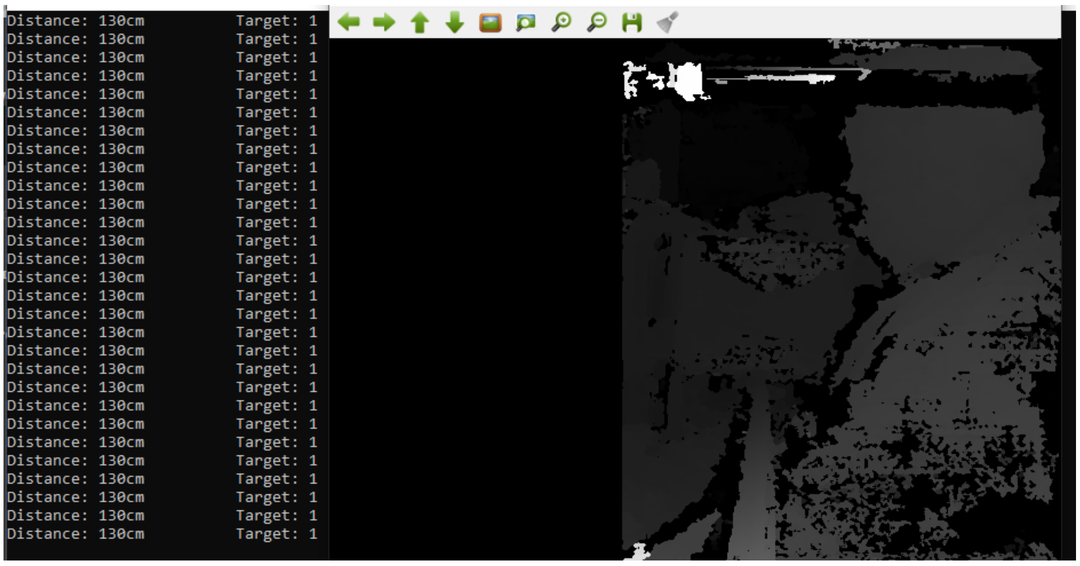
```
Running stereo calibration ...
done with RMS error=0.826912
average epipolar err = 1.24223
```

Once the calibration is complete we can create a disparity map. Using the stereo pair of cameras and this code:

```
    //Match left and right images to create disparity image
        numberOfDisparities = numberOfDisparities > 0 ? numberOfDisparities
: ((img_size.width/8) + 15) & -16;
        int cn = Left.channels();
        int sgbmWinSize = SADWindowSize > 0 ? SADWindowSize : 3;

        sgbm->setBlockSize(sgbmWinSize);
        sgbm->setPreFilterCap(63);
        sgbm->setP1(8*cn*sgbmWinSize*sgbmWinSize);
        sgbm->setP2(32*cn*sgbmWinSize*sgbmWinSize);
        sgbm->setMinDisparity(0);
        sgbm->setNumDisparities(numberOfDisparities);
        sgbm->setUniquenessRatio(10);
        sgbm->setSpeckleWindowSize(100);
        sgbm->setSpeckleRange(32);
        sgbm->setDisp12MaxDiff(1);
        sgbm->setMode(StereoSGBM::MODE_SGBM);
        sgbm->compute(Left, Right, disp);
```

The disparity map can be measured (fig 11) the disparity was taken from the L value from a pixel of the object and plotted over the measured distance of each image provided. As seen in Figure 9 the graph plotted has a Y = 1/X relationship. Using a simple equation of Distance = some constant divided by disparity. Rearranging to calculate for the constant we find it is: 62610.



```
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
Distance: 130cm          Target: 1
```

Figure 11- Disparity map

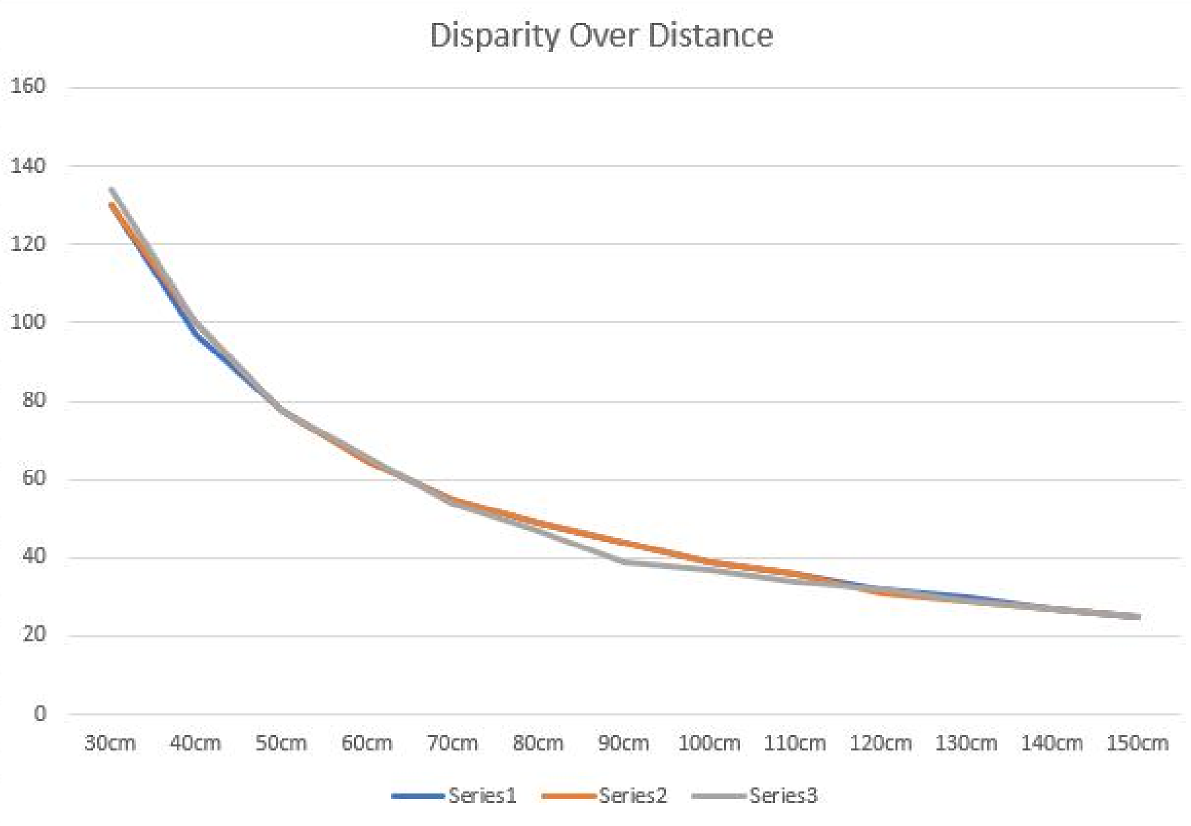


Figure 12 - graph showing disparity over distance

Now that the constant value has been characterised it can be used to calculate distance. This is done by utilising the code in *figure 13*. A new Matrix is created and two for loops iterate through every pixel in the y and x axis respectively. For every pixel the distance value is calculated in the line: uchar pix_Distance = (uchar)(62610/cal_Distance);

Then the value is saved in the distance Matrix.

```
//Convert disparity map to an 8-bit greyscale image so it can be displayed
        disp.convertTo(disp8, CV_8U, 255/(numberOfDisparities*16.));
        imshow("left", Left);
        imshow("right", Right);
        imshow("disparity", disp8);

        // make new M
        Mat Distance_M(disp.size(), CV_8U, Scalar(0));

        // get disp value from y then x
         for (int y = 0; y <= 479; y++){
            for (int x = 0; x <= 639; x++){
          // run through function const/disp to get distance.
              float cal_Distance = disp.at<ushort>(y,x);
              uchar pix_Distance = (uchar)(62610/cal_Distance);

          // populate distance at row y coloumn x
             // cast to 8 bit in Mat
                Distance_M.at<uchar>(y,x) = pix_Distance;
             }
         }

           imshow("Distance", Distance_M);
```

Figure 13- code to generate a depth map

Results

https://youtu.be/lBxpQQnVxR4

Assessment 2iii:

Introduction

Human vision has an area of the eye called the fovea, this region when focused allows us to perceive high detail. In order to do this rapid eye movement will scan a section we deem important this can give a more detailed view of an object much larger than the fovea. This is known as Saccadic eye movement. In this assessment the saliency function in Open CV is used to allow the scanning of a given image and weighting given to more important elements. In figure 12 you can see the structure of a Salience model which adds together several feature maps into a “saliency” map.

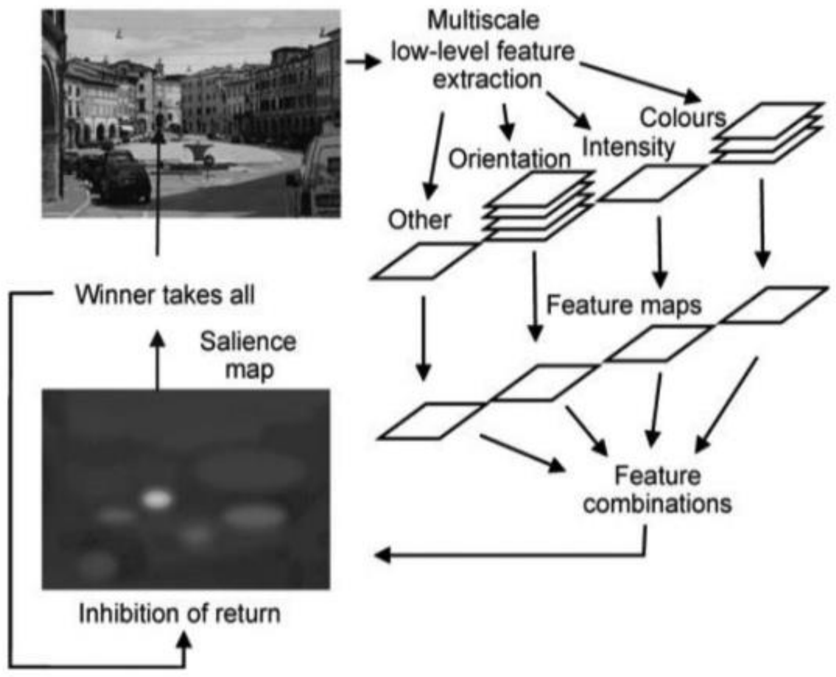


Figure 14 - Schematic of Itti's salience model. Figure from Land and Tatler (2009); redrawn from Ittiand Koch (2000)

Some maps that are of interest are ones that can denote one of the following:

- Strong colour
- High contrast
- Strongly oriented
- Moving Close-by
- Faces (emotive targets)

It is important that each map is only a single feature and then combined into the saliency map.

Solution

A Saccadic mode was provided so two extra maps where created to be added in along with the DoG low bandpass map and the Fovea Map. The first map choice was a canny edge detector.

Canny Edge Detection

A Canny edge uses an algorithm to detect edges the first step is to reduce noise below are the steps to achieve this:

Filter out any noise using the blur function

```
blur( LeftGrey, detected_edges, Size(3,3) ); // reduce noise
```

Below this how the blur function works:

“[blur] removes high frequency content (eg: noise, edges) from the image. So edges are blurred a little bit in this operation (there are also blurring techniques which don't blur the edges). OpenCV provides four main types of blurring techniques.” [4]

Fig 13 shows the defined size of 3x3.

$$K = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 15- 3x3 averaging filter kernel

Next we can apply the Canny function in open CV:

```
Canny( detected_edges, detected_edges, lowThreshold, lowThreshold*ratio,
kernel_size );
```

Here are the variables used in the function:

```
Mat dst, detected_edges;
int lowThreshold = 90;
const int max_lowThreshold = 100;
const int ratio = 3;
const int kernel_size = 3;
const char* window_name = "Edge Map";
```

This function that's the Smooth image and then filters it with a Sobel kernel in both horizontal and vertical direction to get the first derivative in horizontal Gx and vertical Gy. From these two the edge gradient and direction for each pixel can be calculated as shown in fig 14.

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Figure 16 - edge gradient and direction for each pixel

Once this is completed the function will remove any unwanted pixels which it believes not to be an edge by checking if the pixel value is the local maximum along gradient direction. Once this has been completed the final step Hysteresis can be applied. As seen in the variables there are two thresholds (upper and lower)

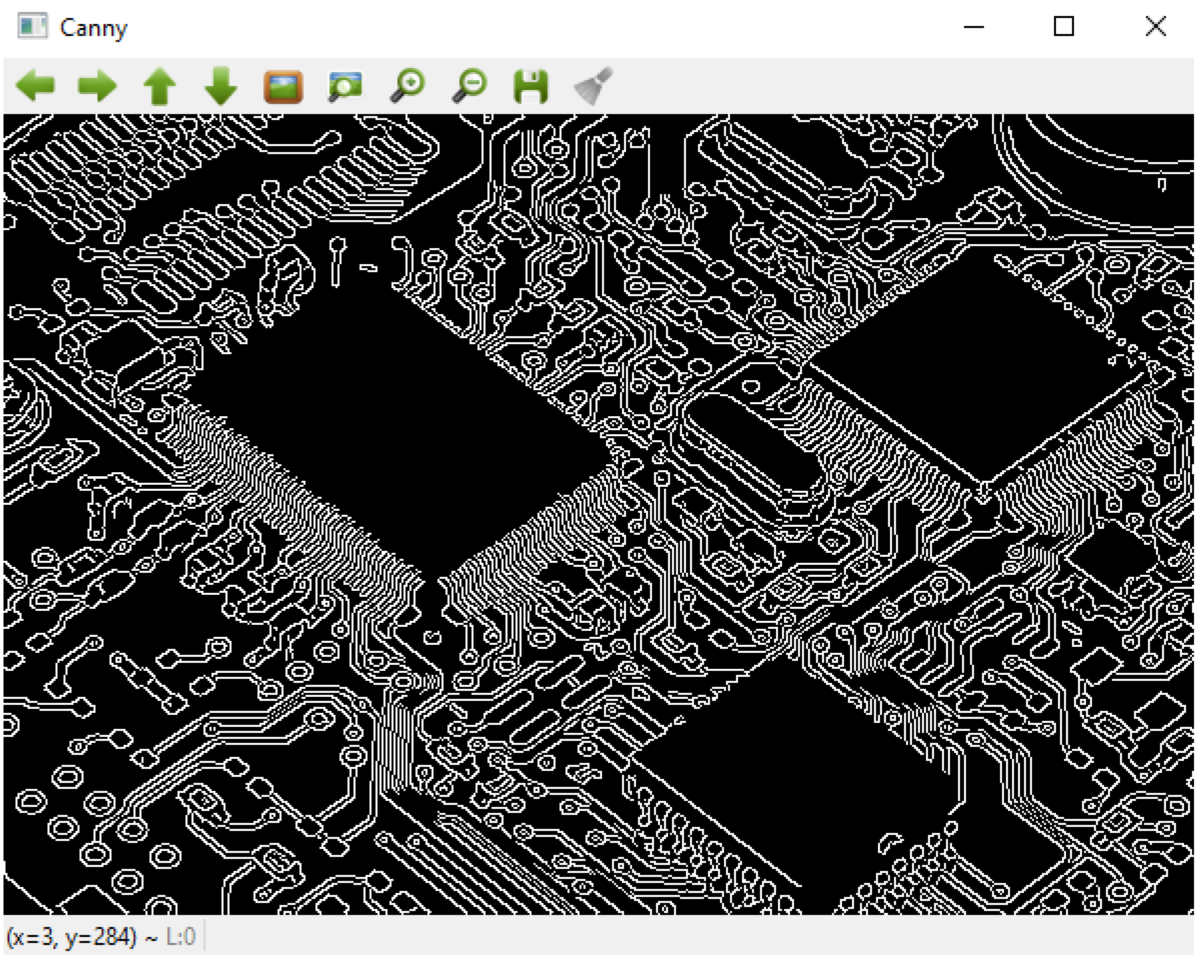


Figure 17 - Canny Edge detection

Colour Intensity

In order to detect strong colours we need to change the BGR (Blue/Green/Red) format of images in OpenCV to HSV (Hue/Saturation/Value) this will allow colours that stand out from the background to be identified. One issues with only using saturation to determine colour strength is that dark areas have a undefined saturation value as a saturated dark colour is hard to find. We can address this by using the value part of the image and multiplying this with the saturation. This can been seen in the code where split_hsv[0] is used to store split_hsv[1] (Saturation) and split_hsv[2] (as Hue is not important).

```
Mat hsv;  // Matrix for converting colour of Left Mat
Mat split_hsv[3]; // Split for Hue[0] Sat[1] and value[3]
cvtColor(Left, hsv, COLOR_BGR2HSV); //Converting image from BGR to HSV
color space.
split(hsv,split_hsv); // preform split into HSV channels
split_hsv[0] = split_hsv[1].mul(split_hsv[2]); // Multiply the Sat with
value to avoid black areas being undefined.
split_hsv[0].convertTo(split_hsv[0], CV_32FC1);
 imshow("HSV", split_hsv[2]);
```

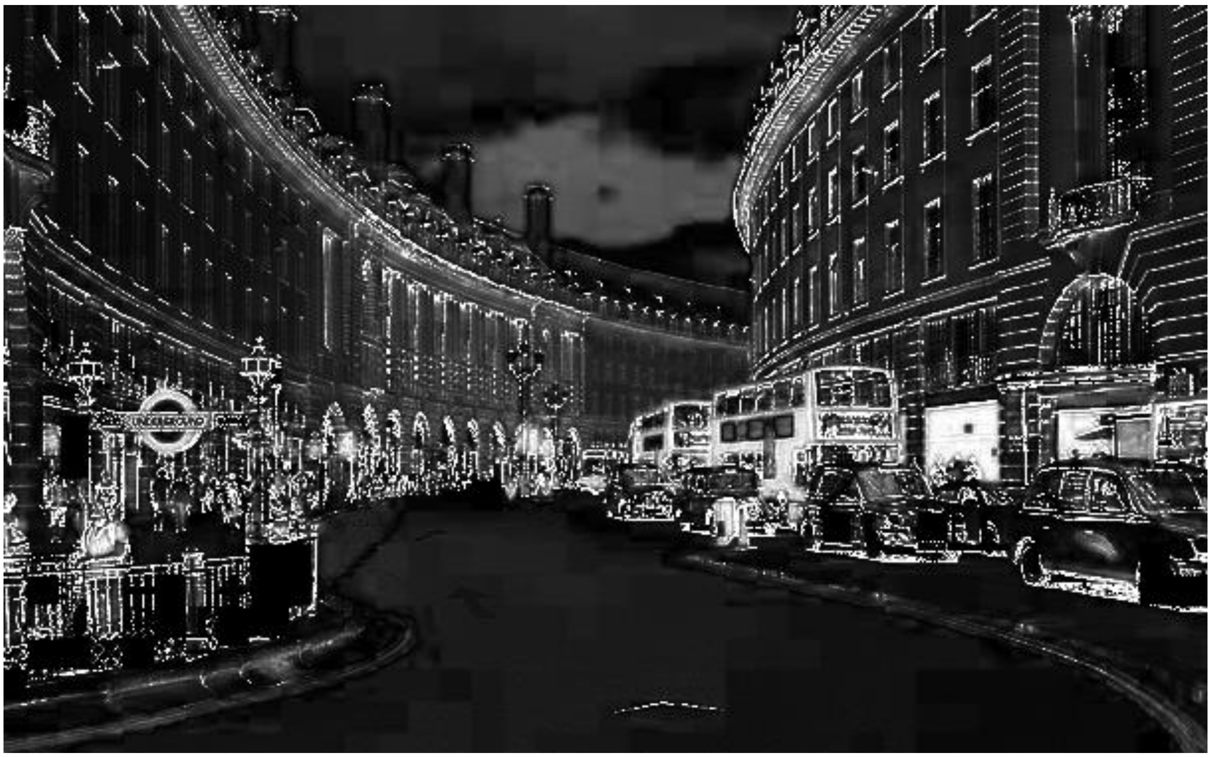


Figure 18 - Saturation map

Results

https://www.youtube.com/watch?v=aho6ibDe_Zs

Table of figures

References

[1] Wikipedia, “https://en.wikipedia.org/wiki/Cross-correlation,” [Online].

[2] “OpenCV.org,” [Online]. Available: https://docs.opencv.org/3.2.0/df/dfb/group__imgproc__object.html#ga586ebfb0a7fb604b35a23d85391329be.

[3] L. M. Jiménez, “Vergence Control System for Stereo Depth Recovery,” 1999.

[4] o. C. documentation, “https://docs.opencv.org,” [Online]. Available: https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html.