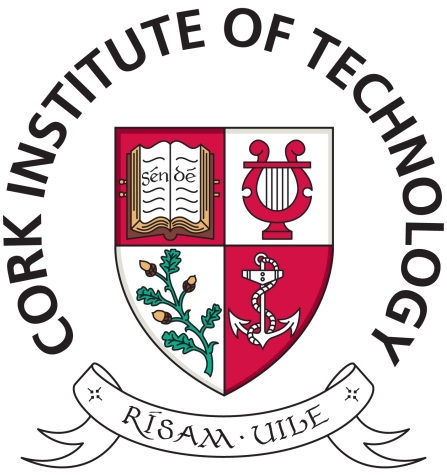


Stairs Detection and Step Segmentation Through Augmented Reality as Visual Impairment Aid

by

Marcus Wong Yew Hon

This thesis has been submitted in partial fulfillment for the degree of Master of Science in Artificial Intelligence

in the
Faculty of Engineering and Science
Department of Computer Science

September 2020

Declaration of Authorship

This report, Stairs Detection and Step Segmentation Through Augmented Reality as Visual Impairment Aid, is submitted in partial fulfillment of the requirements of Master of Science in Artificial Intelligence at Cork Institute of Technology. I, Marcus Wong Yew Hon, declare that this thesis titled, Stairs Detection and Step Segmentation Through Augmented Reality as Visual Impairment Aidand the work represents substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged. I confirm that:

- This work was done wholly or mainly while in candidature Master of Science in Artificial Intelligence at Cork Institute of Technology.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Cork Institute of Technology or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:  Marcus Wong Yew Hon

Date: 17th May 2020

CORK INSTITUTE OF TECHNOLOGY

Abstract

Faculty of Engineering and Science
Department of Computer Science

Master of Science

by Marcus Wong Yew Hon

Augmented reality devices have been gaining momentum in recent years especially in the area of assistive technology, particularly targeted towards assisting visually impaired individuals in navigating around unknown indoor areas. With visual information being a key factor in efficient navigational tasks, visually impaired individuals are in dire need of an intuitive and accessible solution to improve their visual senses. This research will be focused upon leveraging existing Augmented Reality (AR) technologies such as the Microsoft HoloLens to improve object visibility and distinguish-ability for users suffering from Glaucoma in indoor areas of low color contrast, with stairs being the main object of focus in this project. This was achieved by using deep learning based object detection models such as YOLO (You Only Look Once) for stairs detection and Probabilistic Hough Line Transform for stairs step counting.

Acknowledgements

I would like to thank Dr. Sean McSweeney, supervisor of this project, for providing me with guidance regarding implementation issues during development of this project, as well as helpful feedback for the completion of this paper. Your patience with me throughout this semester was truly much appreciated. Huge thanks to Dr. Kevin O'Mahony from Nimbus Research Center as well for organizing this research project and providing me with insightful context in regards to the motivation and general direction for this research. I would also like to thank Dr Victor Cionca, for introducing me to the Nimbus Research Center and giving me the opportunity to apply there in light of research opportunities for this module.

Lastly, I would like to extend my sincere gratitude to Cork Institute of Technology for offering this module that allowed me to be apart of this research project in the first place.

Contents

List of Figures

List of Tables

Abbreviations

AR	Augmented Reality
SLAM	Simultaneous Localization and Mapping
CNN	Convolutional Neural Networks
SIFT	Scale Invariant Feature Transform
SURF	Sped Up Robust Features
RGB	Red, Green, Blue color channel
AMD	Age-related Macular Degeneration
RCNN	Regional Convolutional Neural Networks
YOLO	You Only Look Once
HOG	Histogram of Oriented Gradients
mAP	Mean Average Precision
AP	Average Precision
ROI	Region of Interest
RPN	Region Proposal Network
FPS	Frames Per Second
HSV	Hue, Saturation and Value
GPU	Graphics Processing Unit
CC	Compute Capability
UWP	Universal Windows Platform
DLL	Dynamic-link Library
IoU	Intersection Over Union
TP	True Positive
FP	False Positive
FN	False Negative

This thesis is dedicated to my family for their endless support, love and encouragement during the entirety of this course.

Chapter 1

Introduction

Visual impairment can be defined as a partial or significant loss of the ability to process visual information due to factors such as physical injuries to the anatomy of the eye, diseases and genetic predispositions, whereby conventional methods of vision correction such as glasses prescription are not feasible or even possible [1]. There are various forms of visual impairments, the most common of them being glaucoma, where vision loss occurs due to accrued damage to the optic nerve, and color blindness. Visual impairments among people are increasing by the year, with official estimates from the World Health Organization (WHO) putting people suffering from forms of visual impairments at 285 million as of 2014 [2].

The increasing rate of the visually impaired worldwide is worrisome as the loss of vision in a person could have their quality of life be severely impacted. This is because severe visual impairments could hinder one's ability to perform basic daily tasks such as navigation and object recognition. In recent decades, various real world architecture and design considerations catered to improving indoor navigation and accessibility for the visually impaired such as high contrast way finding lines in corridors, color coordinated toilet facilities and tactile friendly signboards/buttons have been implemented. However, the vast majority of indoor establishments around the world have yet to truly embrace the visual disability friendly design considerations due to the additional cost and planning required for installation. Therefore, a more flexible, feasible and accessible solution to accommodate the visually impaired in indoor environments have to be done, which is where Augmented Reality (AR) and computer vision comes in.

Augmented Reality is an interactive technology which allows for digital information or graphics to blend into real world surroundings, and is usually facilitated through a wearable head-wear device that overlays the digital information on to the visor of the device, which creates the perception of digital elements being integrated into reality.

With that in mind, it is then possible to relay virtual visual corrections into the real-world perception of a visually impaired person, thus mitigating the exorbitant cost and meticulous planning required which deterred construction companies from implementing real world design considerations for people with disabilities. Visual information obtained from the sensors of the AR head-wear device can be utilized to process and perform computer vision operations such as simultaneous localization and mapping (SLAM), which utilizes visual information to generate maps and attain tracking information for objects within the field of view of the device. This can be leveraged for object detection operations for this project.

Since the research will be focused on object detection for the visually impaired through AR technologies, the Microsoft HoloLens will be the targeted hardware for implementation in this research. This is because the Microsoft HoloLens provides better research and software tinkering support out of the box with its "research mode", which allows for direct access to the raw data captured by the on-board sensors of the device [3]. This allows for the live video feedback to be processed with object detection and feature accentuation before being projected back to the field of vision of the user. The project would be implemented using C# and the OpenCV image processing library would be used to compile the object detection and color correction modules of the project. Object detection techniques and algorithms such as disparity segmentation, contours extraction and even deep learning approaches to object classification such as YOLO (you only look once) would be considered for our object detection module. Edge detection techniques such as Sobel edge detection would be used in tandem with object detection in order to build a basic object detection module. Gamma correction would be applied for detected objects to amplify and accentuate the color contrast of certain objects to create a more distinctive indoor environment for visually impaired people.

As there are simply too many forms of visual impairments which requires different considerations and implementations to be catered, glaucoma will be the main area of focus as far as visual impairment types are considered. This is because Glaucoma is one the most common visual impairments in the world, accounting 2.2% of all patients suffering from moderate to severe visual conditions [4]. The project aims to heighten the awareness of users' surroundings, particularly users suffering from glaucoma, in indoor environments, where various obstacles or objects such as tables, chairs and stairs might impede or obstruct user navigation. For our case, the object detection will first be limited to stairs only.

1.1 Motivation

Sufferers of glaucoma encounter severe impediments in physical navigation, especially in indoor establishments. Glaucoma is a group of ocular diseases, where the most common variant of glaucoma being open-angled glaucoma, accounting for almost 90% of all glaucoma cases [5]. Open angled glaucoma is caused by the clogging of the drainage canals within the eye, slowing down the flushing of eye fluid, hence increasing internal eye pressure [5]. Sufferers of open angle glaucoma report visual symptoms such as tunnel vision, color desaturation and diminished brightness in their sight [6], as depicted in Figure 1.1. These symptoms contribute to the difficulties of indoor navigation, where everyday objects (such as stairs) are considered an obstacle or even hazards due to the confined area of movement.

With stairs, glaucoma patients might not have issues with identifying the existence of stairs within their line of sight, but they might have trouble distinguishing the edges of steps in stairs. With that in mind, a step counting algorithm will also be incorporated within our proposed system to inform users of the number of steps in a flight of detected stairs, in hopes of giving users more real world context when encountering a flight of stairs.

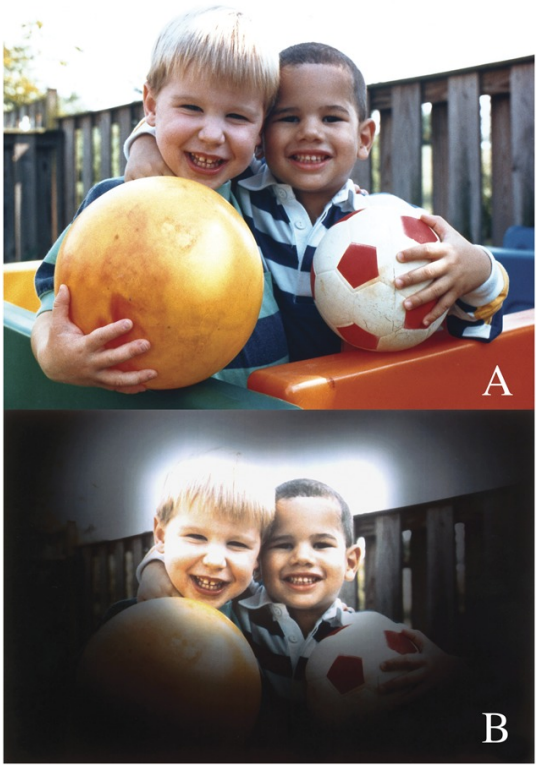


FIGURE 1.1: Normal Vision (A), What Glaucoma Patients See (B)

Therefore, with the advancements of AR devices, coupled with its ability for contextual awareness, it is imperative that we make use of AR to develop a system using the Microsoft Hololens for stairs detection, step counting and visual feedback enhancements through color accentuation to alleviate difficulties of glaucoma patients in traversing through environments with stairs involved.

1.2 Contribution

This project aims to alleviate navigational difficulties of glaucoma users in indoor environments by providing contextual information, like the existence of potentially hazardous obstacles, such as stairs, and complementing the detection with step count information as well. The project managed to achieve a deployable HoloLens system that is able to detect stairs while providing the step count during traversal with a reasonable rate of accuracy for both stairs detection and step counting operations.

1.3 Structure of This Document

In this report, Chapter 2 describes the relevant research papers and state of the art in regards to object detection through AR, stairs counting, image classification algorithms and color alteration. Chapter 3 describes the general thoughts and considerations that went through while planning the implementation of the project, which includes problem description, requirements, architecture design choices, choices of algorithms, as well as tools and software used. A detailed report of the implementation of the project was described in Chapter 4, along with the challenges faced and solutions presented to combat said difficulties for each implementation phase in the project. Chapter 5 presents the final results of the implemented stairs detection module and the step counting algorithm while an objective evaluation on the whole project is done at the end. Lastly, Chapter 6 details the explanation for certain aspects of the results presented in Chapter 5 and a subjective evaluation on the whole project was done. Possible improvements and future implementation decisions were also described in this chapter.

Chapter 2

Literature Review

While there are not many research papers or currently existing solutions that describes real time object detection and feature accentuation on Augmented Reality devices for the visually impaired, specifically glaucoma patients; there are however, relevant papers in regards of the components of this research topic that can be combined, studied and applied to address the problem statement of our research, which includes object detection, step segmentation and color modification/accentuation.

This section will detail the approaches of each paper in achieving its objective in its respective subsection, as well as the drawbacks of the methods observed in the context of our research topic. Object detection will be covered in two distinct subsections, first being object segmentation in augmented reality, and another being object classification in images.

2.1 Object Detection

Object segmentation is inherently different from object classification. While object segmentation is a process of distinguishing between objects in an image by means of edge separation, disparity differences or foreground/background extraction, it could not classify or determine the type of object it segments. Object classification in images on the other hand, allows for detection and classification of objects. Image object classification are usually performed through the usage and training of deep learning frameworks such as convolutional neural networks (CNN), whereas object segmentation is usually done using algorithms involving pixel manipulation and analysis.

2.1.1 Object Segmentation in Augmented Reality

Several approaches to object segmentation in augmented reality have been proposed. One of them includes using a disparity depth approach by Costa *et al.*[7] to detect the depth information of the visual information obtained from the augmented reality camera sensors. This approach is effective in obtaining the relative distance or depth of multiple objects within an environment. The distinction of various objects based on its depth could be useful for segmenting objects in an indoor environment. Wang *et al.*[8] also implemented a depth based object recognition approach albeit in a different method from [7], where instead of obtaining depth features from a disparity image, [8] obtains depth information from calculating the score of the detected ground level plane with the points from a point vector space. The scores are then used to classify objects from a set of already known objects. The approach used by [8] is not as robust as the depth approach proposed by [7] as the reliance of a set of known objects in [8] means a higher possibility of miss-classification or detection failure rates for arbitrary indoor objects.

A contour based approach was proposed by Younis *et al.*[9] whereby foreground and background segmentation was implemented in order to detect moving objects. However, further enhancement was done by [9] to allow for contouring to work when motion was introduced into the video feed as contour based object detection assumes the video to be stationary. To tackle this issue, key points of the current frame and previous frames were determined and using a homography matrix, the distances were translated and calculated for each subsequent frames of the key points in the video. This approach might work suitably well for cases of motion tracking and object movements, but when presented to environments (indoor in our case) with stationary objects, the effectiveness of the approach remains questionable.

Feature extraction based object segmentation approaches such as SIFT (Scale-invariant feature transform) and SURF (Sped-Up Robust Features) were explored by [10] [11] and [12]. The SIFT feature extraction generally works by identifying key points of a targeted object within an image and classifying the features against a database of training object images to its belonging object class through the nearest neighbour heuristic using the calculated euclidean distance [13]. SURF on the other hand relies on detected key points and integral images for convolutional processing of images to detect and classify objects. According to a survey research done in [13], SURF techniques performs significantly better in terms of accuracy, efficiency and performance as compared to SIFT. Also, unlike SIFT, SURF is invariant to rotation and scaling[13]. However, both approaches suffer from being unable to process or recognize color information, which might present a problem for augmented reality devices such as the Microsoft HoloLens which captures images or videos in RGB.

Lastly, an edge based object segmentation was proposed by Hwang *et al.*[14]. The Laplacian Edge Detection algorithm was implemented in [14] for their Google Glasses augmented reality device to great results in terms of edge segmentation for indoor environments. By applying two types of Laplacian edge detection algorithms, positive and negative, Hwang *et al.*[14] was able to allow up to three levels of edge enhancement intensity settings for users suffering from AMD (age-related macular degeneration). The edge segmentation approach used could be useful for further enhancing object distinction in an indoor environment after object segmentation have been applied. The ability of varying intensity settings by combining various algorithms of similar approaches could be simulated for our current research as well.

2.1.2 Object Classification

As mentioned earlier, object classification methods typically refers to deep learning models, specifically convolutional neural networks (CNNs), that are specially catered for image classification. Deep learning has seen a huge resurgence in research interest the past decade due to the success of convolutional networks in image classification operations. This subsection will detail two state of the art deep learning object classification model families, namely RCNNs (Region Convolutional Neural Networks) and YOLO (You Only Look Once).

2.1.2.1 R-CNNs

In 2013, Girshick *et al.*[15] have proposed a novel approach to object classification and detection for its time with R-CNNs. Prior to this, the best performing neural networks in image classification involves complex systems and configurations in the form of ensembles, usually based on tried and true methods such as SIFT and HOG (Histogram of Oriented Gradients). However, R-CNNs are not only much simpler in complexity and higher in scalability, but also showed significant improvements in accuracy. Test results on RCNNs showed a 30% increase in mAP (mean average precision) score on the VOC2012 data set compared to the previous best mAP score, with an mAP of 53.3%.

RCNNs are composed of three modules. The first module generates region candidates within an image that are not category dependent. Selective search was the main approach implemented for the region candidate selection module, where regions of interest are selected by grouping similar characteristics, such as color, texture, shape patterns and sizes [16]. The grouping of said characteristics would generate bounding boxes around detected characteristic similarities, segmenting regions in the process. As this

would cause over-segmentation, further processing is done by merging adjacent bounding boxes, again, by metrics of similarity. The merging process will be repeated, with gradually larger merged bounding box regions be added to the list of region candidate proposals to be returned.

The second module is the feature extraction module. This module utilizes Krizhevsky's *et al.*[17] AlexNet convolutional network layer to obtain predicted image features. The features are propagated through 5 convolutional layers and two fully connected dense layer with 4096 neurons each with input RGB images first resized to dimensions of 227x227, regardless of aspect ratio [15]. The last module is the classifier, using sets of linear Support Vector Machines (SVMs) to classify the proposed regions to its belonging class. Figure 2.1 illustrates the overall model architecture of RCNNs with its three modules for predicting object classes within images.

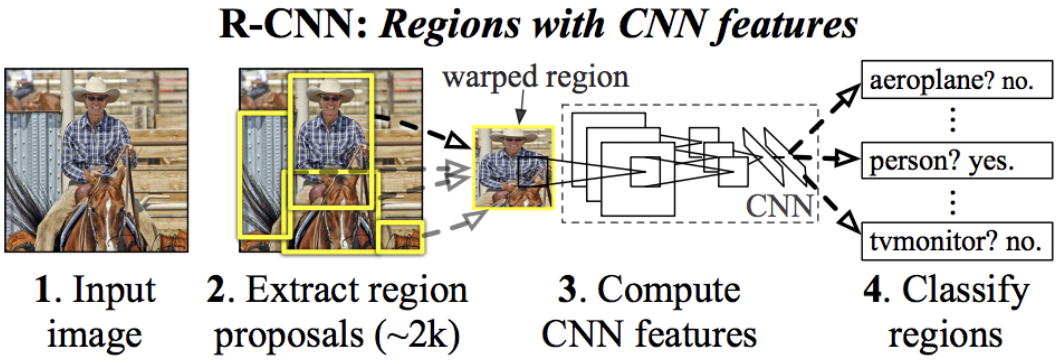


FIGURE 2.1: Model Architecture Visualisation for RCNNs

However, the issue with the RCNN approach is that it is slower in classifying objects within an image, as it requires for all candidate regions proposed in the first module to be forward passed into the CNN layer for feature extraction before a classification can be made. This can be seen when [15] describes an average of roughly 2000 proposal regions were generated by the model per image during testing. This issue will soon be addressed in [18], known as Fast RCNN, a successor to the original RCNN. [18] not only addressed the speed issue encountered in RCNNs, but also brought up other limitations such as the multi pipeline approach used in RCNNs with its three modules, and the slow training process due to the use of CNNs for feature extraction on the large number of proposed candidate regions.

Fast RCNNs now use a single staged approach instead of the multi pipelined approach in RCNN for detecting and classifying candidate regions. This was achieved by implementing a ROI (region of interest) pooling layer at the end of a pre-trained feature extraction convolutional network layer (VGG16). The ROI pooling layer extracts a fixed

feature vector from the output of the feature extractor, corresponding to its specific input candidate region. Then, the extracted feature vectors from the ROI pooling layer are propagated through a sequence of densely connected layers, just like RCNNs. But, instead of piping the extracted features through a set of linear SVMs for classification output, the fully connected dense layers in Fast RCNNs branch into two output layers: a softmax layer for classification output, and another layer for bounding box output [18]. This resulted in a more streamlined training and classification process for the model, leading to state of the art performance at the time with improved detection quality, achieving an mAP of 66.9% on larger data sets, and faster detection time.

In 2015, Shaoqing *et al.*[19] have further enhanced the RCNN family of models with Faster-RCNN, the latest incarnation of RCNN as of writing. Again, following the trend of previous version of RCNN, the Fast RCNN, Faster RCNN is built on a single unified network for obejct classification and detection. Contained within this unified network are two modules, a CNN network module and the existing Fast CNN detector module from [18]. This version of the RCNN model further improved the speed of training and classification by proposing the use of a Regional Proposal Network (RPN) as the aforementioned CNN module. It is a novel solution that provides nearly cost free region proposal computation. The RPN aims to unify and refine the region proposal process during training phase of the model by sharing convolutional layers in the object detection model. Figure 2.2 depicts the new model architecture featuring the new RPN layer.

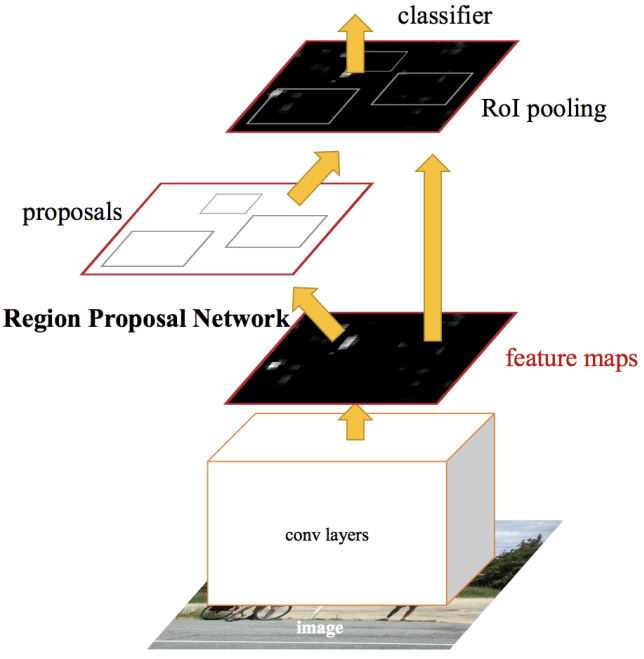


FIGURE 2.2: Model Architecture Visualisation for Faster RCNNs

Being a shared convolutional layer, the RPN convolves a miniature network over the convolutional feature map output by its preceding shared convolutional layer. A n x n sized sliding window of the convolutional feature map was fed as input to the miniature network and the output is then mapped to its next shared convolutional feature of lower dimension. The resulting feature from this will then be two child fully connected dense layer: the box regression layer and the box classification layer [19]. Each sliding window location in the RPN will simultaneously perform multiple region proposals (denoted by k). The k will be referenced to what is known as *anchors*. It is important to note that the resulting anchors are translation invariant and scale robust. Figure 2.3 illustrates the sliding process of RPNs and its resulting anchors.

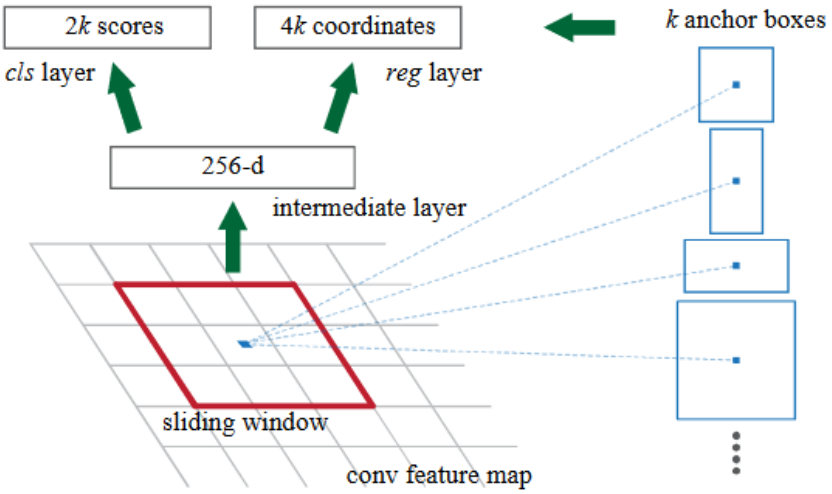


FIGURE 2.3: Sliding Process Output of Regional Proposal Network (RPN)

With RPNs, [19] reports that Faster-RCNNs now achieves near real time object detection performance of 5 fps (frames per second) on a K40 GPU without sacrificing accuracy, achieving an mAP of 75.9% on the VOC2007, VOC2012 and MSCOCO union data sets.

2.1.2.2 YOLO

You Only Look Once (YOLO) is an object detection and classification approach proposed by Redmon *et al.*in [20] that, similar to RCNNs, utilizes bounding boxes to frame objects within an image. However, unlike RCNNs, which uses selective search for obtaining bounding boxes, YOLO frames object bounding boxes and its associated probability of classes as a regression problem when separating them spatially.

Also, unlike RCNNs, YOLO performs its object classification and detection through only one convolutional network to predict its bounding boxes and class probabilities as opposed to the multi pipeline approach found in [15]. YOLO does so in only one

evaluation, hence the origin of its name "You Only Look Once", and the contributing characteristic to its extremely fast detection rate. [20] reports that the single unified architecture of YOLO manages to perform image recognition at a rate of an impressive 45 fps with its base model, and an even more staggering rate of up to 155 fps for the smaller network version of YOLO.

The heuristics for YOLO's approach in object detection is relatively simple. First, the image will be divided into $n \; x \; n$ grids. Each grid cell will then be responsible in predicting a bounding box as well as the confidence score of an object should it fall within the grid cell. Confidence score refers to the confidence of the model on whether it detects an object or not as well as the accuracy of its respective bounding box. Each grid cell also has the class probability of its detected object associated with it, and is limited to only one class probability set. This gives the model class specific confidence scores for each bounding box it predicts. Figure 2.4 illustrates the prediction process of YOLO with its grid cells approach.

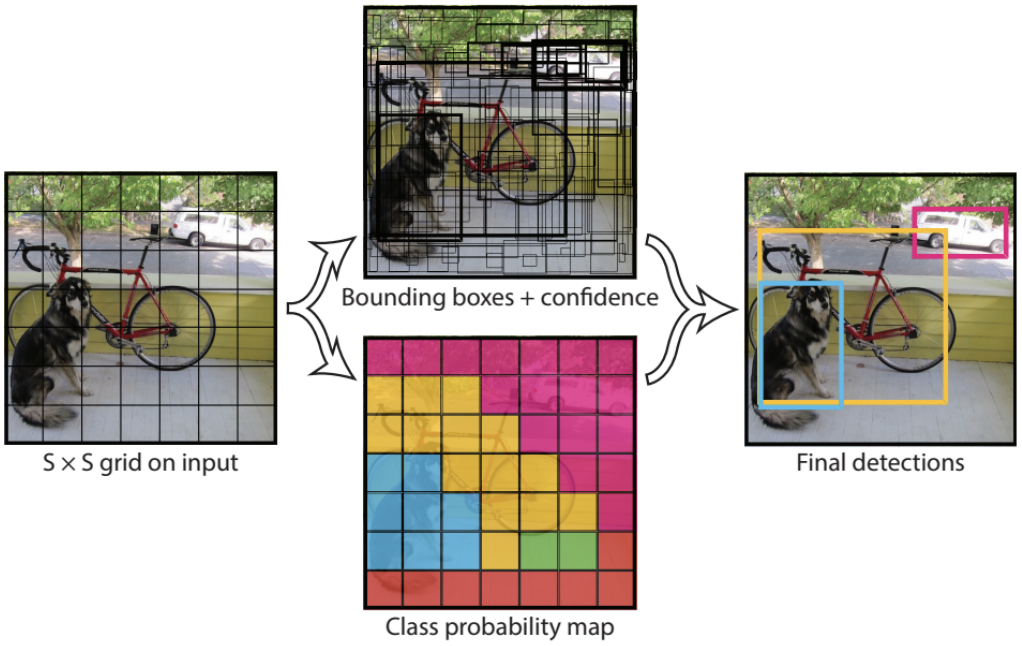


FIGURE 2.4: Flow of Prediction Process of YOLO

With the grid cells approach, YOLO models are more likely to make localization errors as compared to other object detection models such as RCNNs but are less likely to make false positive predictions for objects in the background of images [20]. This trade off of accuracy however was compensated by its astounding speed and it still presents a highly valued solution to real time object detection as it manages to outperform other real time object detection models of its time in accuracy, almost double the mAP score.

In 2016, Redmon *et al.*[21] presented the second iteration of its YOLO model, known as YOLO V2 but dubbed the YOLO9000, due to its ability to classify up to 9000 object classes. The model addressed the glaring issue in the first incarnation of YOLO, its accuracy. In YOLO v2, the model has achieved an mAP of 78.6% mAP whilst still running at 40 fps, giving accuracy exceeding that of even Faster-RCNNs while providing more than adequate real time performance.

It achieves this by bringing a few changes to the existing structure and implementation in the first incarnation of YOLO, most notably with the introduction of anchor boxes as seen in [19], batch normalization, high resolution classifiers and multi-scale training [21].

Anchor boxes implementation in YOLO v2 differs slightly from [19], where fully connected layers in YOLO were stripped off and the anchor boxes were used solely for predicting bounding boxes. The new YOLO model also differs slightly Faster-RCNNs in a way where class probability prediction responsibilities no longer rests on each spatial location as seen in Fig 2.3, but instead shifts class and object prediction tasks to each anchor box. The use of anchor boxes has led to a slight decrease in accuracy from 69.5 mAP without anchor boxes to 69.2 mAP with anchor boxes, but showed an increase of 7% in recall rate to 88%. [21]

Batch normalization in YOLO v2 has allowed the model to forgo normal regularization process while still yielding great improvements in convergence during training. The use of batch normalization also increased the mAP by 2% and provided the model the flexibility to remove dropout layers without an increase in overfitting [21]. This is because normalizing the input image batches acts also as a form of regularization to the model.

In [20], the feature extraction process was based upon the convolutional network found in [17], which receives input images with resolutions of 256 x 256. [20] then takes a 224 x 224 resolution input image when training and up-scales them to 448 x 448 when detecting. This has now changed in YOLO v2, where the training process initializes with a 448 x 448 resolution input image training phase for 10 epochs, giving time for the network to adjust itself for high resolution detection [21]. This change has brought a gain of almost 4% in mAP.

For multi-scale training, YOLO v2 picks a random resolution with factors of 32 for training for each 10 training image batches. The resolution scales goes from 320 x 320 resolution at its smallest to 608 x 608 at its highest. This allows the network detection to be robust in image sizes and resolution while also capitalizing on the speed gains when training on smaller resolution image batches. According to [21], the model detects

at a rate of 90 fps with low resolution inputs while still showing similar mAP scores to [18]. The model is a state of the art detector when given high resolution inputs while still showing performance that exceeds real time speeds (40 fps).

Lastly, YOLO v3 served as a minor update that was released in 2018 with minor improvements overall to YOLO v2 with small network configuration changes applied. However, the new changes to the model has led to a very slight decrease in speed as compared to YOLO v2 due to its larger network structure [22], but yet manages to improve its accuracy to an AP score of 57.9 while still providing above real time performance (51 milliseconds on the Titan X) as compared to other state of the art object detectors such as RetinaNet (57.5 AP at 198 milliseconds).

2.2 Stairs Segmentation

There are two main approaches when it comes to stairs segmentation. In earlier days of stairs segmentation, pixel manipulation with image filters were used in tandem with line detection algorithms such as Hough Transform to isolate and group defining edges found in stairs as a means of detection. [23] spearheaded the above approach with the use of Gabor texture filters to detect distinctions between textures in an image. The resulting difference in texture patterns can then be used to derive the edges within an image that suits their detection needs. For [23] and [24], Gabor filters with specific frequencies must first be created before being convolved with the input stairs image to yield desired stair edges. The resulting image was then subjected to thresholding to remove noise and unwanted responses before edge selection of desired length can take place. Stephen Se *et al.*further enhanced the above approach in their next work [24], where the refined edges detected in the image are subjected to a Hough Transform to cluster nearby groups of parallel lines for improved stair detection confidence.

The approach presented by [23] and [24] proved to be rather popular, as Danilo C. *et al.*[25] [26] emulated the approach with their works by also using the Gabor texture filter for stair edges isolation before using a vanishing point triangulation algorithm to determine the vertical starting and ending point of a flight of stairs. Hough Transform was also applied in [26] for steps reconstruction to further enhance detection accuracy and reliability. The resulting segmentation results of Danilo C. *et al.*are effective enough for obtaining important information in regards to stairs for future autonomous system use. The approach was also robust enough to detect stairs in images of indoor and outdoor origins.

The second main approach in stairs segmentation more commonly used in recent years are done with the aid of depth sensors and wearable hardware devices. These approaches leverage the advantages of AR and its context awareness for extracting stairs more effectively. Young Hoon Lee *et al.*[27] proposed and designed a wearable head-wear system for real time stairs detection by utilizing the Adaboost neural network. The resulting detections from the Adaboost learning algorithm yielded numerous false positives, which were then suppressed by additional information provided by the SLAM (simultaneous localization and mapping) algorithm from the head-wear device. The SLAM provided spatial context to the system in the form of a flat ground plane. Since stairs are characterized by its varying elevation levels in ground planes, false positives can be detected if it does not match its corresponding ground plane patterns.

[28] have also proposed and implemented a real time step detection system with a head-mounted stereo camera device. The approach uses depth disparity differences calculated by the stereo visual input obtained from the head-mounted device to estimate the ground plane differences between each step in a flight of stairs. The segmentation of ground planes through the disparity information was built up throughout the process of stairs traversal, as [28] explains that stairs are usually too large to be observed from a single point of observation. This gradual processing of planes during traversal allowed for a stable and reliable step count detection.

2.3 Color Alteration

For color related operations on AR devices, [29] implemented a real-time color correction algorithm which uses marker images of black and white borders to calibrate and to provide a baseline for a scaling vector to be used for color correction. In an RGB (red, green, blue) image, each color channel was scaled independently and was scaled accordingly based on a Gaussian color distribution between the white and black border. This allows the algorithm to work even without the illumination information of the target object. This method is appropriate under the condition in which a marker feature is present. However, for our case, where arbitrary features are prevalent in real world situations (video feed of indoor environments), the algorithm might fall short in accuracy.

Tanuwidjaja *et al.*[30] proposed a color correction algorithm implemented for AR enabled devices targeted for color blind patients by modifying the HSV (hue, saturation, value) values of an image feed to alter the contrast intensity of the feed. The research also proposed further pixel color processes such as color detection and filtering, as well as daltonization. Daltonization is a process where the entire landscape or field of view

was converted to a different spectrum of hue/color. This is usually applied for accommodating individuals with color blindness, where a more suitable color spectrum will be shifted to allow users to perceive shades of color more effectively. Ananto *et al.*[31] also proposed a similar color correction method to [30] where daltonization was implemented by increasing the contrast of specific colors in order to distinguish certain hues from another.

2.4 Comparison

Based on existing techniques proposed on object segmentation from studied literature, there are 2 methods that could possibly be used in tandem with one another for our use case. Depth based object detection [7] [8] presents itself as an effective and conceptually simple algorithm for object detection. The disparity parameters are easily configurable and the segmentation results for objects in an environment are more satisfactory as compared to other approaches. Edge segmentation [14] can then be applied upon the disparity image to further enhance and distinct the segmented objects within the image.

Contour based approaches [9] are more computationally expensive as compared to the two approaches above as computations must be executed on a frame by frame basis to accommodate contour motion tracking in a non-stationary video feed. Feature extraction based approaches [10] [11] [12] are less robust than other proposed algorithms for our case as SIFT and SURF methods are not applicable for RGB based inputs, which is present in our AR device inputs.

As for object classification methods, the YOLO family of models [20] [21] [22] are generally known to be more prone to object localization errors compared to the RCNN family of models [15] [18] [19]. However, it makes up its lower detection accuracy for its astounding detection speeds (above real time performance) as compared to RCNN family of models. Besides, with the introduction of YOLO v3 [22], detection rates have reached and even eclipsed Faster RCNNs in some data sets while still achieving real time speeds. Real time detection speeds are crucial for our context of research, as it is required by AR devices for real time navigation when worn by users suffering from glaucoma.

Step segmentation algorithms based on pixel manipulation and image filter approaches proposed in [23] [24] [25] [26] proved to be fairly reliable in stairs detection and is more straightforward to implement compared to [27] [28] as it does not require external sensors or hardware. However, the neural network classification approach used in [27] for providing initial stairs classification can be adopted in our research with the use

of previously discussed classification models such as RCNNs and YOLO. The image filter and Hough transform approach proposed in [24] could also be considered for stairs clustering and detection refinement.

For color alteration, HSV value modifications [31] [30] allows for effective and efficient contrast enhancements on visual inputs as compared to the approach used in [29], where prerequisite marker images are required for color correction to take place. The robustness and efficiency of contrast intensification makes the method a better solution in terms of color accentuation.

Chapter 3

Design

3.1 Problem Definition

Glaucoma is a visual disability that causes reported symptoms of tunnel vision, diminished brightness and reduced contrast distinction as mentioned in Chapter 1 [6]. This makes navigation of indoor places for users with glaucoma especially dangerous as various every day indoor objects in an enclosed area not only limits movement options, but also pose as tripping hazards. Stairwells and kerbs in particular, pose a higher risk of traversal for glaucoma patients in indoor establishments as poor lighting conditions could affect legibility of stair edges, especially since users already have compromised brightness and color contrast sensitivity. Figure 3.1 below illustrates how low contrast vision affects how stairs are perceived [32].

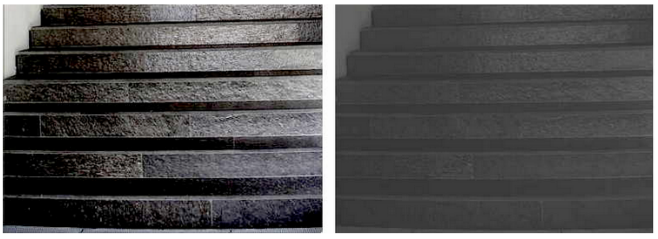


FIGURE 3.1: High contrast stairs (left), low contrast stairs (right)

Therefore, it is paramount that the contextual awareness of AR devices be leveraged to aid sufferers of glaucoma in detecting stairs and kerbs in their surrounding environments. Besides detection, additional information regarding the detected steps, such as the number of steps in a flight, should be provided. Using the cameras of AR devices, live video input could be processed in a frame by frame basis for steps detection and counting. Each frame could be individually processed using approaches presented by [23]

[24] to obtain the number of detected steps. However, [24] reported that its approach in detection, while straightforward and effective, leaves much to be desired in terms of speed, as it took roughly 3 seconds to process a single image for detection.

A right balance of speed and accuracy must be achieved by the proposed system for creating an effective visual aid for people with visual disabilities. An effective navigational aid system must strive for performance as close to real time as possible whilst maintaining reliable accuracy for detection, something that have yet to be achieved by any proposed systems in the field as of writing.

3.2 Objectives

The objective of this research is to propose a solution that utilizes AR for detecting stairs in indoor environments at a reasonable frame rate while also presenting with an algorithm for counting the steps of the detected stairs with reasonable accuracy. While indoor environments are the main focus for this project, the system should be able to work for outdoor environments as well. These objectives complement each other to form a navigational aid for users suffering from glaucoma.

3.3 Requirements

3.3.1 Functional Requirements

- Stairs detection accuracy of at least 60% mAP
- Stairs step count accuracy of at least 70%

3.3.2 Non-functional Requirements

- Deployable solution to work with the Microsoft HoloLens.
- Able to detect staircases or kerbs during traversal.
- Provide step count information of detected steps during traversal.
- Achieve reasonable performance of at least 5 frames per second.
- If time permits, perform contrast and brightness enhancements for detected stairs region only.

3.4 Design

3.4.1 Software and Tools Used

3.4.1.1 HoloLens 2 Emulator

As part of a research collaboration with Nimbus Research Center, this project was initially planned for a hardware implementation (Microsoft HoloLens 2, provided by Nimbus) to go along with the final proposed system. However, due to the in-availability of the Microsoft HoloLens before the submission deadline of this project, the system will be implemented and deployed to the HoloLens 2 emulator.

As the HoloLens 2 emulator is rather new (inital build released in April 16th, 2019), there are various features that have yet to be available in order to accurately emulate the physical HoloLens 2 device, such as the lack of external camera support, thus limiting real life environment testing. Also, due to its emulation nature, features that required on board sensors such as depth sensing and spatial mapping are not available in the HoloLens 2 emulator. The HoloLens 2 emulator used for this project is of version 10.0.18362.1053.

3.4.1.2 Unity Editor

The Unity Editor will be the bridge between the system implementation process and deployment of proposed system to the HoloLens 2 Emulator environment. This is because all interactive components and camera view behavior of a HoloLens 2 application are performed by scripts written in the C# language using the Unity Editor. The version of Unity Editor used in the project is Unity 2018.4.16f1.

3.4.1.3 Microsoft Visual Studio Community 2019

As stated prior, since the HoloLens 2 application behaviour are performed using C# scripts, the Visual Studio 2019 IDE will be used for scripting purposes, as the program supports the C# language implementation natively with plugins support, syntax highlighting and debugging features. Besides, the Visual Studio 2019 IDE is also required for deploying the application solutions generated from Unity Editor into the HoloLens 2 emulator environment. Version 16.4.5 of the Microsoft Visual Studio Community 2019 was used for this project.

3.4.1.4 Darknet YOLO

YOLO will be the chosen object detection and classification method for use in this project, mainly due to its above real time performance in image classification [20] [21] [22]. This is crucial as it allows our project to achieve our objective of having a system that performs at a reasonable frame rate. The YOLO v2 model will be our YOLO model of choice as it provides a higher detection accuracy than [20] with faster training times than that of YOLO v3, albeit being lower in detection accuracy [22]. However, the tradeoff is deemed acceptable considering real time performance is of priority for this project.

Darknet is an open sourced neural network framework that supports a multitude of deep learning and classification models such as RNNs, ResNet and YOLO [33]. Darknet provides all the required tools necessary to build, train and run custom YOLO object detection models. As such, provided with enough stair images as training data, a custom YOLO model for detecting stairs can be built. There are various prerequisite components required in order to get Darknet to work. Table 3.1 below describes all required components to enable Darknet training with GPU along with the current versions of said components used for this project.

Required Components	Currently Used Version
CMAKE	3.17.0
CUDA	10.1
CuDNN	7.6.4
OpenCV	3.3.0
CC Enabled NVIDIA GPUs	MX250 GPU with 6.1 CC
Microsoft Visual Studio	Community 2019

TABLE 3.1: Prerequisites for Darknet

3.4.1.5 OpenCVSharp

OpenCVSharp is a C# wrapper for the popular image processing library available in Python, C++ and Java. It provides all image processing functions needed for image processing operations for this project. Also, the OpenCVSharp library also supports native Deep Neural Network support for image classification with its dnn module, allowing users to just pipe in trained YOLO weights and model configuration files to perform classification. The version of OpenCVSharp used for this project is a UWP version of the wrapper that was based on OpenCVSharp 3.3.1.

3.4.2 System Design Architecture

The proposed system will be implemented and broken down into two distinct parts: the training phase as depicted in Figure 3.2 and the overall system depicted in Figure 3.3.

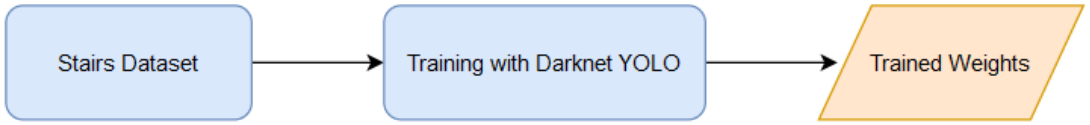


FIGURE 3.2: High Level View of Detection Training Flow

In the training phase, a set of stairs image data set will be fed into the DarkNet neural framework to be trained with YOLO to obtain final training weights. The weights will then be used as part of the overall system architecture.

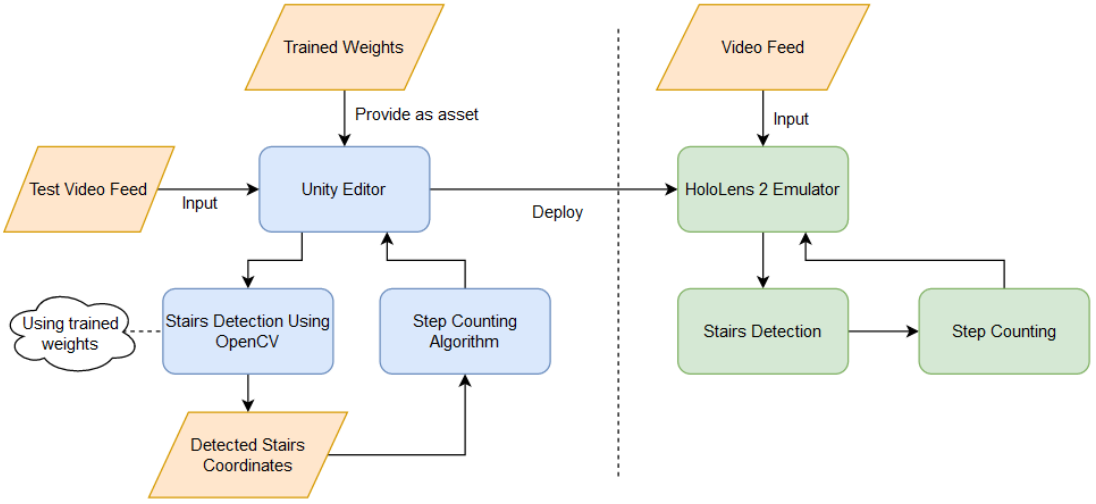


FIGURE 3.3: High Level View of Proposed System Architecture

In the overall architecture, the Unity Editor will be the main hub for all implementation and testing operations prior to deployment, which will be detailed later. A video feed will be used as input for testing purposes to simulate video input of real life environments. This will be akin to a user wearing the Microsoft HoloLens traversing through an environment, with the device camera providing video input. The video input will then be passed to a script containing the stairs detection implementation, which requires the training weights acquired from the training phase in Figure 3.2.

The C# script running the stairs detection algorithm will be implemented using the OpenCVSharp library. This is because the OpenCVSharp library not only contains the required functions for pre-processing of video input, but also allows direct input of

YOLO training weights and training configuration files to be used for stairs detection and classification out of the box. The classification process will return a set of bounding box coordinates as discussed in [20] that could then be used for further processing.

The further processing stage mentioned refers to the step counting algorithm. After reviewing several works in regards to steps segmentation and counting in Chapter 2, the pixel manipulation approaches described by [23] and [24] will be used for this project. This is because other approaches described in [27] and [28] requires the use of physical components found in AR head wear devices such as depth sensors and stereo cameras, which are not available as of yet for this project, and could not be simulated in the HoloLens 2 emulator. **Edge segmentation** (subsection 3.4.2.1) and **Hough Transform** (subsection 3.4.2.2) were chosen as algorithms of choice for step segmentation and counting, due to its relative straightforward approach and effectiveness as reported by [26] in terms of step line grouping. The resulting step count obtained from the step counting algorithm will then be returned to the main script in Unity to be displayed to the user. The stairs detection and step counting process will then be repeated until the end of the input video feed.

So far, the above processes were describing the internal testing and implementation process to be done within Unity Editor itself, as the editor has an built-in player capable of running said scripts and produce log outputs. Therefore, the Unity Editor can be used as a testing ground for early builds of the proposed program before being deployed to the HoloLens 2 emulator. Prior to project deployment into the HoloLens 2 emulator environment, a project solution must first be generated by Unity Editor. The generated C# project solution will then have to be opened with Visual Studio 2019 in order to build and deploy the project as an executable program in the HoloLens 2 emulator's environment. Figure 3.4 describes the flow of project deployment for the proposed system into the emulator.

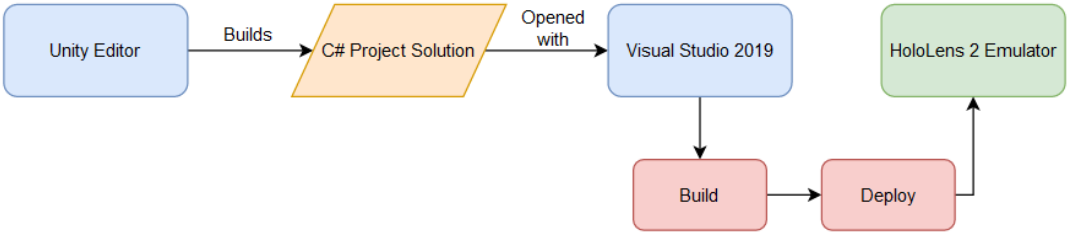


FIGURE 3.4: Flow of Deployment from Unity to Emulator

Once deployed, the program will go through an almost identical process as the test runs performed in the Unity editor once executed. As seen in the right portion of Figure 3.3, the program will execute through the emulator and receives video as input, just like in

Unity. Next, the program will run its stairs detection module and sends the detected stairs bounding box output to the step counting module to count the number of steps in the detected object. The calculated steps will then be returned to the main program for display, and the cycle repeats until the program is terminated.

3.4.2.1 Edge Segmentation

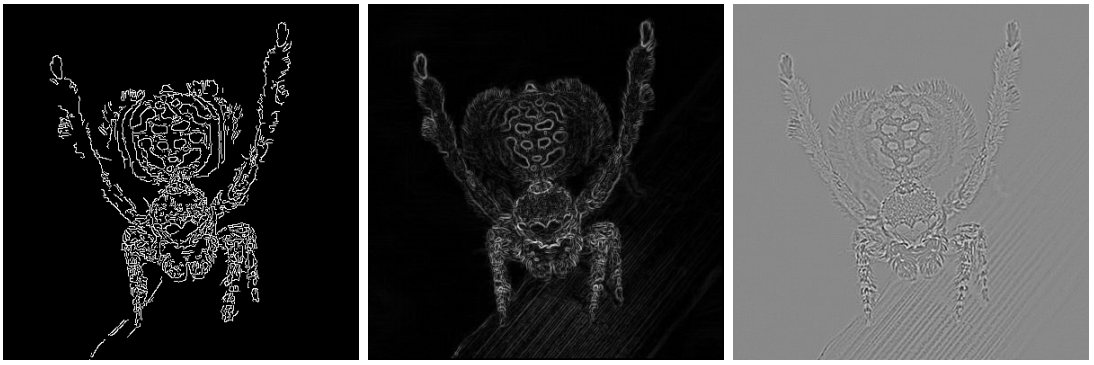
Several edge segmentation algorithms were considered for our system. Canny edge detection is one of the most popular forms of edge detection. Being that it is a multi-stage detector, it usually provides excellent segmentation results for edge lines. The process of Canny edge detection can be boiled down to five phases. The first phase involves convolving a Gaussian smoothing filter (also known as a blurring filter) over the image to remove noise. Next, the intensity gradients for all directions (vertical, horizontal and diagonal) of the input image are computed through the use of four sets of filters. Non-maximum suppression is then applied to the intensity gradients to reduce the amount of unwanted edges within the image. The suppression also acts as a edge thinning procedure for locating edges in the image that are most prominent. The suppressed edges are then subjected to double thresholding to remove weak edges while only preserving stronger edges as characterized by its higher intensity values. Lastly, edge tracking or hysteresis is applied to the thresholded image to connect the stronger preserved lines with certain weaker preserved lines as weaker lines that are closer in proximity to strong lines are more likely to be contributing edges to the original image.

The Sobel operator is a edge detection filter that segments edge lines within an image by convolving it with two 3 x 3 kernels to approximate the derivatives of the image in two directions: horizontal and vertical (equation 3.1). As such, the computation resources required for Sobel operation is much less expensive as compared to the 5 stage approach used in Canny edge detection. The Sobel operator also has the advantage of segmenting lines that are only of horizontal and vertical origins as it calculates the derivatives of both directions separately before being combined to obtain the overall gradient magnitude (equation 3.2).

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A} \tag{3.1}$$

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2} \tag{3.2}$$

Lastly, the Laplacian operator has seen common use in the fields of machine vision as an edge detector. Unlike the Sobel operator, which only calculates first order derivatives with the use of two filter kernels, the Laplacian operator calculates second order derivatives with only one kernel. Due to second order derivative estimation of the input image, noise are more prevalent. Therefore, just like in Canny edge detection, the input image will usually be subjected to a Gaussian smoothing kernel first before Laplacian operator is performed.



(a) Canny Edge Detector (b) Sobel Operator (c) Laplacian Operator

FIGURE 3.5: Comparison of Edge Detection Methods

With all that being said, the Sobel operator was chosen for this project as the ability of the operator to individually isolate horizontal and vertical edges could be useful in obtaining parallel lines within an image, a characteristic shared by stair edges. Also, its relatively low computation requirements will allow our proposed system to perform at better frames per second, which is of high importance for visual aid systems.

3.4.2.2 Hough Transform

Hough Transform is a general shape detection technique that could also be re-purposed for line detection. For line detection, the Hough Transform has to gather enough evidence or votes to determine the existence of a line as given in Equation 3.3, where p denotes the perpendicular distance of the line from the origin and the θ denotes the angle formed by the perpendicular line from a horizontal axis in clockwise direction (Figure 3.6).

$$p = xcos\theta + ysin\theta \tag{3.3}$$

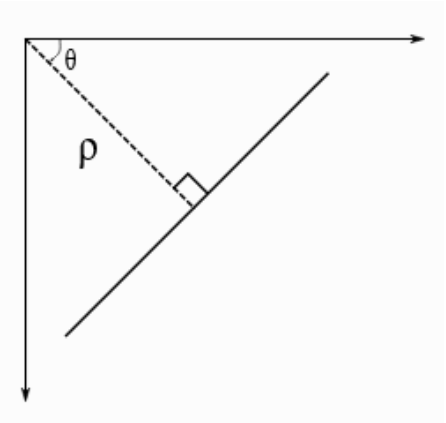


FIGURE 3.6: Visual Representation of Equation 3.3

Essentially, Hough Transform works by first taking any point in an image to construct a line encompassing the maximum diagonal size of the image. The imaginary line will then be rotated at a set of angles (usually 0 to 180 degrees), while an accumulator will record the amount of points within the image that corresponds to the point of the imaginary line at that specific angle. This basically allows the Hough Transform to behave like a histogram bin, where each line rotation records the number of points in the image that the current rotated line comes into contact with. The angle of rotation for the current point that yields the most contacted points will be deemed as a detected line, and the process repeats for all remaining points within the image.

Therefore, the Hough Transform for lines can be used in tandem with edge detection for a more refined line detection. The angle parameter used for the Hough Transform could also be leveraged to target lines of specific angles, which could be useful for parallel lines clustering for edges in stairs.

Chapter 4

Implementation

This section will detail the implementation process for our proposed visual aid system. Implementation process can be broken down into 4 phases in general: the initial setup for development environment, integration of OpenCVSharp image processing library into Unity, Darknet YOLO training and its integration into proposed system architecture, and the stairs step counting algorithm implementation. Each phase will also have its challenges faced during the course of development and its accompanying solutions explained in detail. The full system implementation C# script can be found in Appendix A.

4.1 Environment Setup

4.1.1 Installing Required Components

A few components are required for developing a Hololens 2 application which includes:

- Unity Editor
- Visual Studio for C# development
- HoloLens 2 emulator

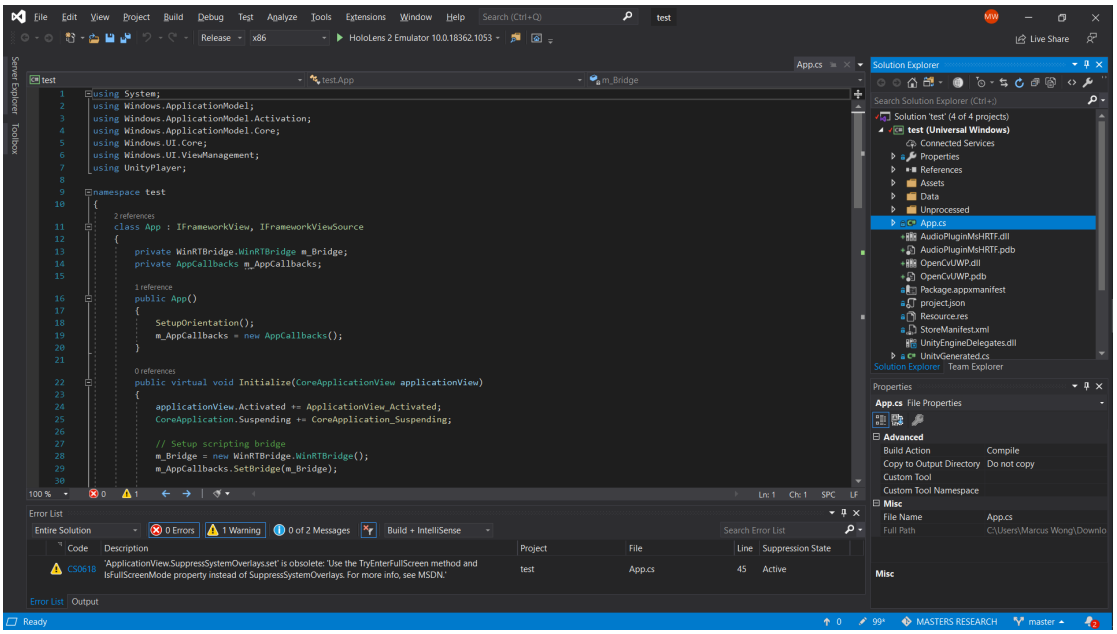


FIGURE 4.1: Visual Studio 2019 Community

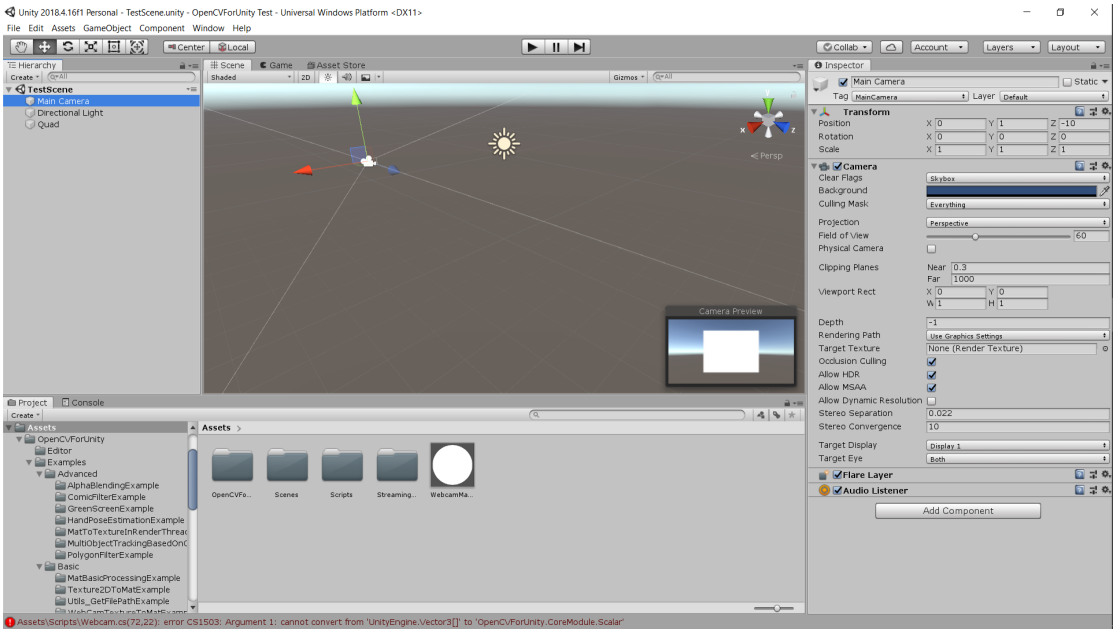


FIGURE 4.2: Unity Editor

The exact versions of installed components was detailed in the previous chapter (Chapter 3). Early test builds of the program would first be tested in Unity Editor itself, as it has a built-in application player for running developed applications. Later development builds would then be deployed and tested on the HoloLens 2 emulator as the physical HoloLens 2 device which would have been provided by the Nimbus research center was unavailable due to delayed shipments of HoloLens 2 units.

Prerequisites are required in order to run the HoloLens emulator. Since the HoloLens 2 emulator is essentially a virtual machine running the HoloLens operating system for device emulation, the Hyper-V feature in Windows host machine must first be activated. Figure 4.7 shows the home page of the installed HoloLens 2 emulator.

FIGURE 4.3: HoloLens 2 Emulator Home Screen

4.1.2 Live Video Feed Simulation

The lack of access to the physical HoloLens 2 device as stated several times prior throughout the report have limited the option of live video input for testing our proposed system. Therefore, a 360 degree video will be integrated into the surrounding view of our proposed system to simulate real world environments up to a certain degree as it allows users to look around the video feed in all directions. This can be done using the Unity Editor based on a tutorial found in [34]. The process can be broken down into 6 steps:

1. Download the panoramic shader from [35] and import into Unity Editor as asset along with 3D video of choice.
2. Enable virtual reality feature in Player Settings of Unity Editor.
3. Create a new Render Texture with matching dimensions as selected 3D video, then set depth buffer as none.
4. Place selected 3D video into scene view and set the target render texture property to the previously newly created render texture.

5. Create a new empty material with the following properties:

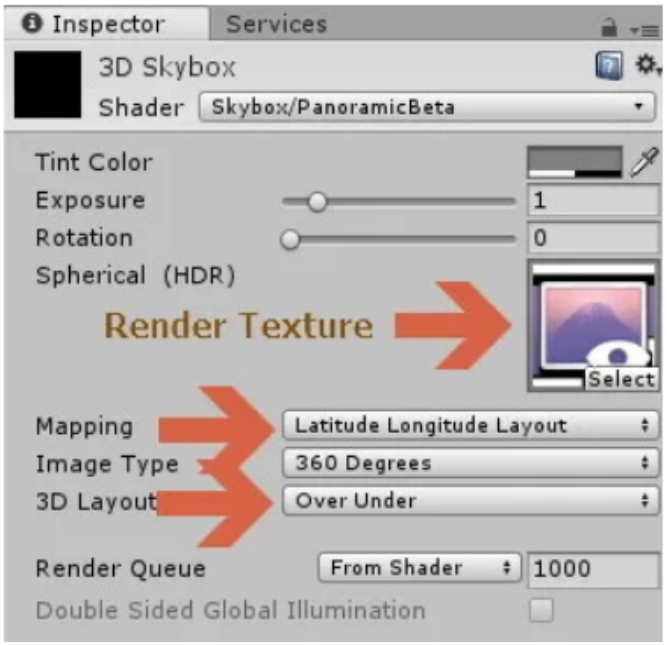


FIGURE 4.4: Properties for Skybox Material

6. Assign the created skybox material into the lighting settings of the Unity project.

4.1.3 Deploying to HoloLens 2 Emulator

Once the 360 video feed was implemented as a Skybox for the application, the application could then be deployed to the HoloLens 2 emulator for testing. As mentioned before, since all Unity applications could be executed within Unity itself, initial testing will first be executed in the editor. However, the deployment process to the emulator environment could be performed at any time during the development phase to ensure the program works properly in the HoloLens. This section details the steps required to deploy a working Unity program into the HoloLens emulator environment, which include:

1. Building the Unity project with the following settings in Build Settings:

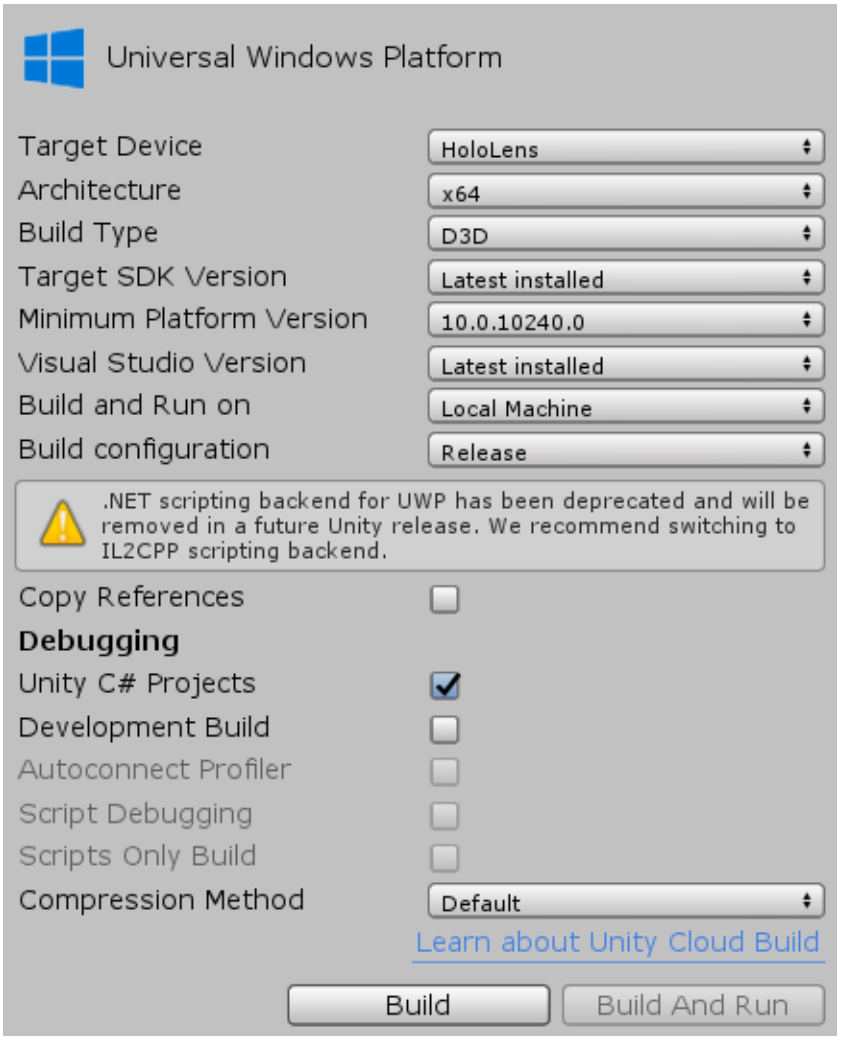


FIGURE 4.5: Settings for Unity Build

2. Open the generated project solution with Microsoft Visual Studio
3. Set the project solution in Microsoft Visual Studio to run in Release mode.
4. Set the target architecture of the project to x86.
5. Set the project to run with target device set as HoloLens 2 Emulator.

Once the steps above are done, the project can then be executed in Visual Studio. Once executed, the project will be compiled and deployed into the HoloLens 2 emulator.

4.1.4 Difficulties Faced

As mentioned at the start, the HoloLens 2 emulator required the Hyper-V Windows feature to be enabled first. However, the Hyper-V feature is only available for certain versions of the Windows operating system (Windows 10 Pro and Windows 1- Education). Since the current development machine is running on Windows 10 Home Edition, the HoloLens 2 emulator could not be launched.

Initially, using Oracle's VirtualBox software, a virtual machine was set up with the supported Windows version (Windows 10 Pro) to run the HoloLens emulator. However, attempts to launch the emulator in the virtual machine remained unsuccessful, as it was later found that the VirtualBox software does not support virtual nesting. Another virtual machine was then set up, this time using the VMWare Workstation program. While the HoloLens emulator did manage to launch in the VMWare Workstation virtual machine, performance was dreadful with the emulator running at only 2-4 fps. This is because VMWare Workstation could not virtualize and fully utilize the graphic prowess of the GPU in the host machine.

Therefore, the development machine itself had to be upgraded to one of the supported version of Windows. With the help and advice from Sean, the project supervisor of this research, a product key for the supported Windows version was obtained from the Computing Department of CIT. Once that is done, the HoloLens 2 emulator did perform better than the virtual machine at the end, albeit by a only a small amount at around 10-15 fps. This is due to the virtual GPU of the emulator not working as pictured in Figure 4.6, which might be due to the GPU of the current test machine (NVIDIA GeForce MX250) not being supported by the emulator as of yet. The Nimbus Research Center did initially offer a powerful development machine for use within their office premises which might solve this issue. However, due to the quarantine lock-down for the COVID-19 pandemic, the project could not be moved to the new machine.

FIGURE 4.6: GPU Not Running Within HoloLens 2 Emulator

4.2 OpenCVSharp-Unity Integration

As the project will rely on the OpenCVSharp library for image processing operations, integration into the Hololens development environment is crucial. As there are currently no free commercial wrappers of the OpenCVSharp library for UWP applications development on Unity and Hololens, manual efforts have to be done to configure and migrate the OpenCVSharp library into the development environment.To have OpenCVSharp

to work in Unity, C++ Universal DLLs containing the OpenCVSharp packages must be generated. Once generated, the DLLs must then be moved into the plugins directory in the Unity project's asset folder. This would allow C# unity scripts to access OpenCVSharp functionalities. To quicken the process, the OpenCVSharp DLLs were directly obtained from an open source repository [36] and was integrated into our development environment.

4.2.1 Unity and OpenCV Interactions

A Unity test script was created to test the functionalities of the imported OpenCVSharp library. The script involved applying Canny edge detection on a 2d projected video file input to achieve a binary video footage with its edges segmented as shown in Figure below.

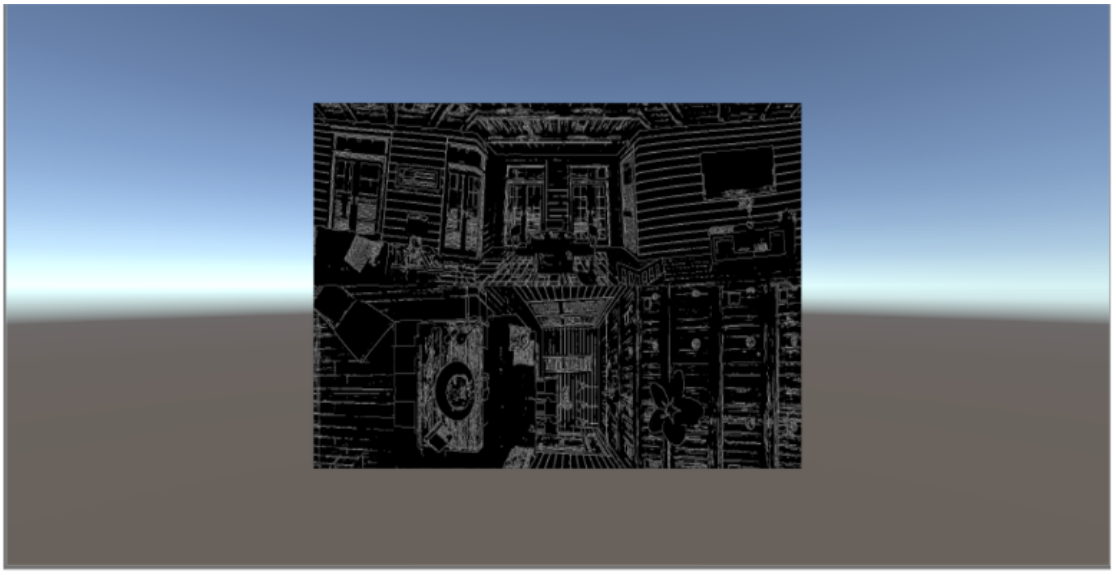


FIGURE 4.7: Testing Canny Edge Detection Functionality of OpenCVSharp

To achieve this, understanding of the interactions between OpenCVSharp and Unity must first be established. OpenCVSharp libraries performs processing of image data through the "Mat" data type, whereas Unity provides image data and other visual information through a proprietary data type known as Textures. Various other data types are inherited from the Texture data type class, such as Render Textures, Texture 2D, Mesh Textures *etc.*. Therefore, Textures must first be converted to Mat data type in order to input visual information from Unity's view into OpenCVSharp functions. Once OpenCVSharp processing is complete, the resulting Mat would have to be converted back to the Texture data type for display. Figures 4.8, 4.9 and 4.10 illustrates the codes for the above process.

```
    private void Update()
    {
        TextureToMat();
        Cv2.Canny(_image, _imageOut, 100, 100);
        MatToTexture();
    }
```

FIGURE 4.8: Canny Edge Detection Function Used in OpenCVSharp

```
    private void TextureToMat() {
        // get width and height info
        int width = (int)vid.clip.width;
        int height = (int)vid.clip.height;

        // to convert the Texture datatype of vid.texture into rendertexture
        // then convert the rendertexture into texture2d.
        // to allow access to raw texture data
        rtex = new RenderTexture(width, height, 32);
        Graphics.Blit(vid.texture, rtex);
        RenderTexture.active = rtex;

        _vid_texture.ReadPixels(new UnityEngine.Rect(0, 0, rtex.width, rtex.
↪height), 0, 0);
        _vid_texture.Apply();

        byte[] c = _vid_texture.GetRawTextureData();

        // convert the raw texture data of video frame into mat
        _image = new Mat(height, width, MatType.CV_8UC4, c);

        // release the active rendertexture to prevent memory leak
        rtex.Release();
    }
```

FIGURE 4.9: Code for Texture to Mat Conversion

```
private void MatToTexture()
{
    int width = (int)vid.clip.width;
    int height = (int)vid.clip.height;

    // _cannyImageData is byte array, because canny image is grayscale
    _imageOut.GetArray(0, 0, _imageOutData);

    // Create Color32 array that can be assigned to Texture2D directly
    Color32[] c = new Color32[width * height];

    // Parallel for loop
    for (int i = 0; i < height; i++)
    {
        for (var j = 0; j < width; j++)
        {
            byte vec = _imageOutData[j + i * width];
            var color32 = new Color32
            {
                r = vec,
                g = vec,
                b = vec,
                a = 0
            };
            c[j + i * width] = color32;
        }
    }

    // assign the color32 array into output texture
    _texture.SetPixels32(c);

    // Update the output texture, OpenGL manner
    _texture.Apply();
}
```

FIGURE 4.10: Code for Mat to Texture Conversion

The Update() function in Figure 4.8 is a fundamental part of a C# Unity script which was called every frame during program execution. Therefore, all image processing operations such as texture to mat conversions, mat to texture conversions and edge detection operations should be performed here as the operations will be applied to each frame of the input video feed.

4.2.2 Problems Faced

While the scripts above are functional and working within the Unity editor's built-in player, attempts to run said applications inside the HoloLens 2 emulator after deploying yields an error, indicating missing DLLs when calls to any OpenCVSharp functionalities were made within the test scripts. The missing DLL error is shown in Figure 4.11 below.

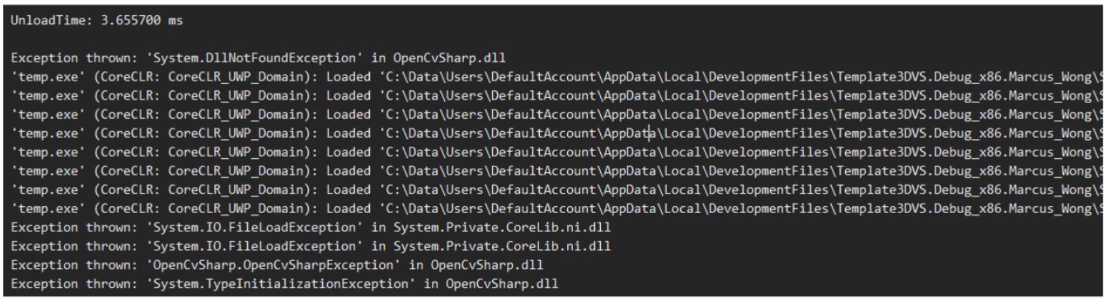

```
UnloadTime: 3.655700 ms

Exception thrown: 'System.DllNotFoundException' in OpenCvSharp.dll
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
'temp.exe' (CoreCLR: CoreCLR_UWP_Domain): Loaded 'C:\Data\Users\DefaultAccount\AppData\Local\DevelopmentFiles\Template3DVS.Debug_x86.Marcus_Wong\
Exception thrown: 'System.IO.FileLoadException' in System.Private.CoreLib.ni.dll
Exception thrown: 'System.IO.FileLoadException' in System.Private.CoreLib.ni.dll
Exception thrown: 'OpenCvSharp.OpenCvSharpException' in OpenCvSharp.dll
Exception thrown: 'System.TypeInitializationException' in OpenCvSharp.dll
```

FIGURE 4.11: DLL Error In OpenCVSharp During HoloLens Emulator Deployment

The issue was later found to be a mismatch of the targeted architecture version for the imported OpenCVSharp library DLLs in Unity Editor. Normally, when the targeted architecture setting in Figure 4.5 was set, the properties would be applied to all relevant DLLs and plugins in the Unity project. However, while the target architecture settings for the OpenCVSharp DLLs were changed accordingly, it was not saved and applied after the Unity project was built. Therefore, manual intervention was performed by manually applying and saving the target architecture changes to all 46 OpenCVSharp DLLs. Once that was done, the built Unity project was successfully deployed into the HoloLens emulator environment. This problem was particularly hard to identify and have caused a delay to the implementation phase of this project by almost 2 weeks.

4.3 Darknet YOLO

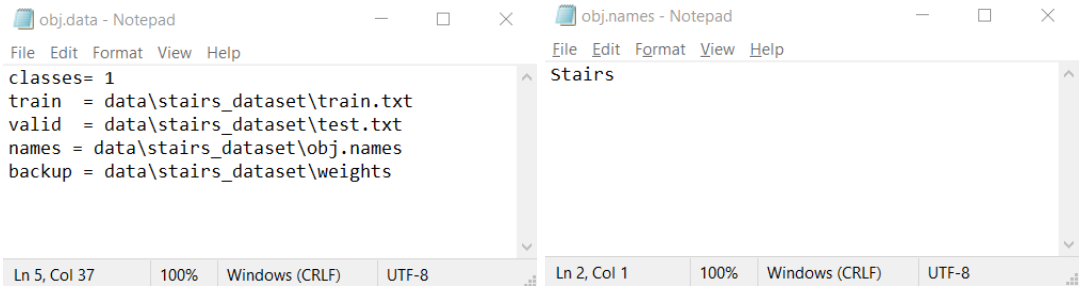
While the chosen YOLO v2 model allows for stairwell detection that was included in its pre-trained weights provided by Joseph Redmon *et al.*[21], the pre-trained weights also comes with object detection of almost 9000 other classes which cannot be selected to be opt out of. Therefore, custom training of the YOLO v2 model on detecting only stairs would have to be done, which would be explained in detail in subsection 4.3.1 below.

4.3.1 Training

As specified in Chapter 3, several components such as CUDA, cuDnn and OpenCV must first be installed to allow YOLO training through Darknet with the use of GPU. Once the required components are installed, two files were created:

1. obj.data
2. obj.names

The *obj.names* file is a file containing only the object class name that a valid detection should be identified with. In the context of our research, since we only need to detect stairs, the file should only contain the line "stairs". Next, the *obj.data* file contains crucial information required during training such as the number of object classes the model is supposed to detect, the training data directory, the validation data directory, the directory of the *obj.names* file and the backup directory. Figure 4.12 shows the created file and its contents used for training the model.



(a) obj.data File (b) obj.names File

FIGURE 4.12: Contents for obj.data (a) and obj.names (b) files

Next, the model configuration file *(.cfg)* for training needs to be created. Again, the configuration file is crucial for the training process as it contains the neural network configuration information of the selected YOLO model used during training such as the number of image batches, number of memory subdivisions, number of classes, number of convolutional filters for each convolutional layer *etc.*. Since the GPU of our current development machine has only 2GB of memory, the number of batches and subdivisions were set to 64. The training images will be resized to a dimension of 320 x 320 during training, denoted as height and width in the configuration file. The number of classes was set to 1 in the file as the model is only detecting stairs. Lastly, the filter attribute for the last convolutional layer should be set with the following criteria : filters=(classes + 5)*5. Making our filter number for the last layer 30.

Now for training the model, a stairs dataset as provided by the works of Patil *et al.*[37] was used for our project. It contains 848 stair images that have been annotated in YOLO format, all of which have been subjected to data augmentation process of horizontal flipping, giving the dataset a total of 1696 images. The dataset also includes 204 (after data augmentation) fully annotated test images. Using these images, the model was trained for around 2000 iterations, which was the recommended stopping point of training as stated in [33]. Once trained, the resulting weights will be saved into the specified directory listed in the "backup" attribute of the *obj.data* file. The weights will be saved every 100 iterations of training and the best performing weights will be saved

with the *best.weights* suffix appended to the weights file name. The obtained weights were then used for a test detection.Figure 4.13 illustrates the weights (along with the best weight) obtained after training and Figure 4.14 shows a prediction made using the newly trained model for stairs detection.

yolo-voc_1000.weights	8/5/2020 4:07 AM	WEIGHTS File	197,574 KB
yolo-voc_2000.weights	8/5/2020 8:30 AM	WEIGHTS File	197,574 KB
yolo-voc_best.weights	8/5/2020 9:27 AM	WEIGHTS File	197,574 KB
yolo-voc_last.weights	8/5/2020 9:52 AM	WEIGHTS File	197,574 KB

FIGURE 4.13: Obtained Weights Files After Training

FIGURE 4.14: Successful Prediction Obtained From Trained Model

4.3.2 Implementation

Once training phase is complete, the best performing resulting weights and the model configuration file used during training will be transferred to the Unity Editor to be used as an asset. In the C# script, the paths for the configuration file and weights will be initialized. Next, the YOLO network within OpenCVSharp will be created using OpenCVSharp "dnn" module's *ReadNetFromDarknet* function, where it accepts the initialized paths of the configuration and weights file as parameters to load the weights and network configuration. Figure 4.15 illustrates the code for the process.

```
var m_Path = Application.dataPath;
var cfg = m_Path + "/YOLO/yolo-voc.cfg";
var weights = m_Path + "/YOLO/yolo-voc_1000.weights";
net = CvDnn.ReadNetFromDarknet(cfg, weights);
```

FIGURE 4.15: Code for Initializing YOLO in Unity

Next, within the Update() function of the Unity script, the *blobFromImage* method was called, where it first takes in the input image (a video frame) to generate a blob data, which was then set as input for the neural network through the *SetInput* method of the initialized network from Figure 4.15. Once the input was set in the network, the *Forward* function pushes the input through the network, initiating a forward pass operation to predict the presence of stairs in the input image.

```
var blob = CvDnn.BlobFromImage(_image, 1 / 255.0, new Size(1147, 536),
                                    new Scalar(), true, false);
net.SetInput(blob, "data");
var prob = net.Forward();

predict(prob);
```

FIGURE 4.16: Code for Pushing Input Image into YOLO Network for Prediction

The results returned by the *Forward* function is an array containing the bounding box coordinates of detected objects, confidence and probability of the detected object. The resulting bounding box coordinates could then be used in the stairs counting algorithm, which will be further explained in the next section (4.4). As seen from Figure 4.16, the *predict* function is a user defined function that accepts the results returned by the forward pass operation as input to visualize the predictions made by the forward pass of the network.

```
private void predict(Mat prob){
    for (int i = 0; i < prob.Rows; i++)
    {
        var confidence = prob.At<float>(i, 4);
        if (confidence > threshold)
        {
            //get classes probability
            Cv2.MinMaxLoc(prob.Row[i].ColRange(prefix, prob.Cols), out
                                    _, out Point max);
            var classes = max.X;
            var probability = prob.At<float>(i, classes + prefix);

            if (probability > threshold)
            {
                var color = Scalar.RandomColor();
                //get center and width/height
                var centerX = prob.At<float>(i, 0) * w;
                var centerY = prob.At<float>(i, 1) * h;
                var width = prob.At<float>(i, 2) * w;
                var height = prob.At<float>(i, 3) * h;
                var label = $"{Labels[classes]} {probability *
                                    100:0.00}%";
                print($"confidence {confidence * 100:0.00}% {label}");
                var x1 = (centerX - width / 2) < 0 ? 0 : centerX - width
                                    / 2;

                //draw result
                _image.Rectangle(new Point(x1, centerY - height / 2),
                                    new Point(centerX + width / 2,
                                    centerY + height / 2), color, 2);
                var textSize = Cv2.GetTextSize(label,
                                    HersheyFonts.HersheyTriplex, 0.5,
                                    1, out var baseline);
                Cv2.Rectangle(_image, new OpenCvSharp.Rect(new Point(x1,
                                    centerY - height / 2 -
                                    textSize.Height - baseline),new
                                    Size(textSize.Width,
                                    textSize.Height + baseline)),
                                    color, Cv2.FILLED);
                Cv2.PutText(_image, label, new Point(x1, centerY -
                                    height / 2 - baseline),
                                    HersheyFonts.HersheyTriplex, 0.5,
                                    Scalar.Black);
            }
        }
    }
}
```

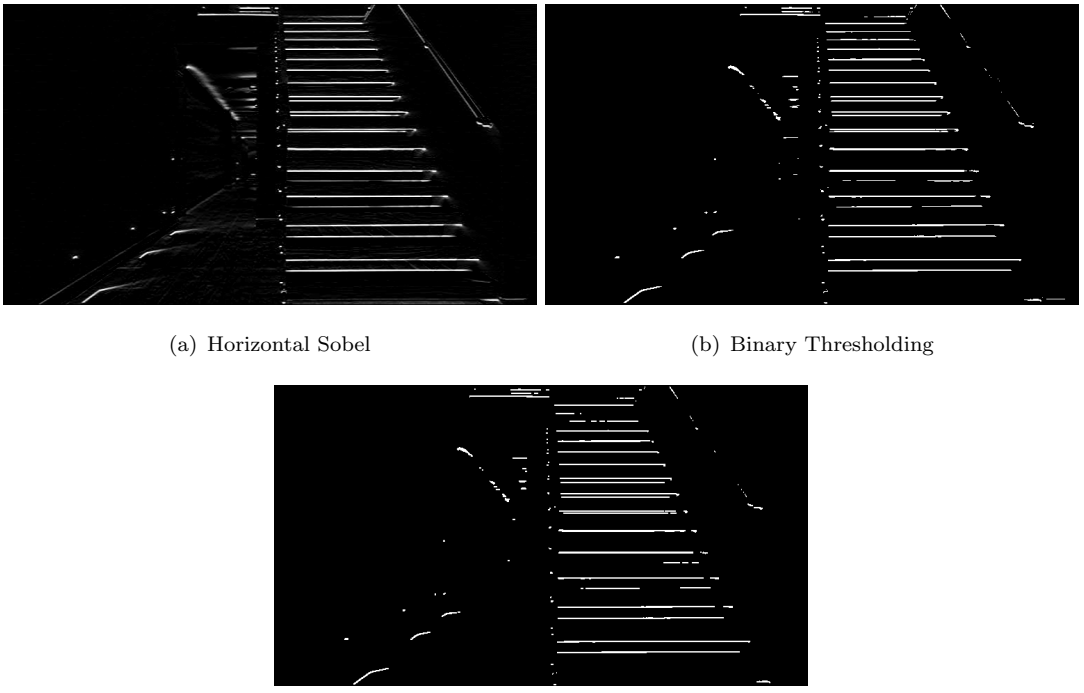
FIGURE 4.17: Code for Visualizing YOLO Predictions

4.4 Stairs Step Counting Algorithm

The step counting algorithm used in the project can be broken down into 4 phases. Implementation of each phase will be represented in chronological order and will be detailed in its own subsection.

4.4.1 Edge Segmentation

First, the input stairs image will be subjected to edge segmentation to isolate the stair edges and also serves as a pre-processing stage for the algorithm.



(a) Horizontal Sobel

(b) Binary Thresholding

(c) Morphological Opening

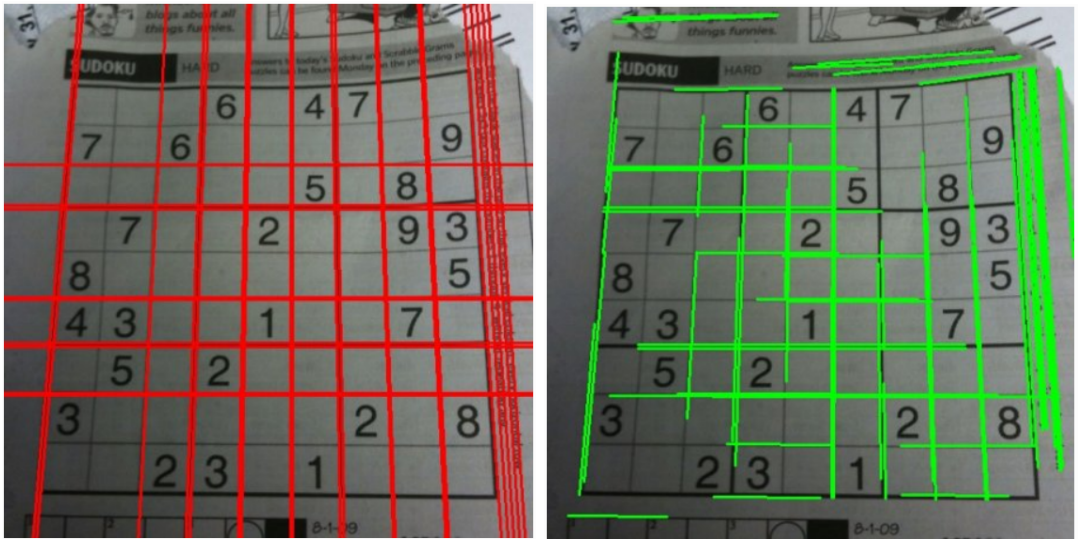
FIGURE 4.18: Edge Segmentation Process of Stairs Image

The image will first be converted its grayscale representation before being subjected to a y-axis Sobel operator. This is done so for the prominent horizontal edges commonly found in stairs to be easily distinguished for future operations. The resulting image after the horizontal Sobel operation will then be subjected to binary thresholding to suppress any remaining noise or soft edges detected by the Sobel operator. This leaves a clean horizontal edge only binary image that would ease and enhance the effectiveness of the next phase (subsection 4.4.2). The binary edge image will finally be subjected to morphological opening to thin out the detected edges and also to remove any remaining noise within the image. Morphological opening is basically an erosion operation followed

by a dilation, which allows noise to be eroded and remaining strong features to be restored. Figure 4.18 illustrates the edge segmentation steps described above.

4.4.2 Horizontal Line Detection

Upon completion of edge segmentation phase, the edge segmented image will be subjected Hough Transform for line detection. Since the segmented edges are already preprocessed to only contain strong horizontal edges, the probabilistic variant of the Hough transform will be used. Probabilistic Hough Transform is an optimized form of Hough Transform, where instead of taking all points into consideration for line detection, this variant of the algorithm only takes a subset of points for determining the existence of a line, thereby increasing the efficiency of the line detection in terms of execution speed. The number of points it takes to determine a line can be controlled with a threshold value. Another advantage to the probabilistic Hough Transform is that unlike the regular Hough Transform, the detected lines returned does not extend to each end of the image (Figure 4.19) as the sampling of a subset of points for line detection allows the algorithm to return the exact length of the detected line.



(a) Hough Line Transform (b) Probabilistic Hough Line Transform

FIGURE 4.19: Difference Between Non-probabilistic and Probabilistic Hough

However, it can be observed in Figure 4.19 that the lines returned by probabilistic Hough Transform are more sporadic and numerous compared to regular Hough lines transform due to the aforementioned sampling of only a subset of points. Therefore, merging of nearby lines must be performed to reduce the amount of duplicate or redundant lines.

4.4.3 Line Merging

The line merging algorithm is based upon two metrics for merging criteria: the minimum distance between each line to merge and the minimum angle between two lines to merge. The lines returned in the Probabilistic Hough Transform phase will be iterated through and each line will be compared against another line in terms of its distance between each other.

To calculate the distance between two lines, the distance between the end point of one line and another line must be calculated. Since the positions of the lines are unknown, there would be 4 possible line distance measurements between the two lines. Figure 4.20 below illustrates the code implementation for calculating the line distance for the 4 points.

```
def get_distance(line1, line2):
    dist1 = DistancePointLine(line1[0][0], line1[0][1],
                              line2[0][0], line2[0][1], line2[1][0],
                                    line2[1][1])
    dist2 = DistancePointLine(line1[1][0], line1[1][1],
                              line2[0][0], line2[0][1], line2[1][0],
                                    line2[1][1])
    dist3 = DistancePointLine(line2[0][0], line2[0][1],
                              line1[0][0], line1[0][1], line1[1][0],
                                    line1[1][1])
    dist4 = DistancePointLine(line2[1][0], line2[1][1],
                              line1[0][0], line1[0][1], line1[1][0],
                                    line1[1][1])
    return min(dist1,dist2,dist3,dist4)
```

FIGURE 4.20: Calculating Distance between Lines Using 4 Line Points

With the end points of the two lines (4 points), each point will be subjected to the formula presented in equation 4.1 as written by Paul Bourke [38] to calculate the value of $\boldsymbol{u}$, which could then be used to approximate the intersection point between the end point and the compared line (equation 4.2). Now, with the intersection point information and the end point coordinates, the magnitude (distance) of the line can be calculated with equation 4.3.

$$u = \frac{(x_3 - x_1)(x_2 - x_1) + (y_3 - y_1)(y_2 - y_1)}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} \tag{4.1}$$

$$(x, y) = ((x1 + u(x_2 - x_1)), (y1 + u(y_2 - y_1))) \tag{4.2}$$

$$magnitude = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.3}$$

The sequence of calculations above was implemented as below:

```
def DistancePointLine(px, py, x1, y1, x2, y2):
    LineMag = lineMagnitude(x1, y1, x2, y2)

    if LineMag < 0.00000001:
        DistancePointLine = 9999
        return DistancePointLine

    u1 = (((px - x1) * (x2 - x1)) + ((py - y1) * (y2 - y1)))
    u = u1 / (LineMag * LineMag)

    if (u < 0.00001) or (u > 1):
        #// if closest point does not fall within the line segment, take
                                    the shorter distance to an endpoint
        ix = lineMagnitude(px, py, x1, y1)
        iy = lineMagnitude(px, py, x2, y2)
        if ix > iy:
            DistancePointLine = iy
        else:
            DistancePointLine = ix
    else:
        # Intersecting point is on the line, use the formula
        ix = x1 + u * (x2 - x1)
        iy = y1 + u * (y2 - y1)
        DistancePointLine = lineMagnitude(px, py, ix, iy)
    return DistancePointLine
```

FIGURE 4.21: Code for Calculating Minimum Distance Between Point and Line

If the distance between the lines are within the minimum line distance threshold, the angles of the two lines will be calculated using the following formula:

$$\theta = tan^{-1} \frac{y_2 - y_1}{x_2 - x_1} \tag{4.4}$$

Upon obtaining the angle of the two lines, the absolute values of the angles will be subtracted to find the angle difference between both lines. If the angle difference between the lines does not exceed the minimum angle threshold, the lines will be merged. The process then repeats till all detected lines have been iterated through. The end result of the line merging algorithm is shown in figure 4.22

(a) Before Line Merging (44 Lines)

(b) After Line Merging (14 Lines)

FIGURE 4.22: Before and After Line Merging Algorithm

4.4.4 Line Centroids and Line of Best Fit

As the step counting algorithm is dependent on the bounding box area returned by the YOLO detection model, which is usually small in dimension and does not fully encapsulate the entire structure of the stairs, the number of steps detected will not be as accurate. Therefore, the algorithm has to estimate the incline angle of the stairs

and create a larger bounding box enveloping the estimated angle in order to limit the inclusion of detected horizontal lines that does not belong to the stairs.

To achieve this, the centroids of the horizontal lines within the initial bounding box will first be calculated. Using the calculated centroids, a line of best fit will be constructed to fit between the centroids. The line of best fit will be extended through the entirety of the y-axis of the image, as it was assumed that a flight of stairs will encompass a large section of the vertical.

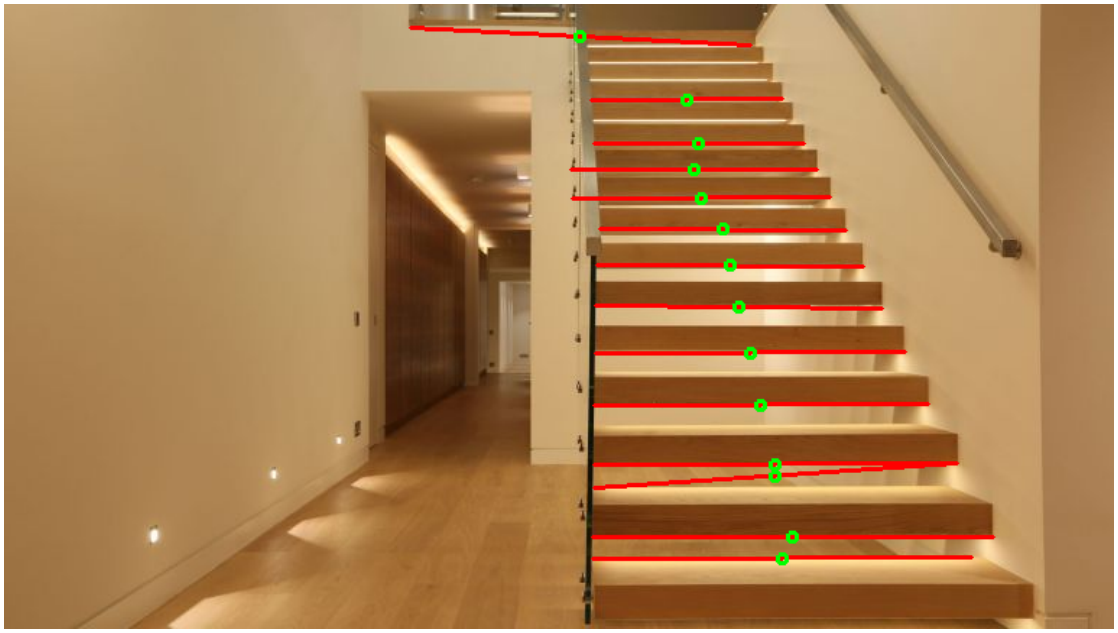


FIGURE 4.23: Centroids of Merged Lines (Green)

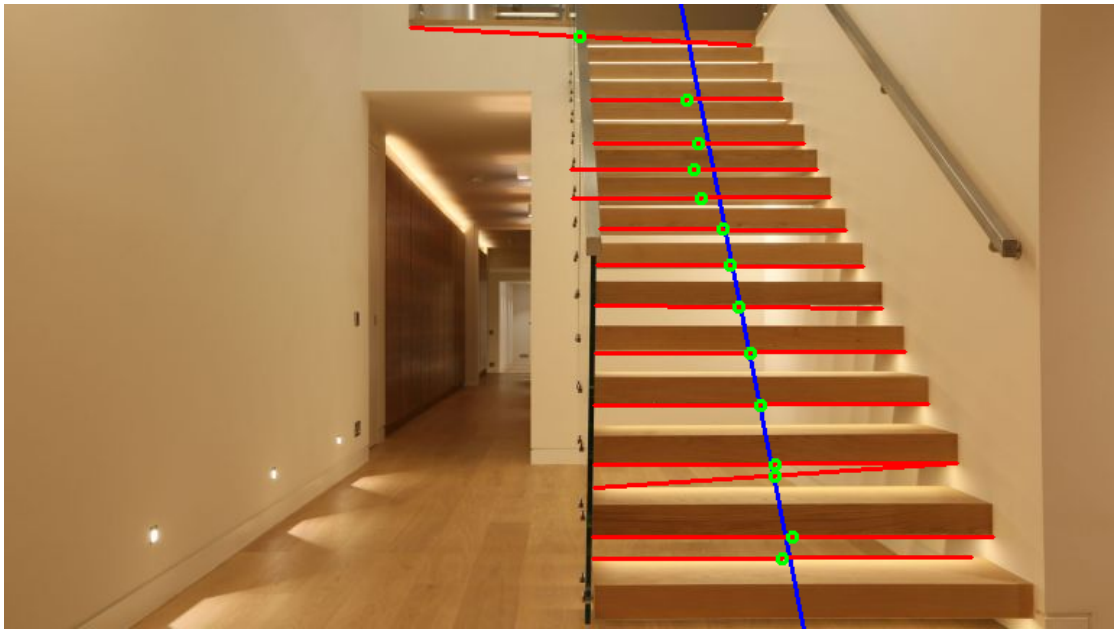


FIGURE 4.24: Line of Best Fit (Blue)

```
for line in storedLines:
    x1,y1,x2,y2 = line[0][0],line[0][1],line[1][0],line[1][1]

    centroid = (x1+int((x2-x1)/2), y1+int((y2-y1)/2))
    centroids.append(centroid)

# get line of best fit for all centroids
vx,vy,x,y = cv2.fitLine(centroids,cv2.DIST_L2,0,0.01,0.01)
```

FIGURE 4.25: Code for Calculating Centroids and Line of Best Fit

The line will then have its angle and direction calculated before a bounding box is constructed to enclose the line of best fit. The newly generated bounding box will become the new bounding box for the detected stairs. Next, each horizontal line or centroid that is in contact with the newly generated bounding box will be classified as a valid stair step, which will be then be accumulated and presented as the final step count for the input stair image.

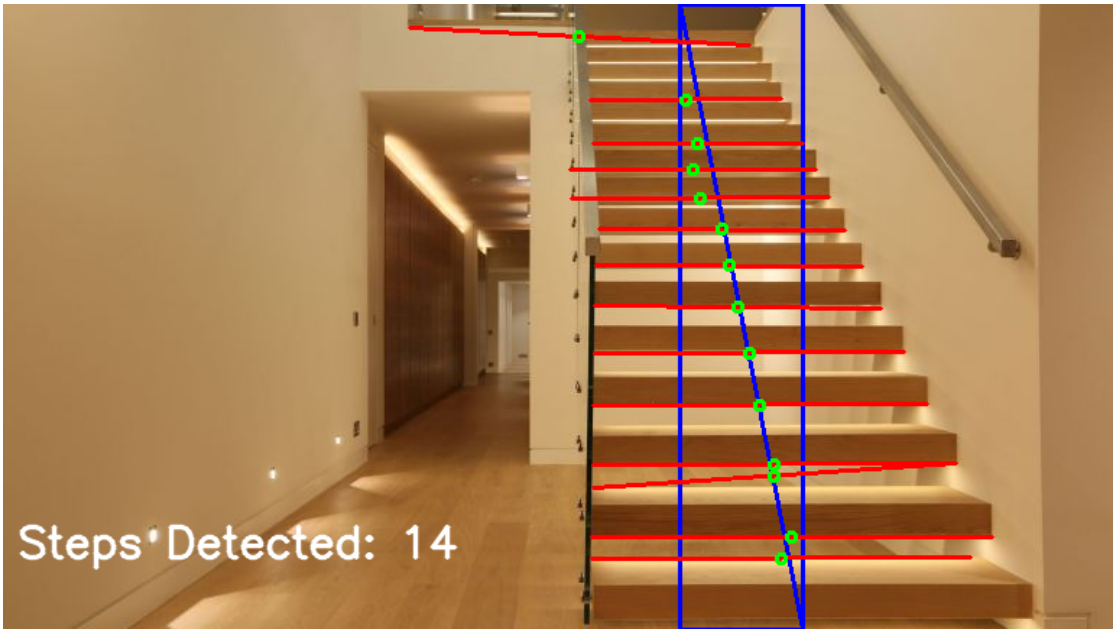


FIGURE 4.26: Final Step Count with New Bounding Box

Chapter 5

Testing and Evaluation

5.1 YOLO Stairs Detection

5.1.1 Testing

The testing phase for YOLO stairs detection occurs simultaneously with the training phase of YOLO. Using the "-map" command during the training, Darknet will automatically calculate important information regarding the performance of the model such as average loss score, mAP, average IoU (intersection over union) *etc.* by testing the model against a validation dataset that was unseen by the model during training. A total of 208 validation images were used for testing, obtained from [37]. Also, the model was trained for a total of 2300 iterations, slightly above the recommended number of training iterations for one object class [33]. The average loss scores were recorded for all iterations, whereas other performance metrics (*e.g* mAP, IoU, F1 score etc.) were calculated every 100 iterations after the initial warm up period of 400 iterations.

A total of 8 performance metrics will be recorded and used to assess the performance of the model: Number of false positives (FP) detected, number of false negatives (FN), number of true positives (TP), recall score, F1 score, average IoU, precision score and mAP (mean average precision). The explanation for some of the above metrics will be listed below:

1. **Precision** : Indicates the percentage of correct predictions. Calculated by (TP / TP + FP).
2. **Recall** : Indicates the accuracy of positive predictions. Calculated by (TP / TP + FN).
3. **F1 Score** : Indicates the balance between precision and recall. Calculated as follows: 2*((precision*recall)/(precision+recall))

4. **Average IoU**: Measures the overlap between the predicted object boundary and the actual object boundary. Returns a true positive or false positive result based on a predefined threshold value, where it is 0.24 by default for Darknet [33].
5. **mAP**: Calculates the mean precision value for each object class probability. Important to note that the mAP metric calculated in Darknet is the same as the AP_{50} metric used in MS COCO competitions.

With that being said, the best performing model will be assessed and determined by the best overall scores for the above listed metrics. The Darknet metric calculations where all performed with a default threshold value of 0.25 for all metrics with the exception of IoU calculation. The training and testing of the YOLO model was performed on the 10 watt variant of the NVIDIA GeForce MX250 GPU, which yielded a detection speed of roughly 50 milliseconds per image. Figure 5.1 depicts one such metrics calculation performed by Darknet after each 100 iterations.

```
 calculation mAP (mean average precision)...

 detections_count = 2417, unique_truth_count = 208
 rank = 0 of ranks = 2417  rank = 100 of ranks = 2417  rank = 200 of ranks = 2417  rank = 300 of ranks = 2417  rank = 40
0 of ranks = 2417  rank = 500 of ranks = 2417  rank = 600 of ranks = 2417  rank = 700 of ranks = 2417  rank = 800 of ran
ks = 2417  rank = 900 of ranks = 2417  rank = 1000 of ranks = 2417  rank = 1100 of ranks = 2417  rank = 1200 of ranks =
2417  rank = 1300 of ranks = 2417  rank = 1400 of ranks = 2417  rank = 1500 of ranks = 2417  rank = 1600 of ranks = 2417
  rank = 1700 of ranks = 2417  rank = 1800 of ranks = 2417  rank = 1900 of ranks = 2417  rank = 2000 of ranks = 2417  ra
nk = 2100 of ranks = 2417  rank = 2200 of ranks = 2417  rank = 2300 of ranks = 2417  rank = 2400 of ranks = 2417 class_i
d = 0, name = Stairs, ap = 69.84%        (TP = 158, FP = 45)

 for conf_thresh = 0.25, precision = 0.78, recall = 0.76, F1-score = 0.77
 for conf_thresh = 0.25, TP = 158, FP = 45, FN = 50, average IoU = 51.52 %
```

FIGURE 5.1: Metrics Calculation in Darknet

The training log output from the console was recorded into a text file. The text file was then used as input for a Python script to scrape the metric results for each calculated iteration. The scraped metric scores were then saved into a CSV file and was used to generate the training loss and mAP graph in the results subsection below. The code for the Python script can be found in Appendix B.

5.1.2 Results

Iterations	TP	FP	FN	Precision	Recall	F1	Avg. IoU (%)	mAP (%)
400	3	24	205	0.11	0.01	0.03	7.16	2.73
500	114	195	94	0.37	0.55	0.44	22.98	33.06
600	80	66	128	0.55	0.38	0.45	34.19	38.7
700	147	138	61	0.52	0.71	0.6	33.43	54.94
800	115	64	93	0.64	0.55	0.59	41.7	51.16
900	120	287	88	0.29	0.58	0.39	18.86	23.02
1000	123	266	85	0.32	0.59	0.41	20.19	26.96
1100	152	75	56	0.67	0.73	0.7	44.76	67.46
1200	124	213	84	0.37	0.6	0.46	23.71	37.01
1300	131	106	77	0.55	0.63	0.59	36.42	43.73
1400	144	75	64	0.66	0.69	0.67	43.13	55.71
1500	126	110	82	0.53	0.61	0.57	36.05	44.65
1600	145	93	63	0.61	0.7	0.65	40.71	54.63
1700	137	69	71	0.67	0.66	0.66	44.72	57.71
1800	119	96	89	0.55	0.57	0.56	37.18	41.29
1900	78	330	130	0.19	0.38	0.25	12.17	11.25
2000	136	110	72	0.55	0.65	0.6	35.67	45.67
2100	139	85	69	0.62	0.67	0.64	40.96	50.71
2200	**158**	**45**	**50**	**0.78**	**0.76**	**0.77**	**51.52**	**69.84**
2300	144	87	64	0.62	0.69	0.66	40.85	52.84

TABLE 5.1: YOLO Training Results on Stairs Dataset

From the tabulated results above (Table 5.1), it can be observed that the model trained during iteration 2200 yielded the best overall performance in terms of detection accuracy with the highest number of true positives, highest precision scores, best average IoU percentage and an mAP score of 69.84%, which is better than the mAP score reported in [18] by Fast-RCNNs. However, the mAP score of the trained YOLO model is lower than the mAP score of the YOLO model of [37], at 80.81%. This was to be expected as [37] utilized the improved YOLO v3 model for training their stairs detection model while also training their model with significantly more iterations (8650 iteration) as compared to our model. As stated in Chapter 3, the use of the older YOLO v2 model for training in our project was due to its slightly faster training speed as compared to YOLO v3, which was needed considering the GPU of our development machine was not very powerful. Figure 5.2 below illustrates the training loss and mAP graph generated after training.

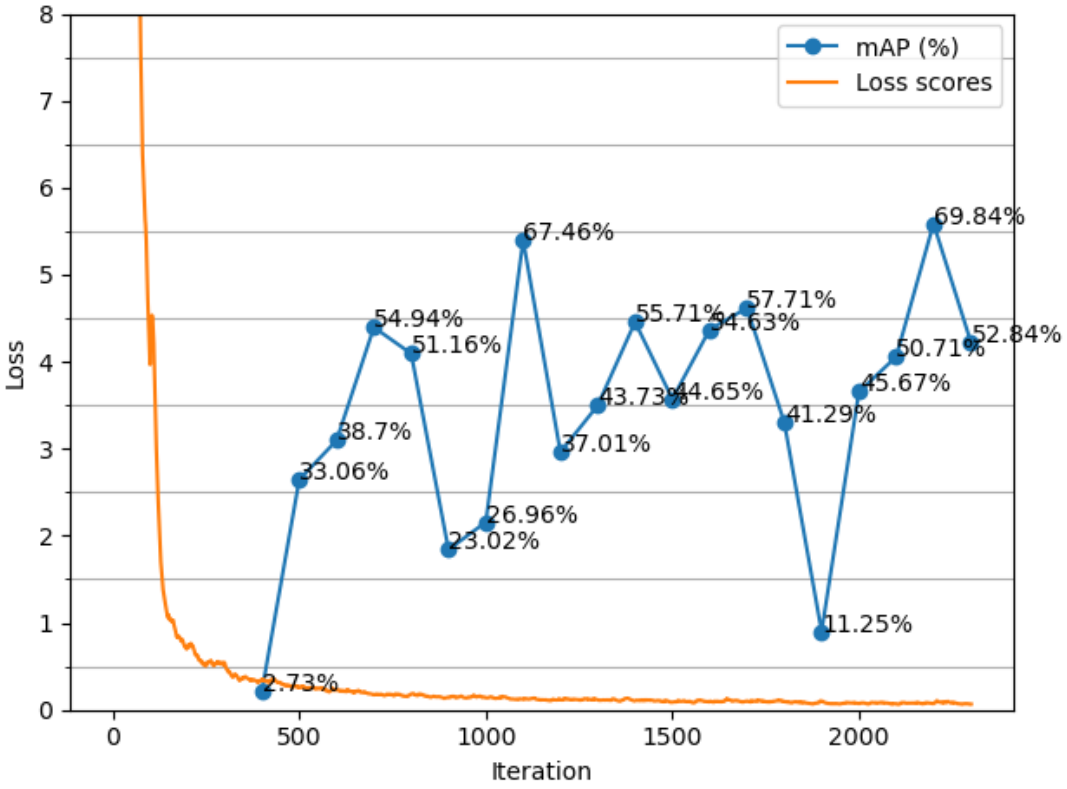


FIGURE 5.2: YOLO Model Training Loss and mAP % Graph

5.2 Stair Step Counting

5.2.1 Testing

The step counting algorithm was tested on 104 stairs images, which was obtained from a subset of the validation test images used by YOLO training phase. The data augmented versions (flipped horizontally) of each stair image was removed, reducing the initial test set from 208 images to 104. Each of the 104 stair image was accompanied by its belonging text file containing its YOLO annotated bounding box coordinates and a manually annotated stair step count label. Figure below shows the sample text file for one of the test stair image.

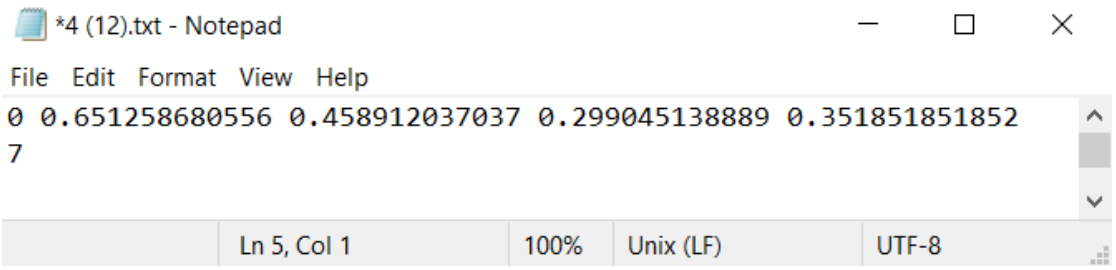


FIGURE 5.3: Text File Content for Each Stair Image File

The first line of each text file contains the following information in order: class label, x coordinate object center, y coordinate of object center, width of bounding box (w) and height of bounding box (h). The coordinates and bounding box dimensions were normalized to the image resolution scale, and can be converted back to its original values with the following equations:

$$w, h = (boxWidth * imageWidth), (boxHeight * imageHeight) \tag{5.1}$$

$$X_{center}, Y_{center} = (x * imageWidth), (y * imageHeight) \tag{5.2}$$

$$TopLeft(x, y) = (X_{center} - (w/2), Y_{center} - (h/2)) \tag{5.3}$$

$$BottomRight(x, y) = (X_{center} - (w/2), Y_{center} - (h/2)) \tag{5.4}$$

The second line of each text file denotes the manually annotated step count label, which will be used to compare against the detected step count by the algorithm to calculate its resulting accuracy. Once the text file information have been extracted, the algorithm will be performed for each test image and the following metrics will be used to assess the overall performance of the step counting algorithm:

1. **Successful Detections**: Indicates the number of times the algorithm has successfully detected steps to count.
2. **False Positives**: Indicates the number of times the algorithm counted steps that exceeded the actual number of steps in the image.
3. **False Negatives**: Indicates the number of times the algorithm counted steps that was less than the actual number of steps in the image.

4. **Correct Predictions**: Indicates the number of times the algorithm correctly counted the number of steps in the image.
5. **Average FP steps**: Indicates the average amount of steps the algorithm exceeded in counting per FP case.
6. **Average FN steps**: Indicates the average amount of steps the algorithm counted less per FN case.
7. **Average Detection Rate**: Calculates the average rate of detection for all successful detections, including the false positives. The detection rate for a single test instance was calculated as : *(Detected Steps / Actual Steps) * 100*
8. **Final Accuracy**: Calculates the average detection rate of the algorithm for only the correct predictions and false negative predictions, disregarding false positive cases.

The tests were performed through a Python script that includes the text file information extraction process, the algorithm execution and metrics calculation for all test images. The code for the test script can be found in Appendix B.

5.2.2 Results

Step Counting Algorithm Accuracy Metrics	
Successful Detections	98 / 104
Number of FP Cases	48
Number of FN Cases	30
Number of Correct Detections	20
Average FP Steps	5.42
Average FN Steps	5.47
Average Detection Rate	146.86%
Final Accuracy	**71.47%**

TABLE 5.2: Metric Results for Step Counting Algorithm

From the results presented in table 5.2, it can be observed that the algorithm managed to achieve a step counting accuracy of 71.47%, meeting the functional requirement for the project. Of the 104 test images when conducting the test, 98 of them have been successfully executed. The remaining 6 images failed due to the algorithm being unable to extract any horizontal lines within the image, which will be explained in detail in Chapter 6. However, the algorithm displayed a high tendency of over-detecting the number of steps in an image, with false positive cases occurring nearly half of all test instances. This caused the overall average detection rate for the algorithm, when false positives

are included, to exceed the actual total steps in all test images by 46%. Therefore, a final normalized accuracy measurement was done by discounting all false positive cases exhibited by the algorithm. For false positive cases, the algorithm detected an average of 5.42 extra steps within an image while an average of 5.47 less steps were detected on average for false negative cases.

Chapter 6

Discussion and Conclusions

This section will detail the obtained results and its context regarding certain aspects of the testing phase for the evaluation process discussed in the previous chapter.

6.1 Discussion

6.1.1 Stairs Detection with YOLO

The stairs detection model performed reasonably well with mAP scores of almost 70%, given the amount of iterations it took for training (2300) as compared to works of [37], where their model was trained for upwards of 10,000 iterations. The detection speed of the trained YOLO model using Darknet was also acceptably fast at 50 milliseconds on an MX250 GPU. However, given enough time and resources, a re-training of the model would be done with an increased number of training iterations as well as utilization of a better GPU. The model would also be changed to the YOLO v3 model as it provides a higher accuracy compared to YOLO v2 [22]. Not only will these changes improve the performance of the stairs detection module, but will also significantly reduce training time.

However, due to the trained YOLO model being implemented with OpenCV, as it was to be deployed into the HoloLens environment, the prediction time was significantly reduced as compared to vanilla Darknet execution time. This is due to the fact that the YOLO Darknet implementation of the "dnn" module in OpenCV did not support processing with GPU. This has caused the proposed system to perform with extremely poor frame rates, even without the inclusion of the step counting algorithm implementation. Fortunately, this issue did not interfere with the functional requirements of our project.

However, it did compromise one of the non-functional requirements in the project, which is to have the system operate at a frame rate of at least 5 fps.

Further research has been done to attempt to resolve this problem. A YOLO object detection HoloLens project [39] was implemented while being able to run at nearly real time speeds. This was achieved by having the HoloLens physical device tethered remotely to a desktop machine, streaming the video feed of the HoloLens device to the desktop. This allowed the desktop with higher processing capabilities to execute the YOLO object detection operations, thus achieving real time speeds. Unfortunately, this could not be replicated for our project as the physical HoloLens 2 device is not currently available and the emulator does not support content streaming as of yet.

6.1.2 Step Counting Algorithm

As seen in Chapter 5, the proposed step counting algorithm for the system managed to achieve a final accuracy of 71.47%, which met the functional requirement for this project with at least 70% step detection rate for step counting. However, after testing phase, it can be seen from the results that the algorithm has a rather high tendency to over-detect or under-detect the number of steps within an image, accounting for nearly half of all test instances during testing.

Upon further investigation, it was found that cases of over-detection and under-detection can be attributed to a combination of 3 main causes. Each cause will be explained in detail in its respective subsection.

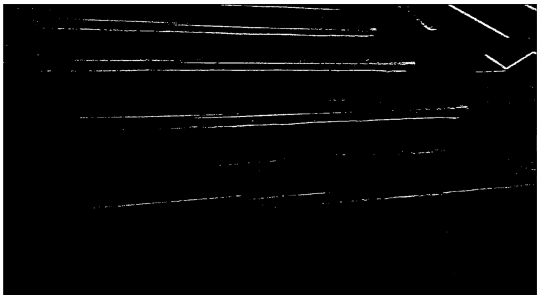
6.1.2.1 Characteristics of Input Images

The image dataset used for step counting tests comprises of 104 stair images. These images exhibit different characteristics as they were all subjected to different lighting conditions, sizes, distance and skew angles, which is a realistic recreation of inputs received by the algorithm during individual video frame processing in the final proposed system. With that said, certain characteristics above have contributed to the false positives or false negatives detection of the algorithm.

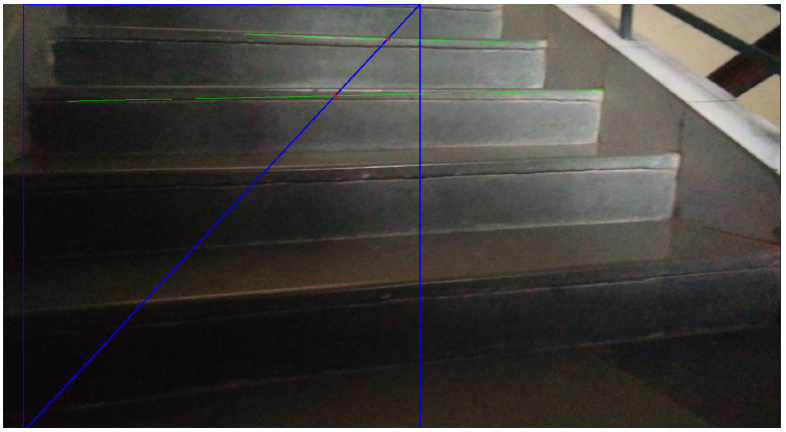
For images with low lighting conditions, under-detection of steps were likely to occur as the lower contrast of the stairs within the image makes it hard for the edge segmentation process of the algorithm to segment horizontal edges. This is because the low contrast blends the edges with the background, blurring any distinguishable edge lines when subjected to the Sobel operator. The lack of segmented horizontal edges would then yield low line counts after subjection to Probabilistic Hough Line Transform, which

in turn causes an under-detected (false negative) step count. Figure 6.1 illustrates an example of the algorithm processing a low contrast input image, causing the algorithm to only detect 2 out of the 4 total steps within the image.

(a) Low Contrast Input Image



(b) Edge Segmentation

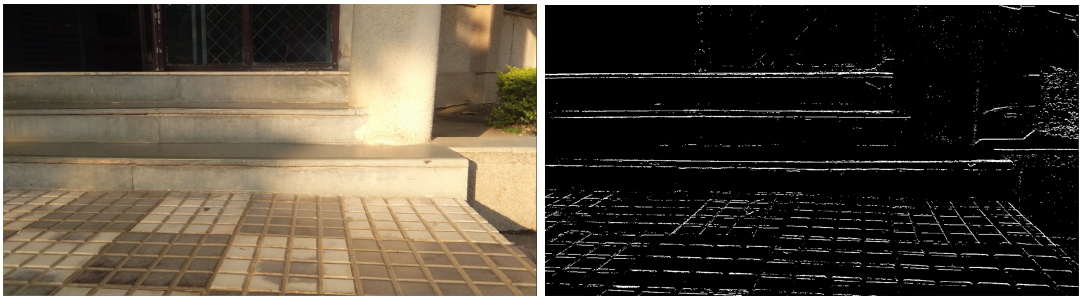


(c) Final Result (Step Under-Detection)

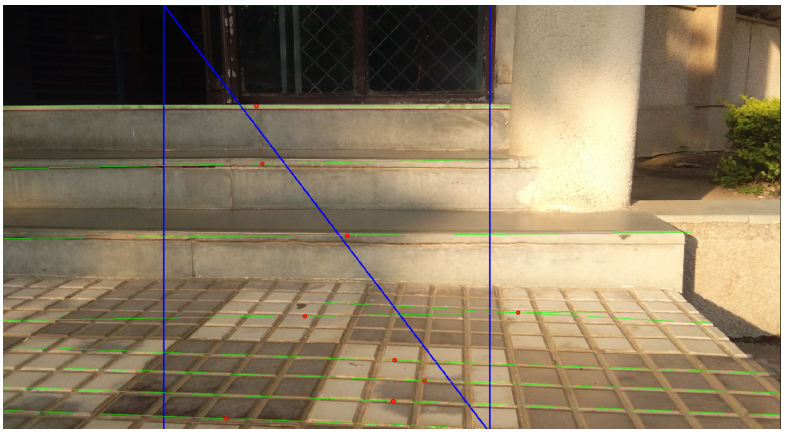
FIGURE 6.1: Low Contrast Image Causing Under-Detection of Steps

From the figure above, it can be observed that only the steps in the top part of the stairs were detected, as the brightness and contrast in that portion was relatively good. However, the bottom part of the stairs with lower levels of lighting yielded terrible edge segmentation results, thus yielded no detection for the first two steps of the stairs in the final result.

Cases with false positives (over-detection) on the other hand, usually involves stairs images that have various non-stair elements populating the surrounding areas that exhibit horizontal edges as well. These elements may include features or objects such as railings, banisters, floor patterns or even shadows, that have caused the algorithm to include their exhibited horizontal edges to be classified as a step. Figure 6.2 below shows an example of a stair input image with over-detected steps due to miss-classified non-step elements, which in this case, is the floor pattern. The step counting algorithm yielded a step detection count of 9 for the example in the figure below, whereas the actual step count was actually 3.



(a) Input Image (b) Edge Segmentation



(c) Final Result with Over-detection

FIGURE 6.2: Horizontal Floor Patterns Causing Over-Detection of Steps

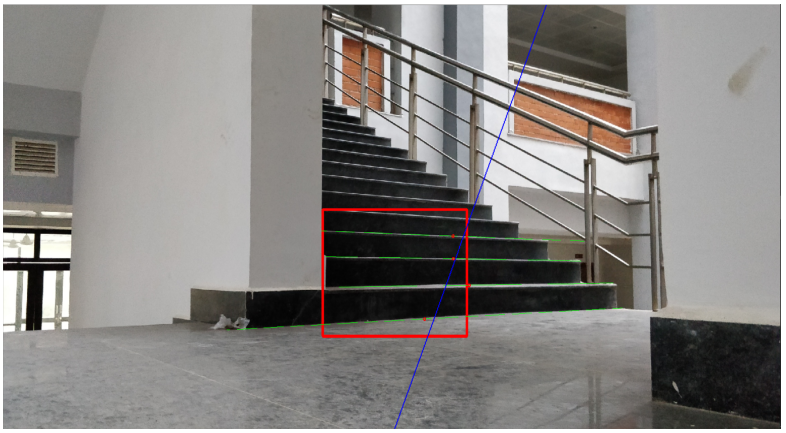
It was observed in Figure 6.2 that the checkered floor pattern in the image was also segmented during edge segmentation. The segmented edges of the floor was then wrongly assumed to be a valid step by the algorithm after the newly constructed bounding box have included these non-step horizontal lines, therefore returning an over-detection in the final result.

6.1.2.2 Erroneous Angle Estimation

The step counting algorithm relies on the Probablistic Hough Transform method to not only determine the horizontal lines present within the image, but to also provide the length of each detected horizontal lines. However, due to the probabilistic nature of the method, the length provided is not very consistent or accurate as it is very sensitive to noise or obstructions. With inconsistent line lengths provided, the centroids obtained for each stair step line will not be aligned to the actual inclination angle of the flight of stairs in the image. This will cause a misrepresentation of the estimated angle derived from the line of best fit algorithm, which uses the centroids as basis for estimation.

Another contributing factor to erroneous angle estimation besides misaligned centroids are the bounding box dimensions returned by the YOLO stairs detection module. The trained YOLO model for our proposed system returns a bounding box that does not envelope the entirety of the detected stairs. Instead, it usually returns a smaller bounding box to represent detected stairs at the base of the stairwell. If the dimensions of the bounding box is too small, naturally, fewer amount of steps would be contained within the box. A low amount of steps within the box indicate low amount of line centroids. This in turn leads to inaccurate angle estimation as the line of best fit algorithm will be more volatile in angles with lesser centroids present to be used as basis for estimation. Figure 6.3 illustrates the erroneous angle estimated by the line of best fit algorithm due to low amount of centroids present within the given YOLO bounding box.

(a) Small YOLO Bounding Box



(b) Erroneous Predicted Angle (Blue Line)

FIGURE 6.3: Erroneous estimated angle due to small YOLO bounding box

An erroneous angle estimation would severely affect the results of the step counting algorithm as the line of best fit would determine the dimensions of the new bounding box of the image, which would be further explained in the next subsection.

6.1.2.3 Erroneous Bounding Box Construction

As mentioned earlier, an erroneous angle estimation would eventually lead to an erroneous bounding box construction. An erroneous bounding box might either lead to severe over-detection or severe under-detection, as the surface area of the bounding box is directly correlated to the detection criteria of the algorithm. For example, a bounding box that encompass 50% of the width of the image will yield step counts that derive from lines that are only in contact with the newly constructed bounding box. Lines that are not in contact with the bounding box will not be deemed as a valid step. However, an erroneous bounding box that encompass the entirety of the image due to erroneous angle estimation will cause the algorithm to consider every horizontal line within the image to be a valid step, leading to massive over-estimation of step count. Figure 6.4 illustrates a constructed bounding box that encompassed too large of a portion of the input image that ultimately led to over-detection. Figure 6.5 on the other hand, illustrates a constructed bounding box that encompassed too small of a portion in the input image that eventually led to under-detection.

FIGURE 6.4: Bounding Box Encompassing Entire Image

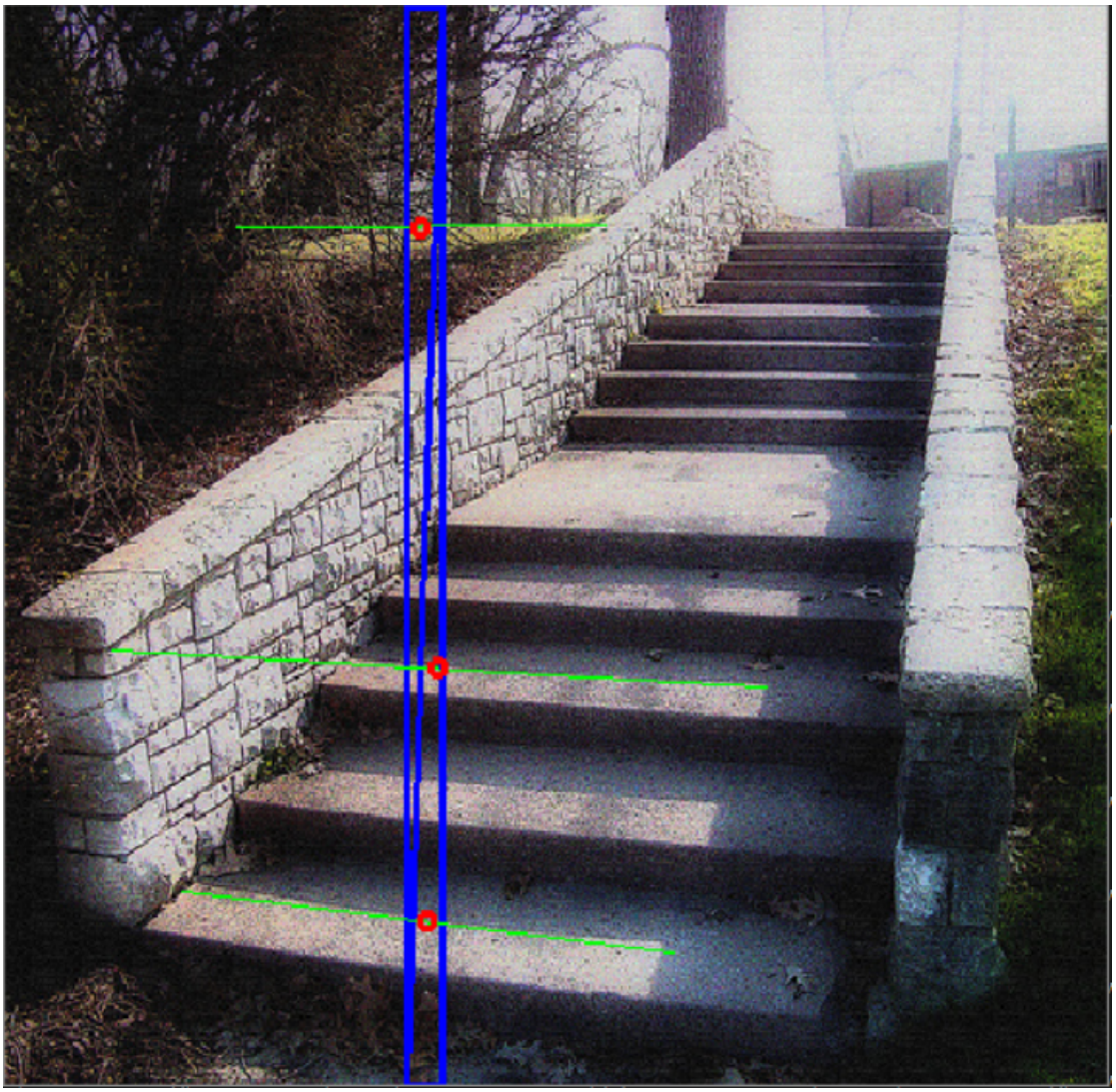


FIGURE 6.5: Bounding Box Encompassing Narrow Portion of Image

6.1.2.4 Summary

In short, the step counting algorithm accuracy was ultimately marred by the edge segmentation algorithm for not being able to discern horizontal edges more effectively and the Probabilistic Hough Line Transform algorithm for not being able to supply with more consistent horizontal line detection results. The two stages above have then caused a snowball effect for the rest of the operations that led to further complications such as angle estimation errors and bounding box construction errors, which eventually led to cases of false positives and false negatives by the step counting algorithm.

6.2 Conclusion

A visual aid system involving stairwell detection and step counting was proposed in this project to ease navigational tasks for users suffering from glaucoma. Stair detection accuracy of nearly 70% mAP was achieved using a trained YOLO v2 model, along with a step counting accuracy of 71.64% for the detected stairs. A Y-axis Sobel operator filter and Probabilistic Hough Line Transform was used to segment and isolate stair step edges. Line merging and stair angle estimation using line centroids were applied to detect and count the stair steps. The stairs detection and step counting algorithm was integrated into a Unity application and was then deployed into a HoloLens emulator to simulate the physical AR device. A 360° video was implemented to simulate real life environments with the emulator and the proposed system was able to detect and count stairs steps within the emulator during simulated motion and traversal. However, the system was unable to perform near real time speeds due to limitations to the image processing library used for initiating YOLO stairs detection.

All functional requirements were met in the final proposed system for this project, meeting the minimum target of 60% mAP for stairs detection by our trained model and achieving more than the targeted 70% accuracy rate for step counting. Some non-functional requirements, however, were unable to be accomplished mainly due to time constraints and the current pandemic situation (COVID-19). The proposed system ultimately was unable to perform at the expected frame rate of at least 5fps, and the color correction module of the system was unable to be implemented in time for the deadline of this project. The step counting algorithm was not included for implementation in the Unity script as well, due to time constraints and the severe performance toll it would incur to the already sluggish system. However, the system proved capable in stairs detection during traversal as simulated by the movements in the 360° video feed, albeit at a slow frame rate. The system is also fully deploy-able and operatable in the HoloLens 2 emulator environment, which translates to a highly likely scenario that it would work well with the physical device as well.

Overall, the project managed to accomplish its functional requirements while presenting itself with many possible future enhancements to be done in order to elevate this project to be truly usable as a navigational aid for users suffering from glaucoma. However, the objective of running the system at reasonable frame rates with the inclusion of the stairs detection and step counting module was unable to be reached given the current time constraints and pandemic situation.

6.3 Future Work

The following changes and tasks should be performed in the future as enhancement to the overall proposed project:

1. Major restructuring of the overall system architecture.
2. Re-train YOLO model with updated YOLO iteration.
3. Rework step counting algorithm.
4. Color/Visual correction module implementation.

Major restructuring of the system architecture would have to be done in the future to accommodate the transition of running the system from an emulator environment to a physical HoloLens 2 device. With the migration to the physical device, computationally taxing components of the system such as the YOLO object detection module and the step counting module could be performed on a remote machine through wireless tethering instead of the HoloLens device itself, which would allow the deployed system to run in real time speed. An updated version of the YOLO (v3) detection model will be used in the future and will be trained for a longer period of time to further increase the accuracy of the stairs detection module. Also, with the depth sensing capabilities of the physical HoloLens 2 device, the step counting algorithm could be reworked by using the spatial information provided by the on-board depth sensors of the HoloLens 2 to detect individual steps on a flight of stairs and using that to achieve a more effective step segmentation result, thus obtaining a more accurate step count after step detection. Lastly, color correction module could be implemented and applied to increase visibility of the detected flights of stairs for users with glaucoma. This would then truly make the proposed system into a viable navigational aid for glaucoma users, as the detection module could also be trained to detect other potentially hazardous indoor objects, further enhancing usability and practicality of the system as a navigational tool for the visually impaired.

Bibliography

[1] Blindness and vision impairment. [Online]. Available: https://web.archive.org/web/20150429145832/http://www.cdc.gov/healthcommunication/toolstemplates/entertainmented/tips/blindness.html

[2] Visual impairment and blindness - fact sheet no.282. [Online]. Available: http://www.who.int/mediacentre/factsheets/fs282/en/

[3] Hololens research mode. [Online]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/research-mode

[4] R. R. A. Bourne, H. R. Taylor, S. R. Flaxman, J. Keeffe, J. Leasher, K. Naidoo, K. Pesudovs, R. A. White, T. Y. Wong, S. Resnikoff, and et al., "Number of people blind or visually impaired by glaucoma worldwide and in world regions 1990 – 2010: A meta-analysis," *Plos One*, vol. 11, no. 10, 2016.

[5] Types of glaucoma diseases. [Online]. Available: https://www.glaucoma.org/glaucoma/types-of-glaucoma.php

[6] C. X. Hu, C. Zangalli, M. Hsieh, L. Gupta, A. L. Williams, J. Richman, and G. L. Spaeth, "What do patients with glaucoma see? visual symptoms reported by patients with glaucoma," *The American Journal of the Medical Sciences*, vol. 348, no. 5, p. 403–409, 2014.

[7] P. Costa, H. Fernandes, P. Martins, J. Barroso, and L. J. Hadjileontiadis, "Obstacle detection using stereo imaging to assist the navigation of visually impaired people," *Procedia Computer Science*, vol. 14, pp. 83–93, 2012.

[8] H.-C. Wang, R. K. Katzschmann, S. Teng, B. Araki, L. Giarré, and D. Rus, "Enabling independent navigation for visually impaired people through a wearable vision-based feedback system," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 6533–6540.

[9] O. Younis, W. Al-Nuaimy, M. H. Alomari, and F. Rowe, "A hazard detection and tracking system for people with peripheral vision loss using smart glasses and augmented reality," *Int. J. Adv. Comput. Sci. Appl*, vol. 10, pp. 1–9, 2019.

[10] J. Sudol, O. Dialameh, C. Blanchard, and T. Dorcey, "Looktel—a comprehensive platform for computer-aided visual assistance," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition-Workshops*. IEEE, 2010, pp. 73–80.

[11] H. Jabnoun, F. Benzarti, and H. Amiri, "Object detection and identification for blind people in video scene," in *2015 15th International Conference on Intelligent Systems Design and Applications (ISDA)*. IEEE, 2015, pp. 363–367.

[12] R. Chincha and Y. Tian, "Finding objects for blind people based on surf features," in *2011 IEEE International Conference on Bioinformatics and Biomedicine Workshops (BIBMW)*. IEEE, 2011, pp. 526–527.

[13] R. Jafri, S. A. Ali, H. R. Arabnia, and S. Fatima, "Computer vision-based object recognition for the visually impaired in an indoors environment: a survey," *The Visual Computer*, vol. 30, no. 11, pp. 1197–1222, 2014.

[14] A. D. Hwang and E. Peli, "An augmented-reality edge enhancement application for google glass," *Optometry and vision science: official publication of the American Academy of Optometry*, vol. 91, no. 8, p. 1021, 2014.

[15] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *CoRR*, vol. abs/1311.2524, 2013. [Online]. Available: http://arxiv.org/abs/1311.2524

[16] J. R. R. Uijlings, K. E. A. V. D. Sande, T. Gevers, and A. W. M. Smeulders, "Selective search for object recognition," *International Journal of Computer Vision*, vol. 104, no. 2, p. 154–171, Feb 2013.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[18] R. B. Girshick, "Fast R-CNN," *CoRR*, vol. abs/1504.08083, 2015. [Online]. Available: http://arxiv.org/abs/1504.08083

[19] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[20] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[21] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[22] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *CoRR*, vol. abs/1804.02767, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[23] S. Se and M. Brady, "Vision-based detection of kerbs and steps," in *BMVC*, 1997.

[24] S. Se and M. Brady, "Vision-based detection of stair-cases," 2000.

[25] D. C. Hernandez and K.-H. Jo, "Outdoor stairway segmentation using vertical vanishing point and directional filter," *International Forum on Strategic Technology 2010*, 2010.

[26] D. C. Hernandez and K.-H. Jo, "Stairway segmentation using gabor filter and vanishing point," *2011 IEEE International Conference on Mechatronics and Automation*, 2011.

[27] Y. H. Lee, T.-S. Leung, and G. Medioni, "Real-time staircase detection from a wearable stereo system," 01 2012, pp. 3770–3773.

[28] T. Schwarze and Z. Zhong, "Stair detection and tracking from egocentric stereo vision," *2015 IEEE International Conference on Image Processing (ICIP)*, 2015.

[29] W. Lee and W. Woo, "Real-time color correction for marker-based augmented reality applications," in *International Workshop on Ubiquitous Virtual Reality*, 2009, pp. 32–25.

[30] E. Tanuwidjaja, D. Huynh, K. Koa, C. Nguyen, C. Shao, P. Torbett, C. Emmenegger, and N. Weibel, "Chroma: a wearable augmented-reality solution for color blindness," in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2014, pp. 799–810.

[31] B. S. Ananto, R. F. Sari, and R. Harwahyu, "Color transformation for color blind compensation on augmented reality system," in *2011 International Conference on User Science and Engineering (i-USEr)*. IEEE, 2011, pp. 129–134.

[32] S. Carduner. Patient's guide to living with glaucoma. [Online]. Available: https://visionaware.org/your-eye-condition/glaucoma/patients-guide-to-living-with-glaucoma/125/

[33] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[34] J. Weimann, "How to play stereoscopic 3d 360 video in vr with unity3d," https://unity3d.college/2017/07/31/how-to-play-stereoscopic-3d-360-video-in-vr-with-unity3d/, 2017.

[35] Unity-Technologies, "Panoramic 2d and 3d video shader for unity," https://github.com/Unity-Technologies/SkyboxPanoramicShader, 2017.

[36] Firifire, "Opencvsharp for *UWP* applications," https://github.com/Firifire/OpenCvSharp-UWP/releases, 2018.

[37] U. Patil, A. Gujarathi, A. Kulkarni, A. Jain, L. Malke, R. Tekade, K. Paigwar, and P. Chaturvedi, "Deep learning based stair detection and statistical image filtering for autonomous stair climbing," *2019 Third IEEE International Conference on Robotic Computing (IRC)*, 2019.

[38] P. Bourke, "Minimum distance between a point and a line," http://paulbourke.net/geometry/pointlineplane/, 1988.

[39] doughtmw. Yolo detection holo lens unity. [Online]. Available: https://unitylist.com/p/sp8/Yolo-Detection-Holo-Lens-Unity