Abstract

This document contains a project report on the design, implementation and testing of a whiteboard capture system. The project is a large portion of my final year in a Bachelors of Mechatronic Engineering course. The goal of the project is to develop an affordable and accessible whiteboard capture system.
Research was performed and a viable design was created. This design was implemented and tested and the results are presented. The system design was established around a 1080p webcam. A program was created to utilize the webcam to captured and extract content from a whiteboard. The system captures high quality images of the content on a whiteboard. The camera placement and light conditions while using the system are variable making the system easy to use.

Table of Contents

Tables of Figures

1. Introduction

The goal of this project is to design and implement an affordable and accessible whiteboard capture system. To make a system that is both affordable and accessible it must avail of cheap tools that are accessible to the public. Due to the global pandemic and online learning has become the norm, many people have bought a webcam to use for online meetings. Taking advantage of the general ownership of a webcam, this is used as the main tool to capture the whiteboard. The most commonly used webcam nowadays is a webcam that has a 1080p resolution and costs somewhere between €40 and €100 euro [1]. Thus, for the purpose of this project, the 1080p Trust webcam costing €44 is used to implement this system.

There are three main aims of the project, these aims are centered around developing certain system characteristics that will ensure a robust system. These characteristics are as follows:

- The system must be effective in varying light conditions, these conditions include but are not limited to, natural light, artificial light and low light conditions.
- The position of the camera in relation to the whiteboard must be variable, meaning if the camera is not in an optimal position the system must still output a clear image of the content.
- The colour of the marker used on the whiteboard must be variable, the common colours used are black, red, green and blue. All colours should be a viable choice.

The scope of the design includes designing a system that is autonomous after a brief setup period. This setup period is to ensure the whiteboard is in frame and the user is content with the capture quality. After the brief setup, the capture system must output a timeline of images. Each image in the timeline must represent a content event. These events are declared when a piece of content is either added or removed from the whiteboard.

The design of the system can be broken down into multiple problems, each problem is solved as a task of its own. The main tasks of the project were as follows:

1. Capturing image with the webcam.
2. Isolating the whiteboard in the image.
3. Making the aspect ratio of the whiteboard in the image equivalent to actual aspect ratio.
4. Making the content have a high contrast with the whiteboard.
5. Recognizing when new content is added to the whiteboard.
6. Outputting a timeline of images showing the progression of the whiteboard over time.

At the start of the project a schedule was set in place, this schedule was used as reference for how much time should be spent on each task. Figure 1 shows the schedule created at the start of the project.

Task:	Start Date	End Date
Semester 1		
Background		
Research Current Products	09-Nov-20	15-Nov-20
Research Previous Studies	16-Nov-20	22-Nov-20
Write Literature Review	23-Nov-20	29-Nov-20
Semester 2		
Early Developement		
Write Status Report	18-Jan-21	5-Fed-21
Make Presentation	5-Fed-21	8-Fed-21
Design & Implentation		
Set up testing enviroment & capture first image	9-Fed-21	14-Feb-21
Isolate whiteboard in frame	15-Feb-21	21-Feb-21
Make the whiteboard asymetrical	22-Feb-21	28-Feb-21
Make content of the whiteboard pronouced	01-Mar-21	07-Mar-21
Recognise content events	08-Mar-21	14-Mar-21
Outputting timeline	15-Mar-21	21-Mar-21
Final Report		
Testing	22-Mar-21	28-Mar-21
Write final report	29-Mar-21	05-Mar-21

Figure 1: Project Schedule

In semester 1 the main background research was performed, this research included finding products that capture whiteboard content and studies that law out methods used to extract content from a whiteboard.

The start of second semester was mainly focused on creating a design and presenting tat design to supervisors. Once the final design was decided on it was split into multiple tasks, each task had a week to complete. The testing phase of the project required more time than allocated and this hindered alterations being made to the program.

When testing was performed and more desirable methods of performing tasks were found, the improvements could not be made to the program. The whiteboard capture program set out in this report is the first iteration and there are several alterations that would be made to this program to improve its efficiency.

2. Ethics

Engineers have a responsibility to be honest, impartial, fair and dedicated to the protection of the public's health, safety and welfare. Engineers must abide by the standards set by Engineers Ireland, to behave with integrity and remain responsible when dealing with their clients, employees, colleagues, and society in general. Since engineering deals with all walks of life care must be taken with every project undertaken. Safety is considered a one of the

highest of priorities, so prior to releasing anything the societal impact must be taken into account. For this project, research was conducted to find the effects this capture system may have on its clients and the people in its environment.

Universal Design for Learning sets out to provide equal opportunities for all individuals in a learning environment, such as student with disabilities. UDL aims to make learning accessible to all students no matter their ability, location or financial situation [1]. given that covid has restricted learning for all there has been an increased reliance on online learning and the use of learning platforms. Thus, this project has never been more timely. The system designed in this project may be used as an extra tool to provide educational content to students across the word who are in remote locations. Due to the dependency this system may have on it, the quality of design must be at the highest of standards as its failure can have a string of negative effects on educators and their students.

During a capture session, images of students may be captured accidentally if a student walks into frame. The capturing and storing of these images is illegal if the student has not given permission. If an image is captured it must be destroyed accordingly or if permission is given by the student the image must be kept in a secure place. If the security of the image is compromised, the storage of unprotected data breaks data protection law [2]. Due to the system not having a repository of images online and all the images captured being saved locally on the user's device, the user is responsible for the storage of the data and any breaches of the data protection act will be blamed on the user. If the capture system did have an online repository for the whiteboard images then the system owners have a responsibility to keep the content secure and dispose of content correctly.

3. Literature Survey

3.1 Current products

Kaptivo

Kaptivo is a camera module that is wall-mounted above the whiteboard, the camera has a polarised lens which suppresses glare by only letting light pass through in horizontal direction. The Kaptivo camera module takes about 5 minutes to detect the white board, which could be considered a long time. Then any content captured from the whiteboard is uploaded to a Kaptivo cloud through a Wi-Fi network where it can be viewed [3]. A big advantage that Kaptivo has is that the post processing of the captured image is very effective at making the image clear, mainly filtering to make a discoloured white board appear clean and content on the board black to give a high contrast image. A disadvantage to Kaptivo is that the arm is

heavy, making it difficult to move from room to room as educational professionals usually do. The product costs between €410-620 depending on how big the whiteboard is and if you want to connect to a Wi-Fi network which is expensive.

eBeam

eBeam is a smart marker that uses a sensor module placed on the wall adjacent to the white board and a marker sleeve to capture all strokes made by the marker within a 16ft x 5ft area. The sensor module creates a GPS map on the whiteboard and can track the movements of the marker, when the marker contacts the whiteboard the marker sleeve recognises this and starts to replicate the movements on a digital whiteboard [4]. The device has local memory and can capture up to 20,000 pages of content without any device connected to it. The eBeam has an eraser that comes in the kit that can remove content from a page with one easy swipe. The marker can easily be synced with a mobile device or computer where the content can be viewed [4]. The eBeam costs about €660 which is a high cost for any educational professional or business.

Two other solutions that do not solve the specific problem given but give similar results are the graphic tablet and using a basic camera or webcam to capture a video of the user writing on the whiteboard. The graphic tablet is an interactive pad that takes inputs from a digital pen and displays the pen stroke on a digital whiteboard. The graphic tablets are hard to adapt to as you are not looking at your hand when drawing, rather you are looking at the screen where the drawing is being displayed, this causes drawing to be harder and therefore be inaccurate [5]. To use these tablets the user would have to undergo training or practice to be able to reach the level of drawing with a normal whiteboard and marker approach.
Having a camera set up on a tripod to capture the whiteboard as a live video for viewers to see is another simple solution but this has two issues, the professor writing the notes will have to move out of the way intermittently to allow the viewer to see the full whiteboard and limits the amount of time the viewer must take the notes down. If the content is to be uploaded in pdf form, then the professor will have to go through the video and take screen captures when the whiteboard is clear of any disturbances. This is time consuming and the professor must put the screen captures together into a pdf in chronological order themselves.

3.2 Previous Research

There have been many articles written about whiteboard captures systems and there are many common aspects from these designs that are implemented in this iteration of the whiteboard capture system.

The first commonality in two of these papers [6] [7] is the identification of a person or object in frame that may be obstructing the view of the whiteboard. The premise of how they find the object in the image is to compare to two images captured within a second of each other. Since the person is the only object moving in these images, one image is subtracted from the other and there will be regions around the person that are not common to both images and will appear on the resultant image. The person is then isolated and the isolated part of the image is ignored in future sections of the program.

Isolating the whiteboard in the image is a vital step in taking the content from the board, many previous iterations of this system have the same method. An edge detector is applied to the image and outside edge of the whiteboard will be the most prominent edge [6] [8] due to either the shadow of the whiteboard or the colour differential between the whiteboard and the wall it is mounted on. The image can then be cropped down to a rectangle that encompasses this prominent edge, this simple method is most effective when the camera is placed directly in front of the whiteboard.

The cropped image of the whiteboard may not be at the perfect angle which causes the whiteboard to appear in the shape of a quadrilateral rather than a rectangle. A quadrilateral is a four-sided shape where each inside angle is different. One paper discusses an image rectification technique which transforms the whiteboard from the quadrilateral to rectangle [8]. Firstly, the four corners of the whiteboard are found, a perspective transform is created and the image is rectified to display the whiteboard as a perfect rectangle. The method used is based on a finding where two edges of the whiteboard meet, this can be difficult to implement to some inside angle being large and the two meeting edges are perceived as a curve rather than a corner.

The goal of this project is capturing the content on the whiteboard making the content clear but this can be difficult. An article from Microsoft Corporation [8] suggest using histogram equalization which takes the colours in the image and stretch them out over the full colour spectrum, this idea is only successful when the content on the whiteboard is one shade of one colour. If there are different shades of any single colour then some content may appear faded.

The shade of a colour can depend on the lighting present so ideal lighting conditions have to be used.

Other Articles suggest changing the colour image of the whiteboard to a binary (black and white) image to have the highest possible contrast [7] [6]. To capture the content effectively the frame captured is split into its red, green, and blue colour channels, each colour channel is changed to binary and then combined. This method is very effective as capturing content because certain content will be clearer on one colour channel over another colour channel, once combined each channel provides its best content.

Event detection is another goal of this project, when a new piece of content is added the program must recognise this and capture said content. The overwhelming majority use a grid method which splits the whiteboard image into cells and each cell is classified with a location and colour. When a new image is captured the cell is compared to its older counterpart of the same location, if the colour of the cell has changed then the content in that cell can be declared different. If enough cells are identified as changed then new content must have been added/removed to/from the board [9].

4. Background

This chapter contains information that will help understand the code and the methods used in the program. The background will be split into two sections, the first will contain the theory behind many of the methods used in images processing and the second will explain the basics of Python programming and the syntax of the programming language.

4.1 Theory

Matrices

Any digital image on the internet or taken on a device is represented by a matrix, each different type of image has its own method of being represented in a matrix. There are three different types of image, colour, greyscale and binary.

In matrix form, each element of a matrix corresponds to a pixel in the image, the element determines colour or grayscale intensity of the pixel. If a digital image as captured in 1280×720 pixels then the matrix will have a height of 720 pixels and width of 1280 pixels. Each element has a location given by a row column similar to that of an excel sheet or the coordinate system in fig [1].

For colour images, each colour in a digital image is represented by RGB, which is the three colour channels red, green and blue. Each colour channel can have a value from 0-255. An example of a colour is (128,0,128) or purple. To show a colour image in matrix format, each colour channel has its own matrix. To change a colour image to a greyscale image, the mean of each corresponding pixels in the channel matrices is taken and this then represents a single pixel in the greyscale matrix with a value between 0-255.

Greyscale is represented as a value from 0-255, where 0 is black and 255 is white. Binary is simply a 0 or 1, where 0 is black and 1 is white.

Co-ordinate System in Images

Every pixel in an image has a location, this is shown by an x and y coordinate. The x coordinate represents the column and y coordinate represents the row, where the column and row meet is the location of the pixel in the coordinate system/matrix. Unlike most coordinate systems the origin of the image is the top left as shown below in fig. 2 where M is the height of the image and N is the width of the image.

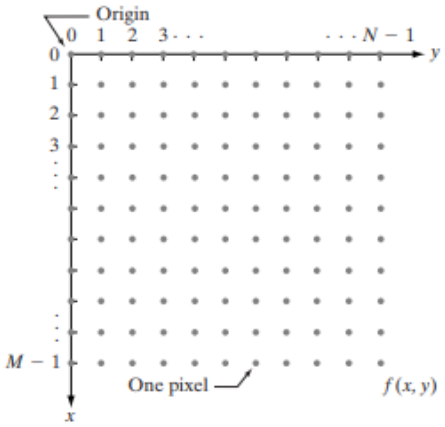


Figure 2:Coordinate system [10]

Neighbourhoods & Structuring Elements

A neighbourhood is the area around a pixel that is used to assess the pixel. In figure 2, the nine pixels around the middle pixel can be used in a series of computations to assess the middle pixel. A neighbourhood's size must be a square and be odd as there must be a middle pixel just like in the 3x3 in fig. 2.

The structuring element is a fundamental component of image processing. Applying many image processing techniques requires superimposing a configuration of pixels in the shape of a square (fig. 3) onto a pixel in an image. The origin of the configuration must line up with the targeted pixel in the image and then each cell in the structuring element interacts with the corresponding pixel in the image through some sort of arithmetic e.g. multiplication.

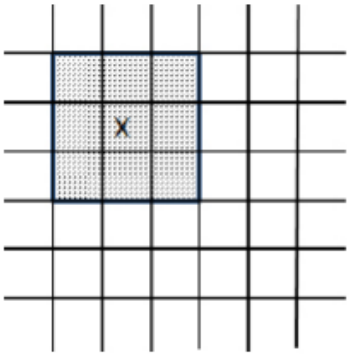


Figure 3:Structuring Element [10]

Noise & Filtering

Noise is the false representation of a pixel colour meaning pixels have the wrong value attached to them and this causes the accuracy of the image is poor. There are 4 main causes of noise in digital images:

- The camera sensor may be negatively affected due to environmental conditions, this can be due to the light shining directly in the lens causing a flare effect.
- Insufficient light levels causing the edges in the image to be faint or colours to be hard to detect causing pixels in an image will be appointed an incorrect colour.
- Signal interference when transmitting/sending the image from one device to another, this is mainly caused by the compression of an image while sending in order to make the file smaller.
- If dust or dirt particle are present on the camera lens this can cause parts of image to be disrupted and appear darker than they are.

Figure 4 shows how the increase in noise can drastically reduce the quality of an image and the clarity of the content in the images.

Figure 4: Noise in an image [11]

The noise removal algorithms or filtering reduce or remove noise from an image by smoothening the image, but this causes small details in an image to become obscure and hard to differentiate. The basics of filtering is to average of a 3x3 pixel block (neighbourhood) and set the middle pixel to the average of the 3x3 block. The larger the neighbourhood the more effective the filter is because there is a larger samples size to take the average from. The issue with a large neighbourhood is small details such as eye colour will be lost. Filtering can be very heavy on computation when applying the filter to colour images. In this project filtering will only be applied to grey images as this required less computation. As seen in fig. 5 below, the 3x3 block has an average and the middle pixel is set to that average.

Figure 5:Mean Filtering

Edge Detectors

An edge is defined as any sharp change in pixel colour/intensity. Edge detectors are used to identify the boundary of objects in images, this is achieved by comparing neighbouring pixel densities and identifying where there is a sudden change in colour. When there is a sudden change in pixel colour this can be identified as an edge as this is a boundary between two regions in an image, maybe foreground and background. To explain how the edges are seen by a computer the below figure shows the pixel intensities in a greyscale image.

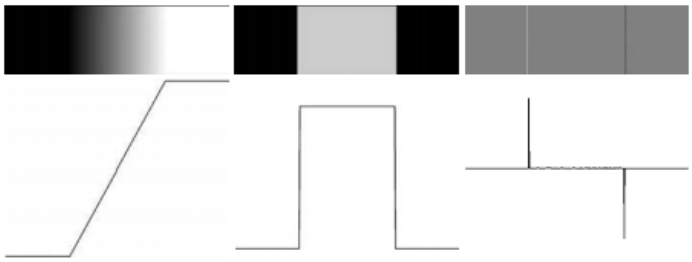


Figure 6:Edge Detection 1 [10]

In figure 6, going from left to right the first graph shows a gradual change in intensity from black to white, of the pixels. The slope of the graph determines if the edge is defined, meaning the steeper the slope, the more confident we can be that it an edge. The second and third graphs can be clearly defined as edge because the slope of their graphs are vertical.

Due to noise in an image, the graphs are not always as smooth, meaning it is hard to define an edge or not. In figure 5 below you can see the same edge but noise is applied.

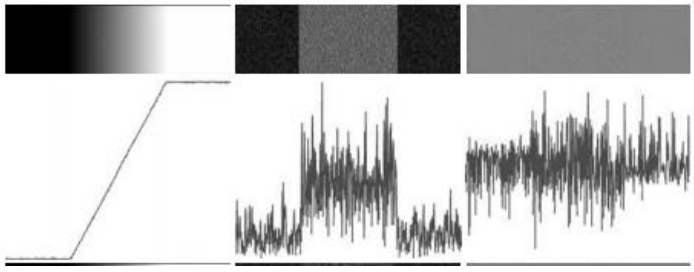


Figure 7:Edge Detection 2 [10]

The noise does not have much effect on the gradual edge but does have a serious effect on the sharp edges due to the two regions becoming closer in colour. The second graph can still be averaged and the edge can be found but the last graph has totally lost the edge. The most effective way to use edge detectors is to look for thick edges and avoid trying to find small edges in an image. Even after avoiding small edges, the thicker edges can still be affected by noise so making the edges more gradual can help give a ramp graph.

To remove noise from an image, the image must be smoothen, this is the process of averaging filtering image to make each region or object on the image have its own colour making edges defined like in fig 6. Once smoothening is done the edge detector can be applied and the edges can be easily identified on the output image. The selection of an edge detector was dependant on the detector that could deal with noise the best as light conditions can change rapidly.

Thresholding Greyscale

Thresholding assesses pixels on their grey level, any pixel grey level that is below a certain value will be set to black and pixels above the specified value will be set to white. The specified value can be set as a global threshold which means all pixels in the image will be compared to that threshold.

Adaptive thresholds are determined by the neighbouring pixels, all the neighbouring pixels are averaged and then a constant is subtracted from the average, the resultant value is then set as the threshold for target/middle pixel. The number of neighbouring pixels included in the average is determine by a block size, this is the size of the block around the pixels being accessed. The block size must be an odd number and is always a square, e.g. 5x5 or 9x9. Since a threshold for every pixel being assessed must be calculated there is a substantial number of computations and this takes far longer than a global threshold.

The benefit to an adaptive threshold is each part of the image can have different thresholds, meaning if there is a shadow on one part of the image then the adaptive threshold will be unbiased and evaluate the shadow based on its surroundings [10].

Closing

Closing is made up of two other processes, dilation, and erosion. Dilation is the gradual enlarging of a white object in a binary (black and white) image. Dilation is performed by have a structuring element, this is usually 3x3 blocks, imposed on the image. If any of cells in the structuring element touch a white(1) pixel, then the pixel being assessed is set to white(1). The below image shows the result of dilation.

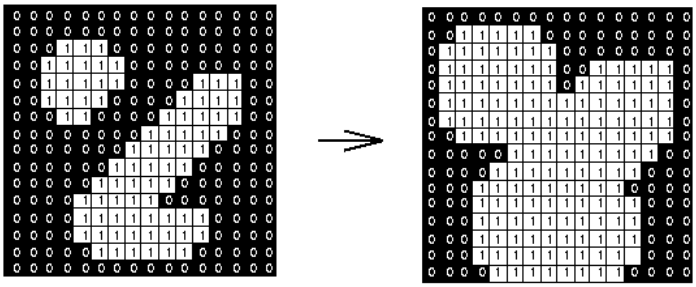


Figure 8:Dilation [12]

Erosion is the second process used in closing. Erosion is the gradual eroding of pixels at the edge of a white object in a binary image. As done in dilation, a structuring element is used but this time if any cell in the structuring element contacts a black(0) pixel then the current pixel being accessed is set to black(1). The resultant of erosion is shown below in fig. 9.

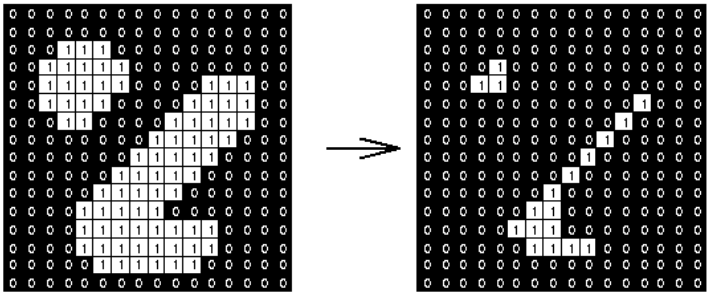


Figure 9:Erosion [12]

Closing is the process of performing dilation followed by erosion on an image, this result of closing is to fill in holes or gaps in the object. Figure 10 shows the resultant of closing.

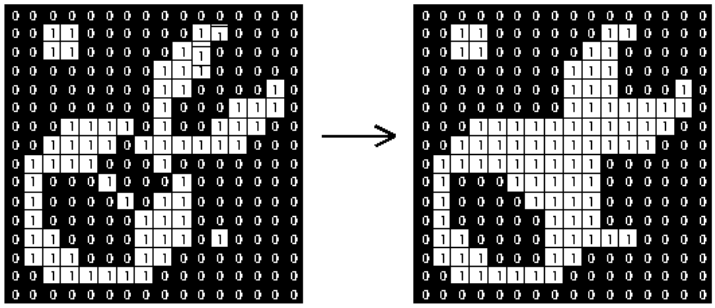


Figure 10:Closing [12]

4.2 Python

To understand the program design there are some python concepts that must be understood, these are concepts that are used throughout the code and give a more in dept understanding of how the program functions.

Each program will have global variables, these variables are used throughout the program, they can be many different types such an integer, image, array/matrix, or list. These global variables are used as inputs/arguments to functions, these functions are just segments of code that achieve one goal. Each function can then return a value that can be saved into one of the global variables.

Python has base functions that can come from a default library or an external library. For example, the library used in this project is the Open CV2 library which is an image processing library. This library contains functions such as Canny edge detector and denoising filter, each of these functions have their own arguments that are used within the function. To use a function from a library, the library is first imported and then you just type name of library followed by the function name with a full stop between them e.g. `cv2.Canny()` for a Canny Edge detector function.

Conditional statements are used to test a condition or a variable's state, the `if` statement is the most common. The `if` is following by a statement, if the state is correct then the indented code below the `if` is executed. An `else` commonly follows and this is executed if the statement is false.

Many sections of code require to be looped multiple times, for example when having to do a repeated computation on an image with slightly different inputs every time. The `while` loop is the first, which repeats indented code while the condition specified is true, if there is no condition present then the loop waits for a `break` statement to exit the loop.
Another type of loop is the `for` loop, this used for lists, where the indented code is run for every item in the list. For example, the following code would print every letter of the alphabet:

```
for letter in alphabet:
      print(letter)
```

Throughout the whiteboard capture program there are some commonly used functions, these are the basic functions of the program and perform simple tasks like displaying an image or saving an image as a file. Below is a list of the commonly used functions.

`def function(Inputs) =>` is a block of code that is executed when called, each function has a name, inputs, and outputs. Inputs are arguments that will be used in the function. These arguments are information that can be in the form of a number or maybe a picture. The arguments used are often the outputs from other functions. At the end of a function there is a `return` statement which returns the program back to where the function was called. The return statement is where the output can be specified, these outputs can then be used by other functions.

`cv2.imread(filename, type)` => reads in an image from a file to be used in the program, *`filename`* represents the path of an image and `type` specifies how the image should be read in, -1 meaning unchanged, 0 meaning greyscale and 1 meaning colour.

`cv2.imshow('name', img)` => creates a computer window displaying an image, `name` representing the name that will be displayed on the window and `img` representing the image that will be displayed in the window. The `cv2.imshow` function is commonly followed by the functions `cv2.waitKey(time)` where `time` specifies the amount of time in milliseconds that the window will be displayed for, if `time` is zero then the window will be displayed indefinitely until a keyboard key is pressed.

`cv2.imwrite('filename', img)` => creates a file with an inputted image taken from the program. `Filename` represents the path of where the image should be saved and `img` is the name of the image in the program that will be saved to the path.

5. Design

The focus of this project is to create an affordable, easy to use and readily available whiteboard capture system. Therefore, an affordable webcam costing €44 which can easily be connected to any PC through a USB was purchased for this project. The desired output of the program is a timeline of images, each image represents a content event. This timeline helps understand the thought process of the user.

To write the program, the programing language used is Python as this has an extensive image processing library called OpenCV. The OpenCV library has a two websites that contain all the functions and tutorials on how to use the library's functions [11] [12]. Most of the functions in the OpenCV library have multiple arguments and the websites explain what the functions do and what arguments are required to execute the function. In order to write and execute the Python code, an IDE (integrated development environment) is required. The IDE allows the OpenCV library to be used and helps identify errors when the program fails to execute. The choice of IDE doesn't affect the outcome of the code but for information purposes the IDE used is PyCharm.

The first action required of the user is to place the webcam with the whiteboard in frame, to help with this the program will create a window with a live video feed of the webcam so the user can place the webcam in a viable position in front of the whiteboard.

Once the webcam has the whiteboard in frame, the next task the user must perform is to confirm that there are no objects blocking the board. The webcam then captures a frame from webcam and this will be a reference frame that all following images will be compared against. The reference frame is only captured once for the first iteration of the program and is named 'prime', all following frames captured are simply named 'frame'.

The reference frame is then subtracted from every following 'frame' of the board, this subtraction reveals any object that is not present in the reference image. If there is a foreground object present, such as a person or arm, the image just captured will be declared 'null'. The system will continue to capture images of the board until there is no foreground object blocking the view of the board.

One issue that may arise that needs to be addressed in the development of the program is: if the if the whiteboard is filled with content and this is compared to the reference image, the content on the whiteboard may appear as foreground object as it is not common to both

images. The solution developed ensures that there must be considerable real estate taken up by the object in the frame to trigger the foreground object warning.

The image of the board will contain noise, any level of noise will hinder the edge detection. There are two types of filtering, linear and non-linear. The linear filtering simply takes the average of the neighboring pixels and replaces the middle pixel with that value. This method averages the whole image and causes details and edges in the image to be lost.
The non-linear filter also averages the neighboring pixels but before accepting a pixel into the averaging process it evaluates the colour of the pixel. If the colour of a pixel is drastically different from the target pixel colour then it does not take this pixel into account as it could imply an edge is present. This method of non-linear filtering preserves the edges which greatly benefits the next process of the program. Therefore, for this system the noise will be removed using a non-linear filter.

There are many different types of edge detectors, some are far simpler than others and can only be used for certain features. The first edge detector is the Sobel operator which is an edge detector which examines changes in colour. However, it only checks for these changes in the vertical and horizontal direction meaning if an edge is running in the diagonal direction it will not detect it. The main benefit to the Sobel edge detector is that it is very effective at finding all horizontal and vertical edges.

The next edge detector is Prewitt operator. This edge detector is even faster than the Sobel and uses a similar method to determine horizontal and vertical edges while failing to find diagonal edges. Even though Prewitt is faster than Sobel, it is only useful on high-contrast, noiseless images.

The third is the Roberts Cross operator, this is most commonly used on greyscale. The difference between the Roberts operator and the two mentioned previously is that is detects edges in the diagonal direction and is not effective at detecting horizontal and vertical edges.

The final edge detector is the Canny operator, this uses the basic Sobel operator as a base but performs the edge detection twice. The first run uses a low threshold which will result in a large number of edges being found and second run uses a high threshold which will produce only the strong edges. Using the two resultant Sobel images, an algorithm takes edges that are common to both images, resulting in the incomplete edges of the high threshold being completed using the low threshold image [13].

Once all of the edges in the image are captured, an algorithm must be applied to find the largest continuous edge in the image which will represent the outer edge of the board. The

corners of the whiteboard will be calculated and then the image will be cropped down to the corners. Since the whiteboard may appear to look like a quadrilateral rather than a rectangle, the corners of the image will be moved to make the whiteboard a less skewed in the image. This will be achieved by making a transform between the actual and desired coordinates of the corners.

To make the content have a high contrast with the whiteboard, the image must be changed to binary (black & white), where the writing is black and whiteboard is perfectly white. A threshold will be applied to the cropped image but since lighting on the whiteboard may vary, an adaptive threshold will have to be used to give a relevant threshold for each region of the whiteboard.

The clean binary image of the whiteboard will be then split up into a grid, each cell of the grid will be the size of an average character on the board, whether that be a letter or number. These cells will be stored and each time a new image is captured of the board, the cells will be compared with the previous cells. If enough cells have changed then the clean binary image of the board must be added to the timeline of images. If very few or no cells have changed between the old and new image, then the binary image is discarded. The new cells are stored and go onto to be compared to the next image captured of the whiteboard.

6. Implementation

The following program is formatted based on different functions. Each of these functions have a purpose and is used at a certain stage within the program. The code contains flags that determine which route the program will take. For example, if there is an object blocking the camera's view of the board the program cannot continue and must return to the capture stage of the program.

There are two flags in this code. The first flag, also known as the 'start flag' begins as zero for the first iteration, and changes to 1 for all following iterations. The second flag, used to distinguish between when an object is blocking the camera and when there is not, begins at 0 if no object is blocking the camera and changes to 1 when an object is blocking the camera.

Figure 11 illustrates how the program works and identifies what triggers the program to be interrupted. The left side of the flow chart is executed first illustrates how the program works and identifies what triggers the program to be interrupted.

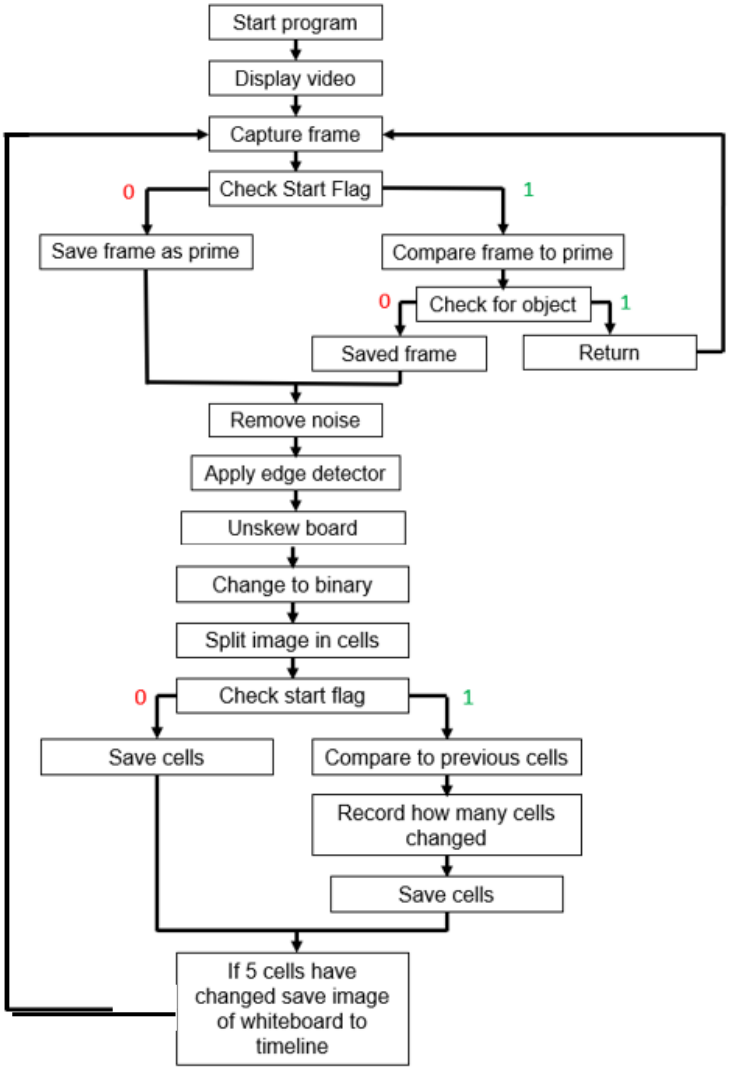


Figure 11:Flow chart

6.1 Declaration of variables

```
path = 'C:/Users/Ciaran/Desktop/Images/'
boxpath = 'C:/Users/Ciaran/Desktop/Images/Boxes/'
print("Package Imported")
start = 0
x, y, w, h = 0, 0, 0, 0
foregrnd = 0
count = 0
board = []
```

Figure 12:Global variables

At the top of the code the global variables are declared, as seen in fig. 12 above. These variables are used throughout the code in multiple functions and may change for every cycle of the program. Both the `path` and `boxpath` variables are strings that refer to the location of where images are stored. The `path` string is where the main images in the program are saved, while the `boxpath` variable is where the cell images are stored.

The variable `start` refers to a flag and has two states, on (1) or off (0). This variable is off at default, this is so the program can capture the reference image and does not compared to any previously captured images for the first iteration. The start flag turns on when the program has completed its first iteration.

The `x, y, w, h` variables shown in fig. 12 represent the position, width, and height of the board in the camera frame. When the camera is positioned, it cannot move, meaning every time the board is captured it must be in the same position in the frame. The position and dimensions of the board are used each time a new frame is captured to isolate the whiteboard.

The `foreground` variable is another flag. The flag is off when no foreground object is present and on when a foreground object such as an arm in frame. This flag is necessary when a foreground object is in frame, the frame captured is declared null as content on whiteboard is blocked. Another frame must be captured before the program can move on.

The `count` variable is an integer, every time an image is added to the timeline, the count increments. This count is used to number the timeline images so they are sorted correctly.

The `board` variable is a list that stores all the points along the contour of the board. Similar to the position and dimensions of the whiteboard, the `board` variable is used throughout the program to unskew the whiteboard when the camera is placed at an angle in relation to the whiteboard.

6.2 Capturing a Valid Frame from the Webcam

```
def VideoCap():
    cap = cv2.VideoCapture(0)
    cap.set(3, 1920)
    cap.set(4, 1080)
```

Figure 13:Setting Capture Device Code

The first function in the code is the video capture. Figure 13 shows the input camera being set as default capture device on your computer and then set the video quality to 1080p.

```
    while True:
        success, vid = cap.read()
        cv2.imshow("Video", vid)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyWindow('Video')
            break
    frame = cap.read()[1]
```

Figure 14:Webcam Window Code

The next part of the video capture function is a while true loop which can be seen in figure 13 above. This loop displays the live video of the webcam. This allows the user to reposition the camera in a suitable place, ensuring that the complete whiteboard is in the frame. As can be seen in figure 13, the if statement in the while loop ensures that the loop continues to run until the 'q' key on the keyboard is pressed. Once 'q' has been pressed the window displaying the live video feed is closed and a frame is captured using the function cap.read( ). Figure 15 below shows an example of a frame captured from a webcam..

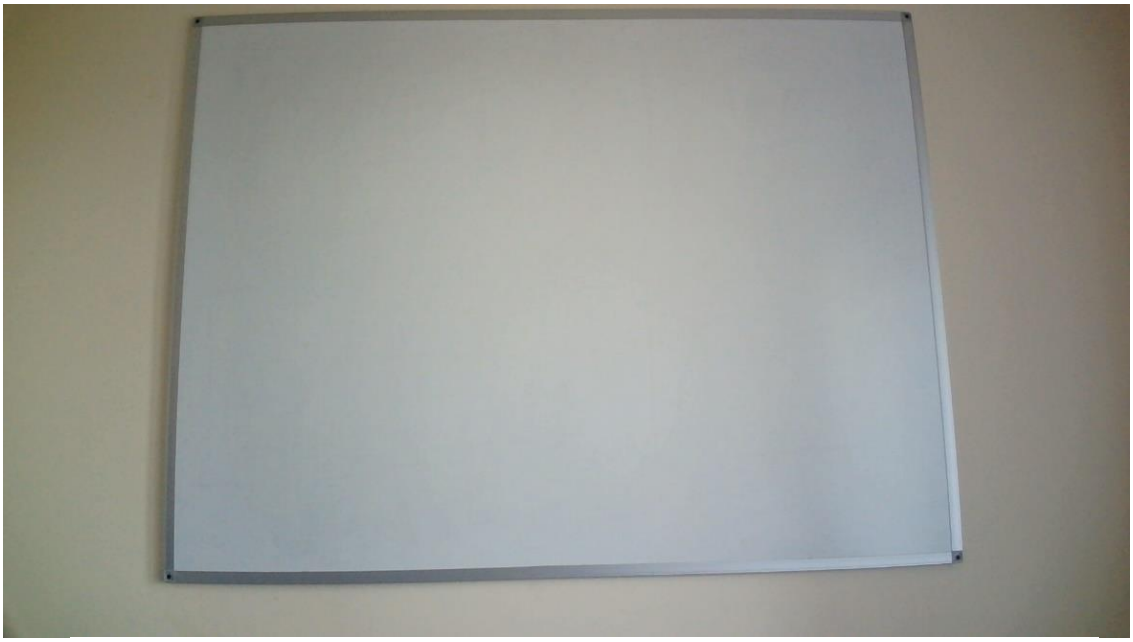


Figure 15:Frame Captured

```
if start == 0:
    cv2.imwrite(str(path) + 'prime.png', frame)
    foregrnd = 0
else:
    cv2.imwrite(str(path) + 'frame.png', frame)
    prime = cv2.imread(str(path) + 'prime.png', 1)
    sub = cv2.subtract(prime,frame)
    subgrey = cv2.cvtColor(sub, cv2.COLOR_BGR2GRAY)
    ret, thrsh = cv2.threshold(subgrey, 50, 255,
                               cv2.THRESH_BINARY)
    if cv2.countNonZero(thrsh) > 1000:
        foregrnd = 1
    else:
        foregrnd = 0
return frame, foregrnd
```

Figure 16:Foreground Object Code

The above code in fig. 16 decides what to do with the image that has just been captured. The code first checks the start flag, if the program is capturing its first image then frame from the webcam is named ‘prime’, this prime image is used as a reference frame for all the following frames captured. It is important that the ‘prime’ image only contains the empty whiteboard and no other objects. The image in fig. 15 is an example of a ‘prime’ image.

If the start variable is 1, then the frame is just named ‘frame’. The ‘prime’ image is read in and then subtracted from the ‘frame’ image that was just captured. This subtraction means that any content that is not common to both images is coloured as shown in fig. 17(a).
The resultant image is changed to a greyscale image and then a threshold is applied with any pixel above the value of 50 will be white and all pixels below the value of 50 will be black as seen in fig. 17(b). The white pixels in the threshold image are then counted. If the count is above 1000, then there is an interfering foreground object present and the foreground flag is set, otherwise the foreground flag is not set. The value of 50 was chosen as the threshold value after vigorous testing to verify that all of the colour in the image was translated to white and the value of 1000 made sure that the content on the whiteboard could not trigger the foreground object flag.

(a) (b)

Figure 17:Foreground Object (a) Colour (b) Binary

6.3 Contour detection

```
def ContourDet(frame, start, x, y, w, h, board):
    imgBlur = cv2.bilateralFilter(frame, 7, 15, 15)
    imgCanny = cv2.Canny(imgBlur, 10, 100)
    kernel = np.ones((5, 5), np.uint8)
    imgDil = cv2.morphologyEx(imgCanny, cv2.MORPH_CLOSE, kernel)
```

Figure 18:Canny Edge Detector Code

The second function in the code is the contour detection function as shown in fig. 18. The arguments of this function are: the frame image, start flag, location, dimensions, and the contours of the board. The operation of this function is to isolate the whiteboard in the frame image and output an image with only the whiteboard present . Sometimes the camera may be at an angle and this function flattens the whiteboard in the image. For the first iteration of this function the location, dimension, and contours of the image are not determined thus the arguments are inputted as zero.

The frame image is firstly put into a bilateral filter. The bilateral filter takes in 3 arguments, an input image, radius, and sigma values. The input image is the image the filter will be performed on. The radius is half the size of the neighborhood used in the averaging process.

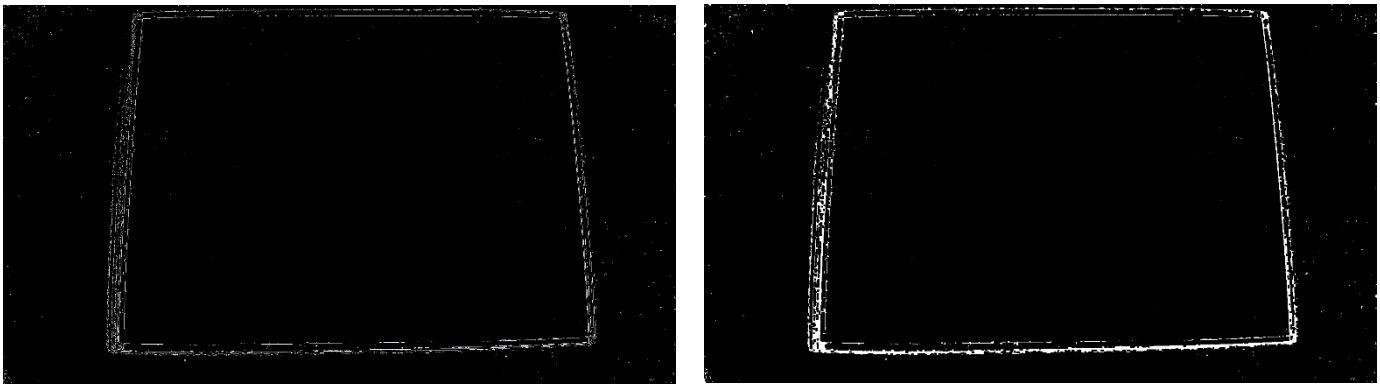
The sigma values set how much of an intensity difference there must be for a pixel to not be included in the averaging arithmetic. This means if there is an edge nearby the edge pixels will not be considered when calculating the average. Edges have high intensity values and this will corrupt the average. Since the bilateral filter takes pixel intensity into account, it preserves edges which in turn helps with the Canny edge detector.

The value seven was chosen based on when there is poorly distributed light, there are many different shades of colour in the frame so the average must be taken over a small area. The sigma values are relatively small because the whiteboard edge is a light grey and does not have high contrast with the wall.

The Canny edge detector is used to find the edges in an image based on sharp changes in pixel colour. The Canny edge detector takes in 3 arguments, an input image, min value, and max value. Any edges intensity value below the minimum value will not be considered as an edge, any edge intensity value above the max value is an edge but any value between the min and max value is only considered an edge if it is attached to an edge that is above the max value.

The values of 10 and 100 were chosen as high threshold only accepts the obvious edges, which is only the edge of the whiteboard and the lower threshold can fill in the discontinuities of the whiteboard edge. The Canny image is shown below in fig. 19(a) and you can see that the edges are specked due to noise.

Next a kernel is declared, as explained in the background section a kernel is a structing element used for closing function. This closing process makes the edges in the canny image much clearer, therefore making it easier for the contour of the whiteboard to be found. The size of the kernel was chosen to be relatively small since the edges found are so close together and a larger kernel would be unnecessary computation. Figure 19(a) shows the closing result which contains a more obvious outline of the whiteboard.



(a) (b)

Figure 19: Whiteboard Edges (a) Canny (b) Dilated

```
contours, hierarchy = cv2.findContours(imgDil, 1, 2)
area = 0
if start == 0:
    for cnt in contours:
        x, y, w, h = cv2.boundingRect(cnt)
        imgCont = cv2.rectangle(imgBlur, (x, y),
                                (x + w, y + h), (0, 255, 0), 2)
        if area < (w * h):
            area = w * h
            board = cnt
    x, y, w, h = cv2.boundingRect(board)
cv2.imwrite(str(path) + 'contours.png', imgCont)
```

Figure 20: Finding Contours

In this section of code in fig. 20, the goal is to find the location of the whiteboard in the image, this is achieved by finding the longest edge in the frame. For visual purposes a bounding rectangle of each contour is printed on the image as shown in fig. 21.

The dilated image is fed into the `findcontours` function which outputs all the contours on the image and their hierarchy. Hierarchy declares if a contour is inside another contour. The arguments of `findcontours` are an input image, retrieval method, and compression method.

The retrieval method is level of hierarchy with 1 being there is only a parent and child relationship given. The compression method is how many points along the contour are stored, either just the corners points of the contour or all the points along the contour. Unfortunately, no matter which method was chosen as the compression method, the contours always included every point along the edge.

The start flag was checked and if zero then a bounding rectangle of each contour was printed onto the filter image. The start flag check means that the whiteboard location was only found once using the first frame captured. As you can see in fig. 21, the largest green rectangle encases the whiteboard. Since the board is not perfectly symmetrical and does not line up with the bounding rectangle, the board must be fitted to the size of the bounding rectangle to achieve a flat image of the board.

Figure 21: All Contours Detected

The `if area < (w*h):` line of code in fig. 20 compares the size of each bounding rectangle, the contour with the largest area is found and stored in the `board` variable.

The coordinates of the top left corner of the contour are saved as (x, y), the width and height of the largest contour are saved in the variables (w, h). The downside to this code is the contour of the board was found in the first iteration of the program. The location and dimensions used throughout the program's life cycle. The program assumes the camera is not moved and whiteboard is in same location in each frame captured.

```
peri = cv2.arcLength(board, True)
corners = cv2.approxPolyDP(board, 0.06*peri, True)
n = corners.ravel()
random = ([n[0],n[1]],[n[2],n[3]],[n[4],n[5]],[n[6],n[7]],)
sort1 = sorted(random, key=lambda x: x[0])
lcorners = (sort1[0], sort1[1])
rcorners = (sort1[2], sort1[3])
leftsort = sorted(lcorners, key=lambda x: x[1])
rightsort = sorted(rcorners, key=lambda x: x[1])
```

Figure 22: Finding Corner Coordinates Code

Figure 22 shows the code used to find the corners of the whiteboard. Firstly, the perimeter of the board is calculated with the `arcLength` function, with *`true`* meaning that the contour must be a closed shape. The `approxPoly` function is used to find the four corners of the whiteboard. This is achieved by finding a shape that has a perimeter within 6% of the perimeter calculated, again *true* stating the shape must be closed. The 6% was chosen based

on testing. If this value is any lower than 6% a shape with less than four corners would be outputted and a value any higher would output more than four corners which is viable. The `approxPoly` function forms a list of coordinates, these coordinates are the corners of the board. Unfortunately, the coordinates are not in any specific order so it is unknown which coordinates correspond to each corner.

The coordinates of the corners are embedded in a list that cannot be manipulated. The `ravel` function forms a list of the numbers but removes them from their pairs(x, y). The coordinates are then put back into a list in their pairs (x, y) called `random` but this time the list can be manipulated and organized.

The list `sort1` is corners sorted from smallest to largest x coordinate, this sorts the coordinates in the horizontal direction from left to right. This means the first two objects in the list correspond to the left corners. The `sort1` list is then split into two lists, one list with the left corners(`lcorners`) and one with the right corners(`rcorners`).

The next two lists in fig. 22 are `leftsort` and `rightsort`. These two lists take the two previous lists, `lcorners` and `rcorners`, and sort them in terms of the vertical direction/y coordinate. Therefore, the top corners are the first object in each list. With the two lists `leftsort` and `rightsort` we know which coordinates correspond to each corner and can use these corners to move the whiteboard to a desired position in the frame.

```
pts1 = np.float32([leftsort[0], rightsort[0],
                   rightsort[1], leftsort[1]])
pts2 = np.float32([[0, 0], [w, 0], [w, h], [0, h]])
M = cv2.getPerspectiveTransform(pts1, pts2)
imgCrop = cv2.warpPerspective(frame, M, (w, h))
cv2.imwrite(str(path) + 'crop.png', imgCrop)
return x, y, w, h, board
```

Figure 23: Perspective Transform Code

The corners of the whiteboard are then put into a list(`pts1`) with the first corner being the top left and then continuing clockwise around the whiteboard. The second list(`pts2`) is the coordinates where the corners will be moved to. The bounding rectangle is ideal rectangle that the whiteboard can fit in so we use the width and height of the bounding rectangle to declare the desired corners.

The function `getPerspetiveTransform` takes the actual corner points of the whiteboard and the desired corner points to make a transform. This transform is used to superimpose every pixel inside the whiteboard contour into a new image the size of the bounding rectangle. The `imgCrop` is the resultant image after all the pixels have been superimposed into the bounding rectangle(w, h) using the transform `M`. The coordinates, size and contour of the

board are then returned to be used for the next frame captured. Figure 24 shows the original frame compared to the cropped and flat image of the whiteboard.

(a) (b)

Figure 24: Captured Frame (a) Original (b) Cropped

```
def cleanimage():
    img = cv2.imread(str(path) + 'crop.png', 0)
    imgBW = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                  cv2.THRESH_BINARY, 21, 5)
    noise = cv2.fastNlMeansDenoising(imgBW, None, 50, 7, 51)
    cv2.imwrite(str(path) + 'noise.png', noise)
    return
```

Figure 25: Clean Image Code

The next function in the code is `cleanimage()`, shown in fig. 25. The operation of this function is to make the content on the whiteboard stand out. This is accomplished by making the image binary. Since the lighting on the whiteboard may not be even due to bad lighting or glare, an adaptive threshold must be used.

The arguments to the adaptive threshold are an input image, set value, threshold calculation method, binary method, neighborhood size, and constant.

- The set value is the intensity of white, 255 being the max intensity for a high contrast.
- The threshold calculation is performed with the gaussian method. This method identifies outliers in the neighborhood and ignores them, therefore any random high or low intensity pixels do not cause noise. The gaussian method is the only viable method as the others accept too much noise.
- The binary method means any pixel above the threshold is white and any pixel below threshold is black. The binary method is the default, the alternate method inverts the colours in the image which is irrelevant in this scenario.
- The neighborhood size is set to a large number. This means a larger sample size is used. A large sample size increases accuracy and is extremely effective for binary images.
- The constant is a value that is subtracted from the average to determine the threshold. The higher the constant the low threshold is. The value five was chosen based on testing

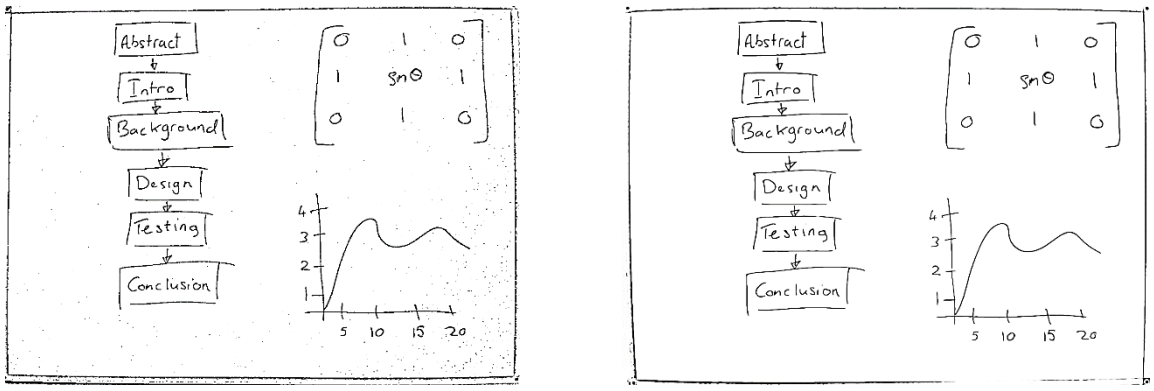
where any constant below 5 starts to introduce noise and any value above 5 causes faded characters and lines.

Figure 26(a) below shows a binary image of the whiteboard with content on it, this is the resultant image of the adaptive threshold.

The threshold binary image still contains noise, to remove the noise and smoothen the content a denoising filter is used. The `fastNlMeansDenoising` function arguments are: the binary input image, 'None' means not a colour image, the strength of the filter, the neighborhood size, and the search window.

The higher the strength value the more noise removed from the image but the image loses detail. Since the image is binary the value must be relatively high as the contrast between black and white is high. The neighborhood size is the area from which the average is calculated from. The value seven is chosen as the noise is not concentrated, so only a small sample was only needed to remove it.

The search window, with binary images this value must be high but does increase computation time.



(a) (b)

Figure 26: Isolated Whiteboard (a) Noisy Image (b) Filtered Image

6.4 Splitting the Image into Cells

```
def gridimage(start, change, nochange, count):
    noise = cv2.imread(str(path) + 'noise.png', 0)
    ni = np.array(noise)
    height, width = ni.shape[:2]
    boxesH = 25
    boxesW = 44
```

Figure 27: Grid Code

The last function shown in fig. 27 splits the image into a grid, each cell in the grid will be saved as an image. These cells are used to compare a current image to a previous image. The cells are compared on their mean colour. If the mean of a cell has changed considerably between two images of the whiteboard, the program assumes that the content in that grid has

changed. If enough cells have changed then the current image of the board is added to the timeline.

Firstly, the cropped and filter image of the whiteboard is read in, the image is then changed into array format to find the height and width of the image. The variable `boxesH` and `boxesW` are number of grids in the horizontal and vertical direction respectively. After writing on the board with different size writing, the dimensions of an average sized letter were estimated.

Comparing the letter's height and width with the dimensions of the board, it was calculated that the dimensions of the average letter were 1/25 the height and 1/44 the width of the board. Given those fractions an average letter could fit 25 iterations in the vertical and 44 iterations in the horizontal. Each grid is estimated to encompass one letter.

```
for i in range(boxesW):
    for x in range(boxesH):
        this = ni[x * height // boxesH:(x + 1) * height // boxesH,
               i * width // boxesW:(i + 1) * width // boxesW]
        if start == 0:
            cv2.imwrite(f'{str(boxpath)}box-{i}-{x}.png', this)
        else:
            this_array = np.array(this)
            prev = cv2.imread(f'{str(boxpath)}box-{i}-{x}.png', 0)
            prev_array = np.array(prev)
            if abs(np.mean(this_array) - np.mean(prev_array)) >= 20:
                change += 1
            else:
                nochange += 1
            cv2.imwrite(str(boxpath) + f'box-{i}-{x}.png', this)
start += 1
if change > 5:
    cv2.imwrite(str(path) + f'final-{count}.png', noise)
    count +=1
return start, count
```

Figure 28: Grid Cropping Code

There are two 'for' loops as shown at the top of fig. 28, meaning that the inside for loop must increment through its values before the outer loop can increment. The outer loop increments through the width and the inner loop increments through the height. There are forty-four images cropped from each column in the image, the image contains twenty-five columns.

Inside the two loops, the program takes the array of the current image and crops it by using the start and end points of the cell, firstly in the vertical direction and then the horizontal.

After each cell is cropped the start flag is checked. If the start flag is zero, then the cell is just saved as there is no previous cell image to compare to as the program is running through its first iteration. If the start flag is one, then there is a previous cell to compare against. The

previous cell image is read in and its average is calculated. The mean colour of the current and previous cells are compared by subtracting the means. If the difference in colour mean is great than twenty then the cell is deemed to have changed and the `change` variable is incremented. Otherwise, the `nochange` variable is incremented. The value of twenty is chosen by comparing a cells mean before and after a character was drawn in it.

After this cropping stage of the program, the program is deemed to have completed an iteration and the start flag is set to one.

If the `change` variable is greater than 5, meaning 5 cells have changed, then the current image of the whiteboard is saved. The count variable is used to declare the image's location in the timeline. Once an image is saved to the timeline the count is incremented for the next image. The start flag and `count` value are returned to be saved and used in the next iteration of the code. Figure 29 below shows the size of the cells compared to the whiteboard.

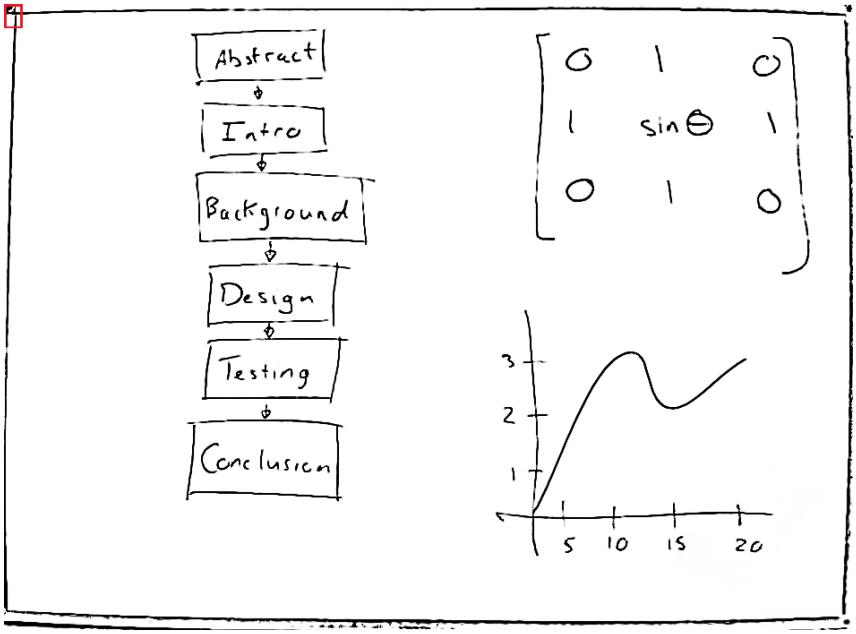


Figure 29: Cell Size

6.5 Main Code

```
try:
    while True:
        frame, foregrnd = VideoCap()
        if foregrnd == 0:
            x, y, w, h, board = ContourDet(frame, start, x, y, w, h, board)
            cleanimage()
            start, count = gridimage(start, 0, 0, count)
        else:
            print('Foreground Object')
        time.sleep(2)
except KeyboardInterrupt:
    pass
```

Figure 30: Main Code

Figure 30 shows the main part of the code where all the functions are called. The `try:` line of code is paired with the `except keyboardInterrupt:` which runs the code within the 'while' loop until there is keyboard key held down for approximately 2 seconds.

The `VideoCap()` function is first to be called and when completed the outputs are saved in the global variables `frame` and `foregrnd`. The `foregrnd` flag is checked and if not set then there are no objects blocking the board and the frame captured is valid. If the flag is set then the text 'Foreground Object' is displayed on the screen. This text is for testing purposes.

The `ContourDet()` function is called and global variables are inputted, once again the outputs from the function are saved. These outputs are the location, dimensions, and contour of the whiteboard. `Cleanimage()` has no inputs or outputs so that doesn't need any outputs to be saved. The last function to be called is `gridimage()` which sets the `change` and `nochange` to zero so every time the function is called the variables are reset.

After every iteration of the program the is a time delay of 2 seconds because when testing, enough time is needed to check that the images have saved correctly. The program can be ended by the user with a long press of a keyboard input.

The average output file of the program is show in fig. 46 in the appendix.

7. Testing

Testing was done with the limitations of the program in mind, this is done by only testing on variables that the program can handle. The main limitations of the program are as follows:

1. The camera must stay in the exact position it was placed in when the first frame was captured. This is due to the prime image being compared to every following frame. If the whiteboard is not in the same location then it will appear as a foreground object and the program will never continue.
2. The entire assembly of the whiteboard must be in the camera frame, preferably with the outside edge visible as this is usually the most obvious edge. If the entire border of the whiteboard is not present in the camera frame then the contour detection will not be able to find the corners and therefore the whiteboard cannot be isolated from the image.
3. Direct sunline into the camera lens will cause the entire whiteboard frame to not be fully captured and therefore the whiteboard cannot be located in the image.
4. Lighting conditions cannot change drastically throughout the capture session. Since the base image is captured at the start of the session and is compared to every image until the end of the session. The last image may have different lighting conditions compared to the first image, meaning the program may think the difference in lighting on the board is a foreground object and not continue.

The tests were limited to a small room where the webcam can only be placed between 1-1.8 metres away from the board. The best quality lighting between 10am and 2pm as that is when the sun gets even coverage of the room. A big issue with the lighting is that the natural light comes from the right side of the whiteboard and therefore the content on the right side of the board is damaged due to a slight glare. Figure 31 shows a picture of the testing environment in the limited space available.

Figure 31: Testing Environment

There were 4 different tests performed and they are as follows:

1. Varying distance from the board.
2. Varying angle of capture.
3. Varying light conditions.
4. Varying colour of marker.

For each test being performed the three other conditions are kept constant as well as the content on the whiteboard was identical or very similar throughout the tests. Since lighting can change fast, test 1, 3, and 4 are done within a small-time frame to keep the lighting as consistent throughout each test. At the start of the video capture function the camera quality can be set, the base quality is 1920x1080 pixels. Test 1 is also performed with 1280x720 pixels to show how decrease in capture quality affects the capture systems output quality. This test will show the limits of the program with a lower quality webcam.

7.1 Distance Test

The distance test is performed by placing the webcam at three different distances directly in front of the whiteboard. Due to the small room where the tests are being performed, the webcam can only be placed between 1-1.8 metres from the whiteboard. The three different distances are 1, 1.4 and 1.8 metres, these values can show how the gradual increase in distance affects the capturing process.

7.2 Angle of Capture Test

The angle of capture is the change of the theta angle in the fig. 30, this is to test how effective the camera can detect and unskew the whiteboard, if the whiteboard is detected how clear is the content displayed on the output image. The camera is moved parallel to the whiteboard to increase the angle, each time the angle is increased, the distance between the camera and the whiteboard had to be increase due to the limitations of the testing environment. This is not an ideal test as the distance should be kept constant to isolate the effect angle has on the capture system.

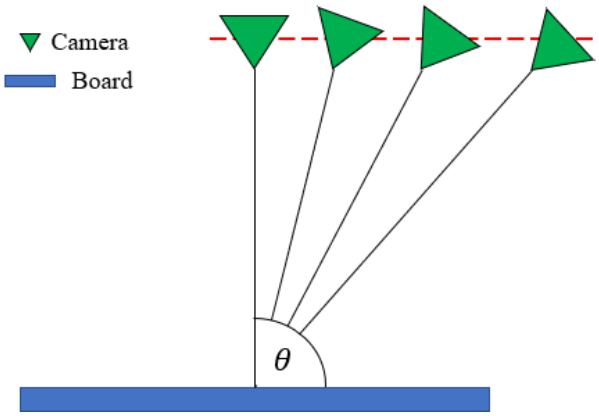


Figure 32:Angle Test Positioning

7.3 Lighting Test

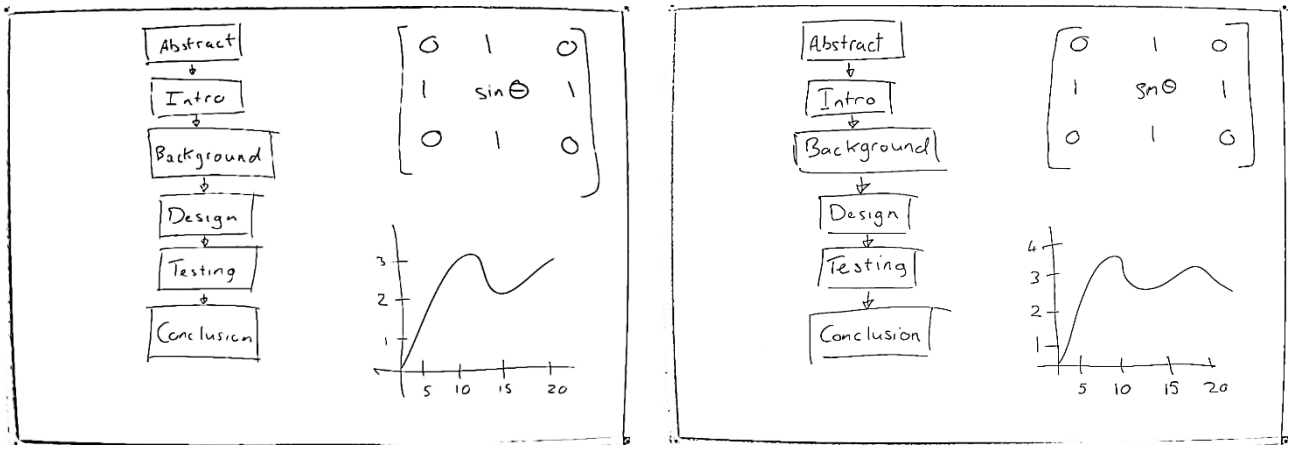
The lighting test will be done in three different scenarios, the first is with optimal lighting at midday, the second is in the evening with minimal natural light and the third is performed with no natural light after sunset using a lamp as the light source. In a lecture hall setting there are halogen lights which are a great artificial light source, but unfortunately this type of lighting cannot be replicated so this lighting test is just testing how decreasing the quality of the light on the board affects the capturing process.

7.4 Colour of Marker Test

When writing on a whiteboard there are 4 basic colours that are used, these are black, blue, green and red. Black is used in every other test and has the highest contrast of all three colours so is not included in this test. This test will compare the capability of the system when using the 3 colours other than black on the whiteboard. The content on the board when testing each colour is kept similar in order to have an accurate comparison between the colours.

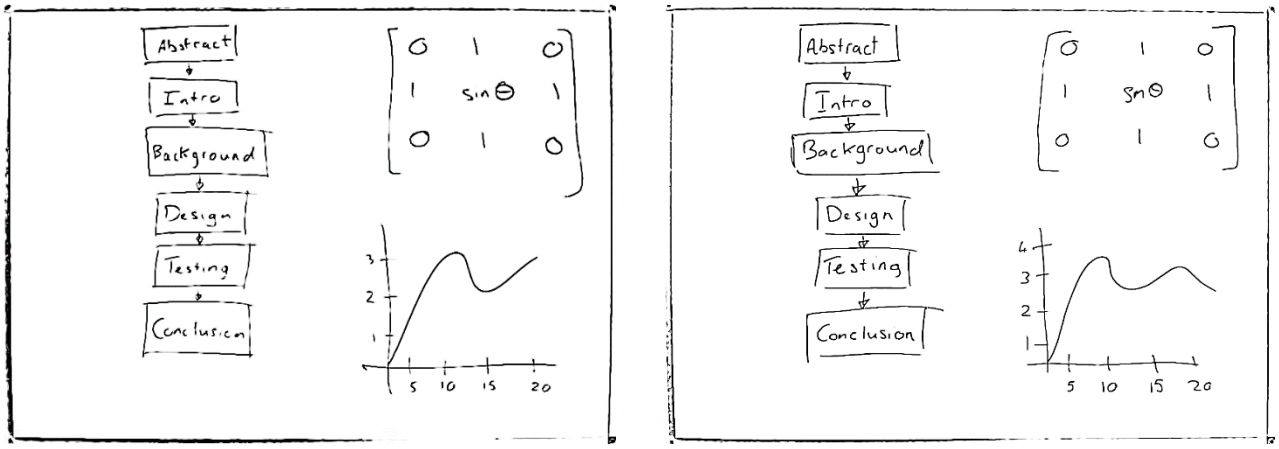
8. Results

8.1 Distance Test



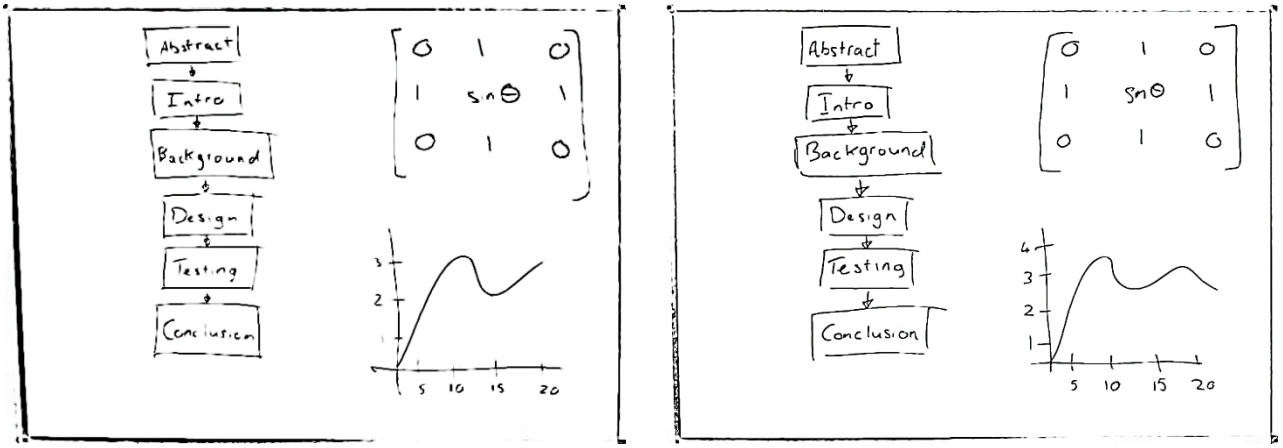
(a) (b)

Figure 33: 1m Distance (a) 720p (b) 1080p



(a) (b)

Figure 34: 1.4m Distance (a) 720p (b) 1080p



(a) (b)

Figure 35: 1.8m Distance (a) 720p (b) 1080p

The distance test is the simplest method of measuring the capability of the capture system because most will position the webcam directly in front of the whiteboard just at different distances. The results of testing the 1080p unfortunately doesn't show us the limits of this capture quality because the output quality between 1 meter and 1.8 metres is not considerable as seen in fig. 33-35.

As for 720p capture quality a clear decrease in output quality is seen from fig. 33-35, this can be assumed to be down to the fewer pixels in the 720p captured images. Due to the base quality used throughout the design and coding phase of the project being 1080p, the arguments used in each function were chosen based on how 1080p performed. If the arguments had to be chosen for 720p, each function would have to allow for a larger error, this allowance would cause more noise to be accepted when 1080p is the capture quality.

If we compared 720p and 1080p, there are 2.25 more pixels in 1080p than 720p, meaning that if the program is designed to be effective for 720p, then 1080p capture quality could potentially have twice as much noise and this will drastically hinder the programs output quality at 1080p. The program therefore must be design separately for 720p and 1080p to deal with the different number of pixels in both capture qualities.

It can be seen in fig. 33-35, when capturing in 1080p quality, the program outputs clear images at distances between 1-1.8 metres, this is a promising result as it is unrealistic that a camera be placed further than 1.8 metres from the board. Trying to place the camera within 1 meter of the whiteboard is not possible while also keeping the outside edge of the whiteboard in the camera frame.

Distance is the main variable that changes when deciding where to place the camera but some may struggle to find a viable position directly in front of the whiteboard meaning the camera may be placed at an angle.

8.2 Angle Test

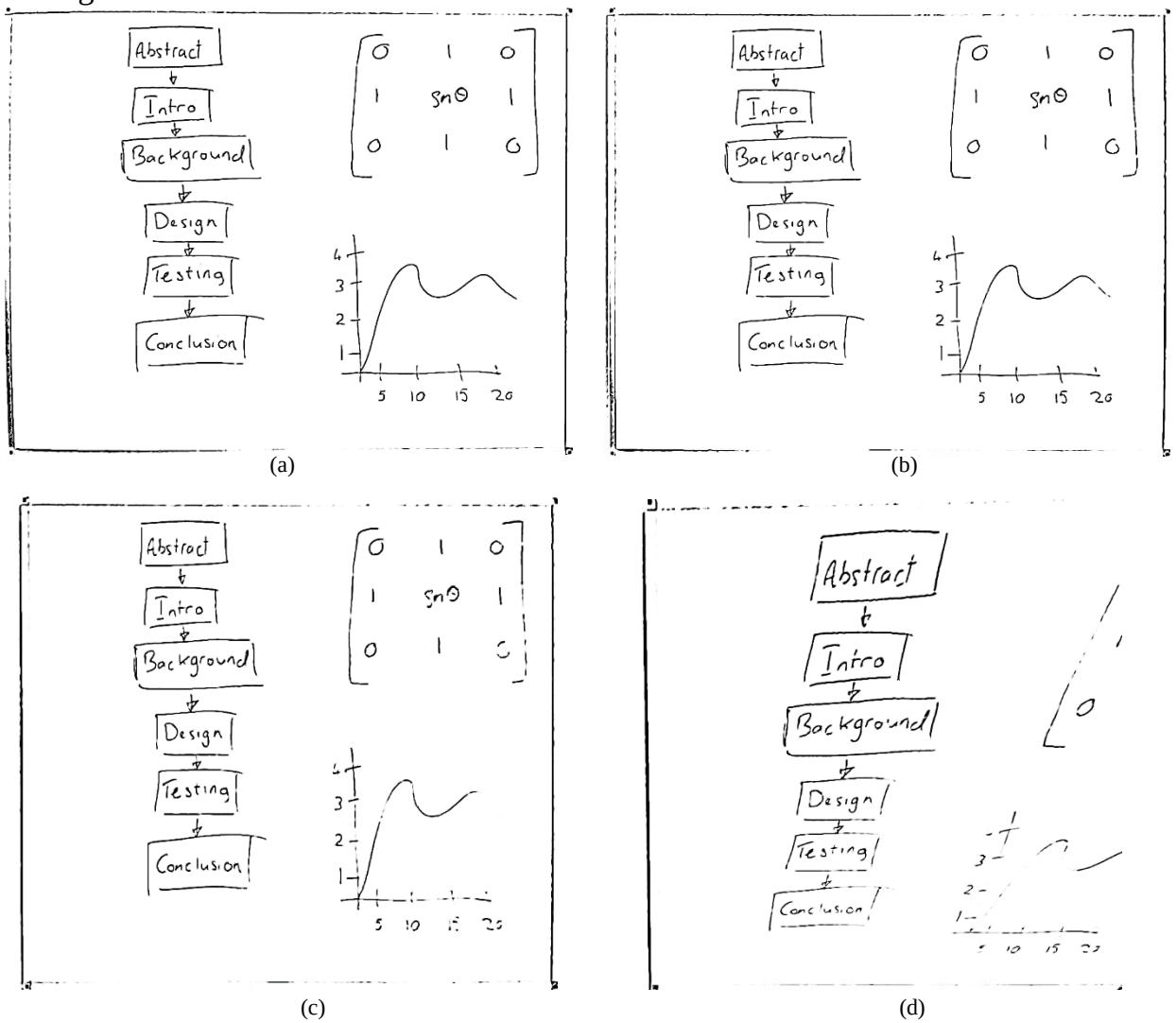


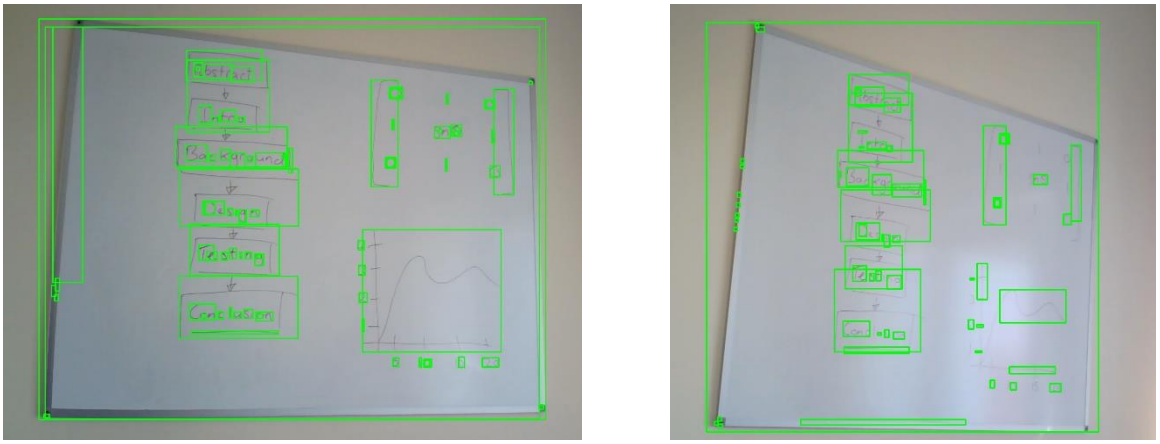
Figure 36: Angle Test (a) 10° (b) 20° (c) 30° (d) 40°

The angle test is used to assess the capability of the system when a viable camera position cannot be found with the whiteboard correctly in frame. The tests the limit of the system when placed at an angle, an increase in the angle causes an increase in the severity of the quadrilateral.

The first observation from the results in fig. 36 to address is the decrease in quality, from left to right, of the content on the whiteboard for each angle test. This quality change is due to the camera position moving to the left, therefore the right-hand side of the board is further from the camera than the left hand side of the board.

Another observation can be made about the size of the pictures for each angle test. As the angle increases the size of the output image decreases, this is due to the whiteboard appearing smaller in the frame, therefore the bounding rectangle of the whiteboard is smaller. When the flattening process occurs, the content on the whiteboard is stored in a smaller image each

time the angle of capture increases. Figure 37 shows the difference in size of the bounding rectangle when the angle of capture increase from 10 to 40 degrees. The bounding rectangle is the largest green rectangle in each image. The size of the bounding rectangle is based on the perceived size of the whiteboard in the image, the greater the angle the smaller the whiteboard appears to be.



(a) (b)

Figure 37: Bounding Rectangles for Angle Test (a) 10° (b) 40°

Taking into account the viability of the system in relation to changing the angle of capture, the largest angle the camera can be placed at to receive a viable output is 20 degrees. Any angle above 20 degrees will cause the quality to considerably decrease and cause content to have discontinuous lines.

8.3 Lighting Test

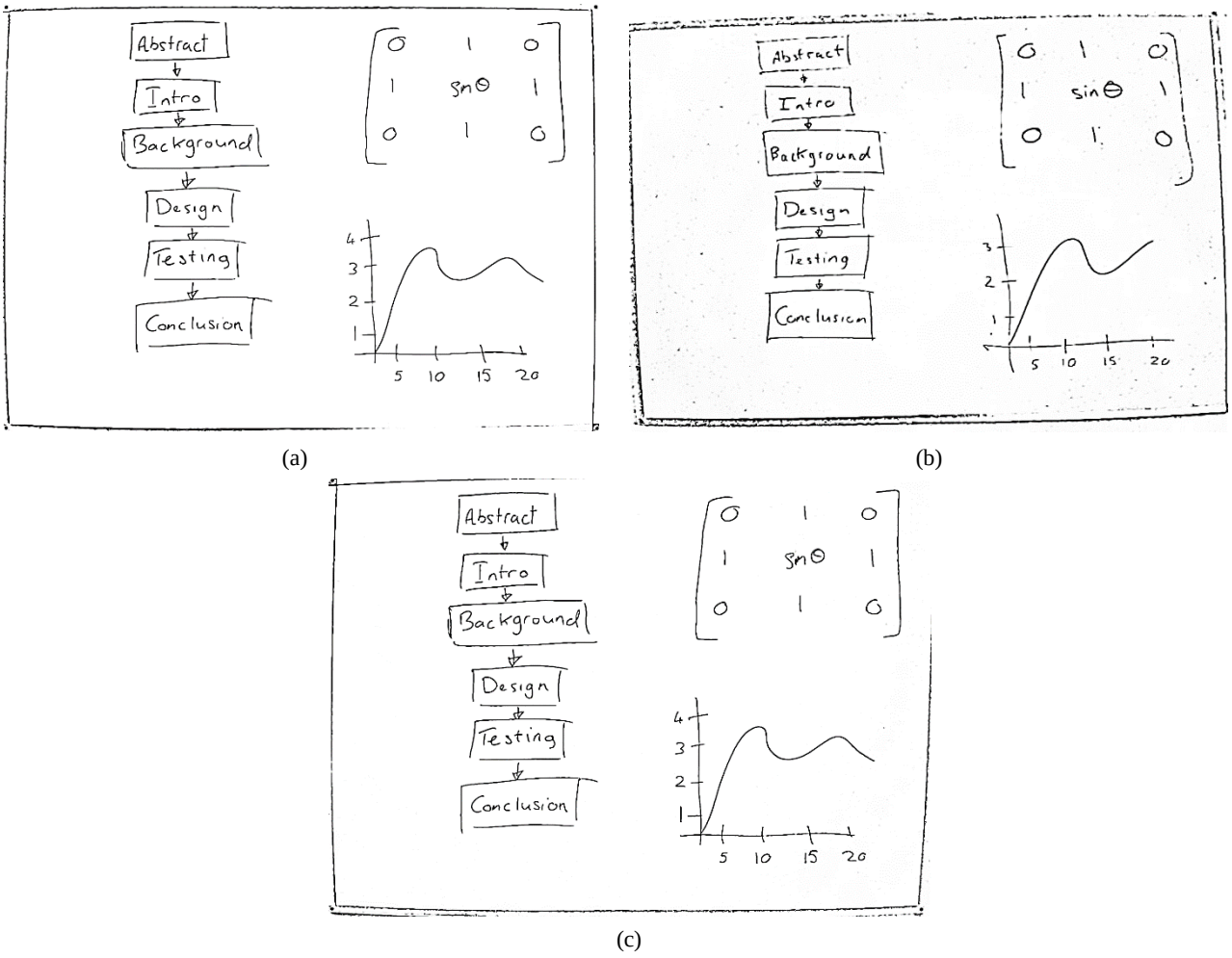


Figure 38: Lighting test (a) Optimal Lighting (b) Dull Lighting (c) Artificial Lighting

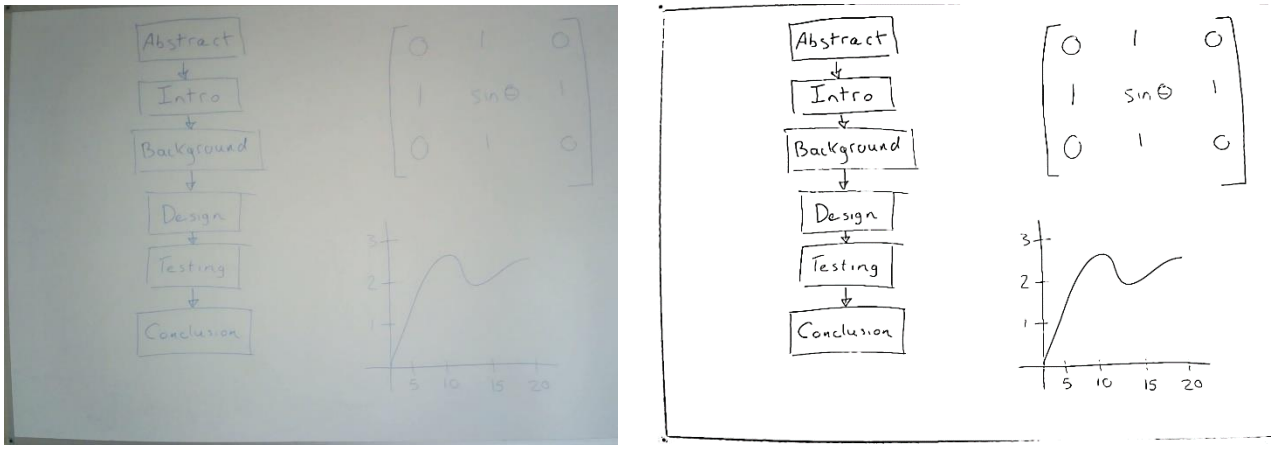
The lighting test is to investigate in the event of low light levels, how the capture system performs. In part (a) of fig. 38 the image shows the output of the system in optimal light levels, the content is clear and of high quality, this image is the reference image that part (b) and (c) are compared to.

Part (b) in fig. 38 shows the output of the system in low light levels directly before the natural light has left the room. The image contains some visual noise, this is due to the camera's inability to determine a pixels colour. While the noise is present in the image, the content on the whiteboard is still readable, but the overall image quality is poor.

In part (c) of fig. 38, the image is captured when there is no natural light in the room and a lamp is the sole source of light. The quality and distribution of the light from the lamp is poor but the system still outputs a noise free and high-quality image of the whiteboard. This image shows that the whiteboard doesn't need natural evenly distributed light and artificial light is sufficient to achieve a clear image of the content. Due to covid restrictions the testing area

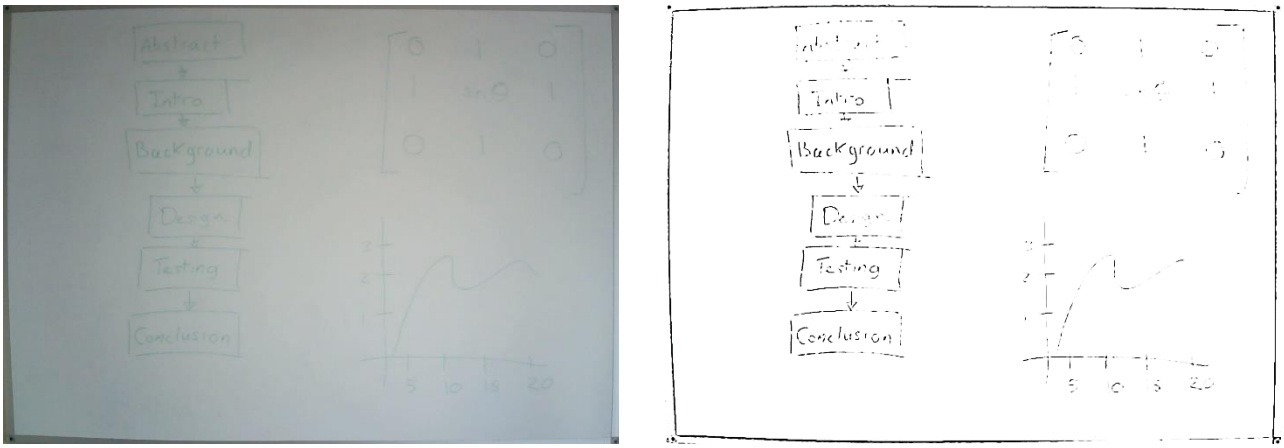
was limited to a bedroom but in the event that covid did not occur, testing would have been performed in a lecture hall where the artificial lighting conditions are optimal. Assuming the optimal conditions in the lecture hall, I believe the program would perform exceptionally and output the content in a clear manner.

8.4 Colour test



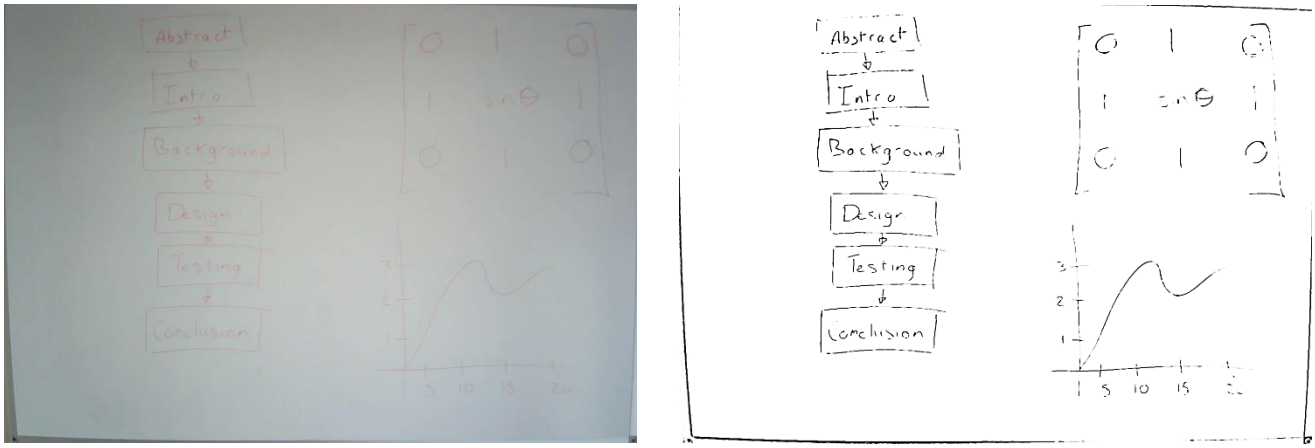
(a) (b)

Figure 39: Blue Colour Test (a) Original (b) Processed



(a) (b)

Figure 40: Green Colour Test (a) Original (b) Processed



(a) (b)

Figure 41: Red Colour Test (a) Original (b) Processed

When using different colour markers on the board, the most important factor is how much the colour contrasts with white. The darker the colour the more it contrasts with the whiteboard and the easier the camera will pick up on it.

In the above results blue is the colour that gives the highest contrast, therefore in fig. 39 you can see the output is clear and readable. The green and red markers were captured less effectively (see fig. 40-41) especially on the right-hand side of the board where the sunlight is more intense. This is due to the green and red markers having less contrast with the whiteboard.

To capture the green and red colour more effectively, changes can be made to the adaptive threshold to allow less contrast colours on the whiteboard be accepted. An issue with making the threshold less strict is that more noise will be accepted as seen in fig. 42.

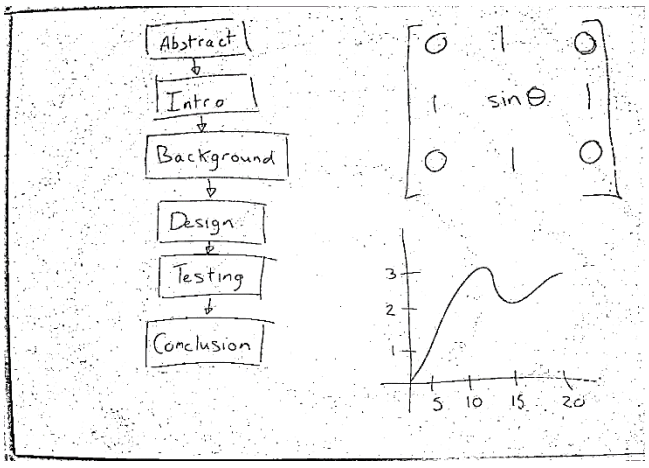


Figure 42: Adaptive Threshold Image

Since there is now more noise in the image the denoising filter must also be changed to account for this increase in noise. The simple solution to removing the noise is to increase the area in which the average for the filter is calculated. With the changes made to the adaptive threshold and filter, the output quality with the green marker increased as seen in fig. 43.

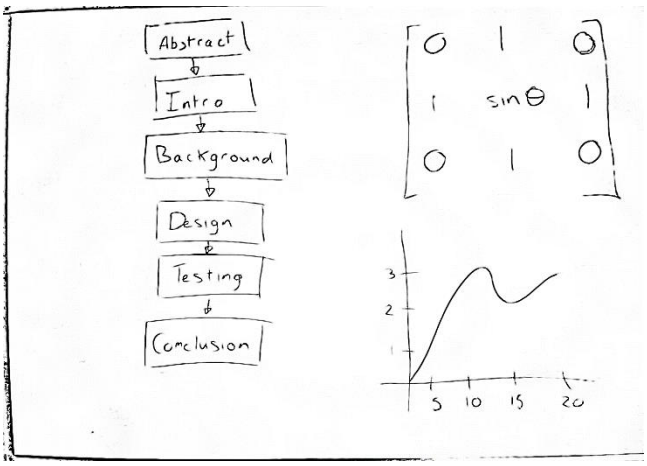
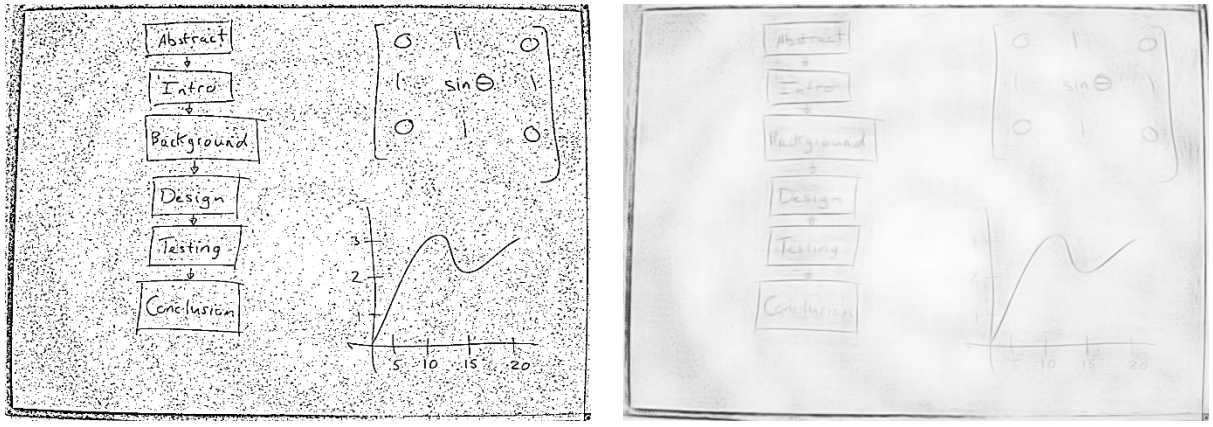


Figure 43: Filter image

The output for the green marker using the new arguments for the threshold and denoising filter are only effective at ideal lighting conditions, using these alterations in a low light or decreased light environment causes elevated levels of noise. This elevated level of noise is because the threshold is lower and the denoise filter cannot remove all the noise without decreasing the quality of the content.

When using the highest contrast colour, black, the output of the capture system has very destructive noise in low light levels that cannot be removed without severely blurring the content as seen in fig. 43.



(a) (b)

Figure 44: Altered Setting Results for Dull Lighting (a) Noise (b) Filter

From this test we can conclude that the green and red colours are not viable unless the program is altered and always used in ideal lighting. This program was made to be used in multiple lighting conditions and is effective at dealing with low levels of light as seen in test 3 (lighting test).

8.4 Other results

During the process of testing there was one major result yielded by accident, this occurred when contour detection was used on the whiteboard while content was present on the board. Due to the contour detection only happening on the prime image, this result was found late in the project cycle as the prime image was always captured with a blank whiteboard.

When the prime image contained a whiteboard with content present, the outputted contour image caught the contours of the shapes and characters on the whiteboard. This image of the contours of the contents (fig. 45) spurred the idea of comparing the number of contours on the whiteboard rather than comparing the grid cells against each other. This method can capture the contours of the content even at angles as shown in fig. 45, meaning the camera position could be changed at any time during the capture session. The sole purpose of the

camera not having the ability to move was due to the cells having to be in the same location on the whiteboard, if the camera is moved the cell would be capturing a different part of the board. If the cell method was not required and the contours of the shapes on the image are counted, the count can be compared for each frame and if the count changed considerably it can be assumed the content on the whiteboard has changed. Since the contours of the whiteboard can be captured at an angle, it can be determined the count will still be relatively accurate up to a 20-degree capture angle.

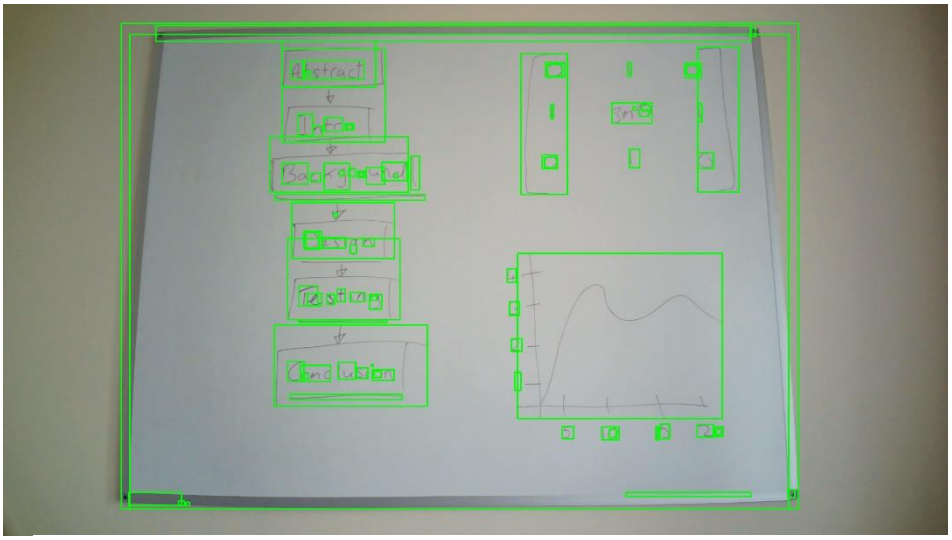


Figure 45: Contours of Whiteboard Content

Unfortunately, due to the late discovery of this method, the program could not be altered due to time constraints. This method would decrease computation time due to the simple fact that there are 1100 cell images saved every time a new frame of the whiteboard is captured. Instead of saving the 1100 images the program would save the number of contours found.

Overall, this method would benefit the program the most by eliminating the limitation of the camera having to remain in the same position throughout the capture session. It is realistic that the camera may fall over during the session and this would cause the program to fail require a reset.

9. Conclusion

The aim of this project was to design and implement an affordable whiteboard captured system that uses tools that are accessible. Throughout the project an affordable webcam was used which satisfies the main goal of design an affordable and accessible capture system .The capture system was successful at capturing the whiteboard at multiple distances and angles with this webcam. The first two aims of the project were reached, with light conditions at multiple levels and the position of the webcam with respect to distance and angle having minimal effect on the output quality.

The third aim was to achieve high quality output with different coloured markers, this goal conflicted with the first aim of the project. The coloured markers were not able to be effectively captured with the current arguments of the system due to the red and green markers nor having a high enough contrast with the whiteboard. The arguments of the system can be changed to effectively capture red and green markers but these new arguments hinder the ability of the system to capture in unideal light conditions. If the system assumes ideal lighting conditions then the system was effective at capturing all colour markers.

During the testing phase of the project an unexpected method of recognizing content on the whiteboard was discovered, this included using contour recognition to count the number of characters on the whiteboard and compare this count each time a frame is captured to pinpoint when a content event has occurred.

During the research phase the overwhelming majority of studies found used the grid method to recognize content events and this is the main reason the cell method was used. If the testing phase had not been so late in the project’s life cycle the contour method could have been implemented. This method of character recognition with contours is one of many additions that will be made to the capture system.

The list of future work includes creating an alternate program for 720p, design a graphic user interface for ease of use and formatting the pictures in PDF form as time constraints made it unachievable during the project.

Overall, the project was a success, the aims were achieved and the system outputted high quality images of the system in multiple environmental conditions.

References

[1] Amazon, "*Amazon webcams.*" amazon.co.uk. [Online]. Available: https://www.amazon.co.uk/s?k=webcams&i=electronics&ref=nb_sb_noss_1. [Accessed 10th January 2021]

[2] AHEAD, "*Universal Design for Learning.*" ahead.ie. https://ahead.ie/udl. (Accessed Jan. 28, 2021).

[3] Data Protection Commission. *Guidance for Data Controllers* (2019). https://www.dataprotection.ie/sites/default/files/uploads/2019-05/CCTV%20guidance%20data%20controllers_0.pdf. (Accessed Feb. 27, 2021).

[4] Kaptivo, "*Kaptivo Technology.*" kaptivo.com. https://kaptivo.com/technology/. (Accessed Jan. 24, 2021).

[5] Luidia, "*Ebeam Smartmarker.*" Luidia. https://www.luidia.com/smartmarker/. (Accessed Jan. 24, 2021).

[6] G. Thorsteinsson, "Piloting a Use of Graphic Tablets to Support," *Studies in Informatics and Control*, vol. 21, no. 2, pp. 201-208, 2012.

[7] R. Y. da Xu, "A Computer Vision based Whiteboard Capture System," *2008 IEEE Workshop on Applications of Computer Vision*, Copper Mountain, CO, USA, 2008, pp. 1-6

[8] A. Fakih, "Real-Time Whiteboard Capture System," *Journal of Electrical and Electronics Engineering*, vol. 2, no. 1, pp. 21-25, 2010.

[9] Z. Zhang, "Whiteboard scanning and image enhancement," *Digital Signal Processing*, vol. 17, no. 2, pp. 414-432, 2007.

[10] L.-W. He, "Why take notes? Use the whiteboard capture system," Microsoft, Redmond, WA, USA. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2002-89.pdf

[11] R. Gonzalez, "Image Segmentation," in *Digital Image Processing*, New Jersey: Prentice Hall, 2001, ch 1, pp 689-794.

[12] Doxygen, "*OpenCV.*" opencv.org. https://docs.opencv.org/4.5.1/.(Accessed Feb. 7, 2021).

[13] A. Mordvintsev, "OpenCv-Python-Tutorials." opencv-python-tutroals.readthedocs.io. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_tutorials.html. (Accessed Feb. 7, 2021).

[14] G. Shrivakshan, "A Comparison of various Edge Detection Techniques used in Image Processing," *International Journal of Computer Science Issues*, vol. 9, no. 5, pp. 269-276, 2012.

[15] P. Patidar, "Image De-noising by Various Filters for Different Noise," *International Journal of Computer Applications*, vol. 9, no. 4, 2010.

[16] R. Fisher, "Image Processing Learning Resources." homepages.inf.ed.ac.uk. https://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm. (Accessed Feb. 15, 2021).

Appendix

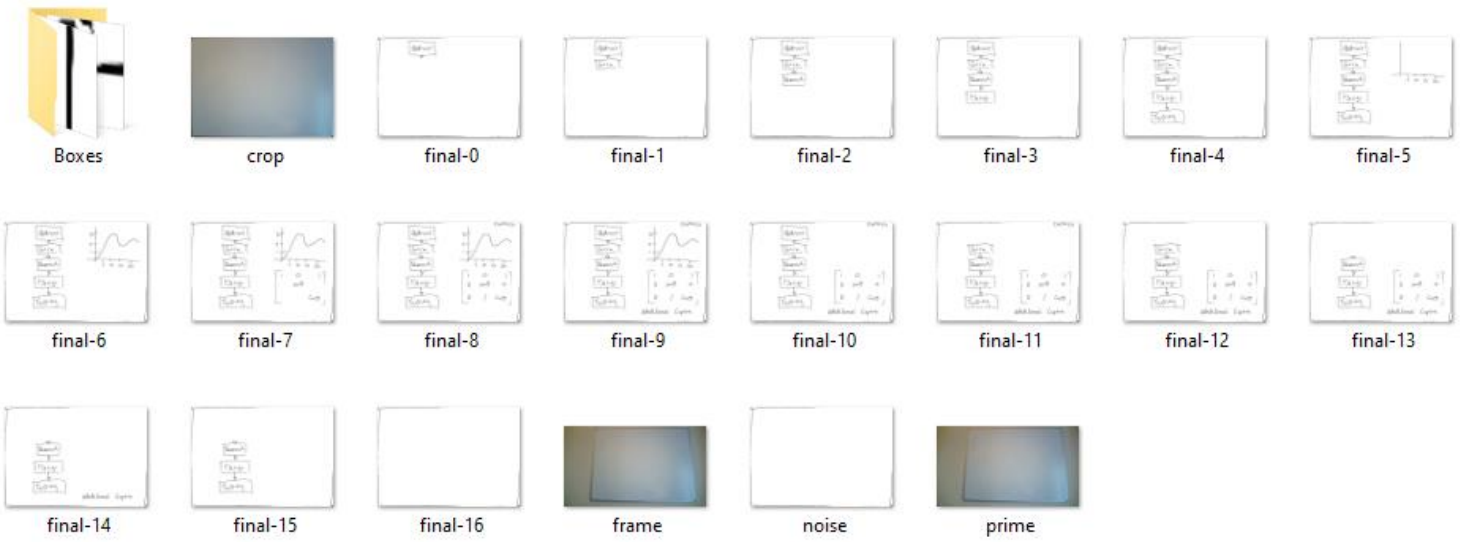


Figure 46: Output File

Full Program Code

```
import cv2
import numpy as np
import time

path = 'C:/Users/the dogs bollox/Desktop/Images/'        #path for basic image
boxpath = 'C:/Users/the dogs bollox/Desktop/Images/Boxes/'  #path for grid images
print("Package Imported")
start = 0                                  #flag for first run through program
x, y, w, h = 0, 0, 0, 0                   #coordinates and measurements for board
foregrnd = 0                              #flag for foreground object
count = 0                                 #count for how many timeline images
board = []                                  #the contour of the board

def VideoCap():
    cap = cv2.VideoCapture(0)                   #set capture device
    cap.set(3, 1920)                            #set quality to 1080p
    cap.set(4, 1080)

    while True:                             #run look until 'q' key is pressed
        success, vid = cap.read()
        cv2.imshow("Video", vid)             #display video window of capture device
        if cv2.waitKey(1) & 0xFF == ord('q'):
            cv2.destroyWindow('Video')
            break
    frame = cap.read()[1]                    #capture frame from video
    if start == 0:                          #if first run through of program
        cv2.imwrite(str(path) + 'prime.png', frame) #set frame as prime        foregrnd = 0
#set foreground object to 0
    else:
        cv2.imwrite(str(path) + 'frame.png', frame)  # read image frame
        prime = cv2.imread(str(path) + 'prime.png', 1)     #read in prime image
        sub = cv2.subtract(prime,frame)                 #subtract to get foreground
        subgrey = cv2.cvtColor(sub, cv2.COLOR_BGR2GRAY) #make image greyscale
        ret, thrsh = cv2.threshold(subgrey, 75, 255,
                                   cv2.THRESH_BINARY)  #apply threshold
        if cv2.countNonZero(thrsh) > 1000:     #counts the number of white pixels
```

```
            foregrnd = 1              #if count is above 1000, foreground flag set
        else:
            foregrnd = 0                            #else foreground not set
    return frame, foregrnd            #return frame image and foreground

def ContourDet(frame, start, x, y, w, h, board):
    imgBlur = cv2.bilateralFilter(frame, 7, 15, 15)    #denoising image
    imgCanny = cv2.Canny(imgBlur, 10, 100)             #apply canny filter
    kernel = np.ones((5, 5), np.uint8)                #set kernel/structuring size
    imgDil = cv2.morphologyEx(imgCanny, cv2.MORPH_CLOSE, kernel)#dilate canny image
    contours, hierarchy = cv2.findContours(imgDil, 1, 2)  #find contours/shapes in
    area = 0                                          #area variable set to 0
    if start == 0:                                 #if first run through of program
        for cnt in contours:             #for loop going through every contour found
            x, y, w, h = cv2.boundingRect(cnt)  #(x,y) is the coordinate of top left corner
            imgCont = cv2.rectangle(imgBlur, (x, y),  #draw all contours
                                    (x + w, y + h), (0, 255, 0), 2)
            if area < (w * h):                 # compares to previous contour area
                area = w * h                        #sets the area to biggest area
                board = cnt                #biggest contours coordinates are saved
        x, y, w, h = cv2.boundingRect(board)
        cv2.imwrite(str(path) + 'contours.png', imgCont)
    peri = cv2.arcLength(board, True)           #finds the perimeter of contours,
    corners = cv2.approxPolyDP(board, 0.06*peri, True)    #finds coordinates corners
    n = corners.ravel()                       #takes coordinates out of embedded array
    random = ([n[0],n[1]],[n[2],n[3]],[n[4],n[5]],[n[6],n[7]],) #saves coordinates into array
    sort1 = sorted(random, key=lambda x: x[0])        #sorts by x coordinate
    lcorners = (sort1[0], sort1[1])#saves largest two x cords in array, two left
    rcorners = (sort1[2], sort1[3]) #saves smallest two x cords in array, two right
    leftsort = sorted(lcorners, key=lambda x: x[1])    #sorts to get top corner
    rightsort = sorted(rcorners, key=lambda x: x[1])
    print(leftsort, rightsort)                                  #print two arrays
    pts1 = np.float32([leftsort[0], rightsort[0],
                       rightsort[1], leftsort[1]]) #array starting top left going clockwise
    pts2 = np.float32([[0, 0], [w, 0], [w, h], [0, h]]) #desired corners
    M = cv2.getPerspectiveTransform(pts1, pts2)     #creates transformation matrix
    imgCrop = cv2.warpPerspective(frame, M, (w, h)) #applies transformation matrix
    cv2.imwrite(str(path) + 'crop.png', imgCrop)
    return x, y, w, h, board          #returning cords and contour of board


def cleanimage():
    img = cv2.imread(str(path) + 'crop.png', 0)
    imgBW = cv2.adaptiveThreshold(img, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                  cv2.THRESH_BINARY, 21, 5)
    noise = cv2.fastNlMeansDenoising(imgBW, None, 50, 7, 51)   #removing noise
    cv2.imwrite(str(path) + 'noise.png', noise)
    return


def gridimage(start, change, nochange, count):
    noise = cv2.imread(str(path) + 'noise.png', 0)
    ni = np.array(noise)                          #creating array of denoise image
    height, width = ni.shape[:2]
    boxesH = 25                                      #setting grid cell height
    boxesW = 44                                      #setting grid cell width

    for i in range(boxesW):                     #for loop count up to grid width
        for x in range(boxesH):                 #for loop count up to grid height
            this = ni[x * height // boxesH:(x + 1) * height // boxesH, #crops cells
                   i * width // boxesW:(i + 1) * width // boxesW]
            if start == 0:                      #if first run through just save image
                cv2.imwrite(f'{str(boxpath)}box-{i}-{x}.png', this)
            else:                                                    #else
                this_array = np.array(this)
                prev = cv2.imread(f'{str(boxpath)}box-{i}-{x}.png', 0)
                prev_array = np.array(prev)
                if abs(np.mean(this_array) - np.mean(prev_array)) >= 20:
                    change += 1
                else:                   #comparing mean colour of cells
                    nochange += 1
                cv2.imwrite(str(boxpath) + f'box-{i}-{x}.png', this)#save box image
    start += 1                             #no longer first run so increment start
    if change > 5:                        #if more than 5 grids have changed
        cv2.imwrite(str(path) + f'final-{count}.png', noise)  #save img to timeline
        count +=1
    return start, count
```

```
try:
    while True:
        frame, foregrnd = VideoCap()
        if foregrnd == 0:
            x, y, w, h, board = ContourDet(frame, start, x, y, w, h, board)
            cleanimage()
            start, count = gridimage(start, 0, 0, count)
        else:
            print('Foreground Object')
        time.sleep(2)
except KeyboardInterrupt:
    pass
```