SE4RP11: Omnidirectional Vision Platform Using ASEF Filters

Scott Alexander
s.alexander@student.reading.ac.uk
Cybernetics MEng
School of Systems Engineering, The University of Reading, UK

Abstract

Computer vision is currently a growing research area due to recent advances in computational power, however, there are still problems with depth and scaling of key features in images. Therefore an omnidirectional vision platform has been developed to tackle these problems on a mobile vehicle, as depth and scaling are key features required by a mobile platform for object detection. The development and design of an omnidirectional platform is addressed for when camera parameters are unknown, producing an adjustable platform. To provide images that can return the distance and the angle of an object of interest, the unwrapping of an omnidirectional image is also addressed, along with the process of inverse thresholding using the contrasts of different environmental features. Two vision-detecting techniques have been applied in the form of Connect Component Labeling (CCL) and Average of Synthetic Exact Filters (ASEF), both of which have shown significant results in detecting objects in a controlled environment, while addressing the speed in which objects can be detected.

1. Table of Contents

2. Introduction

In recent years, computer vision based processes for the purpose of tracking and identifying objects have become popular, primarily due to increases in the computational power of machines, which has allowed video streams to be analysed in real time. Such applications include baggage monitoring in airports, to detect unattended items [1] and human tracking, to monitor the movement of individuals throughout buildings [2]. Many processes have been developed for the purpose of identification and tracking, including Scale-Invariant Feature Transform (SIFT) [3], Connected Component Labelling (CCL) [4] and Histogram of Oriented Gradients (HOG) [5]. However, the biggest problem that vision research is facing is dealing with the scale and depth of key features within images, as it either requires knowledge of the environment or multiple trained feature vectors for an object at different scales, a process that slows down the process of identification considerably.

One field of research that could benefit greatly from vision processes is multi agent robot systems or swarms [6]. This would allow each robot to explore their environment and build up a vision-based map between them, while also searching for objects that can be detected and examined in a human centric manner. However, a mobile robot needs depth information to calculate the distance between itself and an object of interest, if it requires interaction with the object. Therefore two methods have previously been created to provide this depth information. The first method involves using two stereo cameras that have been calibrated with a set distance between them. This allows trigonometry to be used between pixel points in the two images to calculate the distance an object is away from the central point of the cameras [7]. However, this method is computationally extensive which is not particularly suitable for a mobile robot application, as it requires simple algorithms to operate adequately while keeping costs down in the processing region. The second method uses a single camera and a conical lens directly above its focal point [8]. This provides a 360-degree vision range, which is dependent on the lens angle to the horizon and the height at which it is placed. This method is more suitable to a mobile robot application as it is computationally simple.

Currently ground based mobile swarms are still a research area and have not been implemented in a real world application; some suggested uses include cleaning systems inside large buildings and search and rescue inside dangerous environments. However, swarm applications become more suitable to real world situations when the mobile units become aerial or water based, as the units can move in three dimensions eliminating restrictions such as steps. This is particularly useful for military surveillance, as demonstrated by the GRASP Lab at the University of Pennsylvania with their Quadrotor Unmanned Aerial Vehicle grid application [9], or exploring ocean ecologies as demonstrated by Arizona State Universities Biodesign department with Sensorbot [10]. However, the majority of these applications started out as ground based robots, which demonstrates they are a suitable source for research and experimentation.

Through the omnidirectional method, any vision-based process can be applied to the acquired image, however it needs to be fast on a mobile application to allow a robot to interact with its environment in real time. Therefore the aim of this project is to explore

fast and accurate object detection methods on a single omnidirectional mobile platform that could potentially operate in real time, while maintaining costs at less that £100.

3. Social, Legal, Ethical, Health and Safety Issues

Through the specification of the project, issues became apparent in the realm of health and safety, ethics and social concerns. The safety issues were as follows:

- The robots will be operating on the floor which means they are a trip hazard, therefore the robots will only operate within a blocked off testing area.
- Any components that require powering from a mains supply will have the plug labelled clearly for each component to prevent incorrect voltages being supplied to incorrect components.
- All components and wires will be stored tidily on board the robot to prevent shorts that could potentially cause a fire.

Issues that became apparent in the social, legal and ethical areas were:

- Any off the shelf software that is required must have a licence purchased.
- Permission for any public code will be gained prior to use unless it is under an open source licence. All public code, whether copyrighted or not, will be acknowledged accordingly.
- The mobile robot will be used within UK legislation and will not be used to inflict harm or violate an individual's privacy.
- If for any reason an individual is filmed and used in any test data, written consent must be acquired from the individual prior to any filming.
- If for any reason sensitive material is filmed, it shall be kept in a secure manner with the owner's permission.

4. Project Management

4.1 Detailed Specification

Through initial research and discussions with a project supervisor, the following objectives were discussed and agreed to be completed over the course of twenty weeks. These objectives were as follows:

- Develop and implement a hardware system that can transmit visual data wirelessly to a PC-terminal.
- Develop an omnidirectional vision platform on top of a mobile robot.
- Implement an environmental blob detection algorithm that can isolate an object of interest.
- To develop a series of ASEF filters to identify simple objects in a simple environment.

- To implement a distance algorithm based on the closest returned point of a detected object.

4.2 Task Allocation

Each objective was broken down into the key tasks that were necessary to achieve the aims and were scheduled over the 20 weeks, which can be seen in Figure 1 and Figure 2.

Task Name	Duration	Start	Finish
(2) Omnidirectional Platform	30 days	Mon 10/10/11	Fri 18/11/11
(3) Hardware Selection	5 days	Mon 10/10/11	Fri 14/10/11
(4) Platform Design	15 days	Mon 17/10/11	Fri 04/11/11
(5) Platform Construction	11 days	Sun 06/11/11	Fri 18/11/11
(6) Blob Detection Algorithm	45 days	Mon 21/11/11	Fri 20/01/12
(7) Base GUI Design	4 days	Mon 21/11/11	Thu 24/11/11
(8) GUI Construction	4 days	Thu 24/11/11	Tue 29/11/11
(9) Unwrapping Algorithm	6 days	Tue 29/11/11	Tue 06/12/11
(10) Reverse Threshold Algorithm	3 days	Wed 07/12/11	Fri 09/12/11
(11) Research CCL Libraries	5 days	Mon 12/12/11	Fri 16/12/11
(12) Implement CCL	5 days	Mon 16/01/12	Fri 20/01/12
(13) ASEF Implementation	33 days	Mon 23/01/12	Wed 07/03/12
(14) ASEF Research	4 days	Mon 23/01/12	Thu 26/01/12
(15) ASEF Code Design	6 days	Fri 27/01/12	Fri 03/02/12
(16) ASEF Code Development	10 days	Mon 06/02/12	Fri 17/02/12
(17) Data Gathering	3 days	Mon 20/02/12	Wed 22/02/12
(18) Data Training	5 days	Thu 23/02/12	Wed 29/02/12
(19) ASEF GUI Design	2 days	Thu 01/03/12	Fri 02/03/12
(20) ASEF Testing	3 days	Mon 05/03/12	Wed 07/03/12
(21) Position and Control	16 days	Mon 12/03/12	Mon 02/04/12
(22) Implement Ultrasonics'	3 days	Mon 12/03/12	Wed 14/03/12
(23) Implement Tachometers	4 days	Thu 15/03/12	Tue 20/03/12
(24) Calibrate Distance Range	3 days	Wed 21/03/12	Fri 23/03/12
(25) Implement Movement control	3 days	Mon 26/03/12	Wed 28/03/12
(26) Implement Turning Control	2 days	Thu 29/03/12	Fri 30/03/12
(27)Implement distance algorithm	1 day	Mon 02/04/12	Mon 02/04/12

Figure 1. Gantt Time Sheet

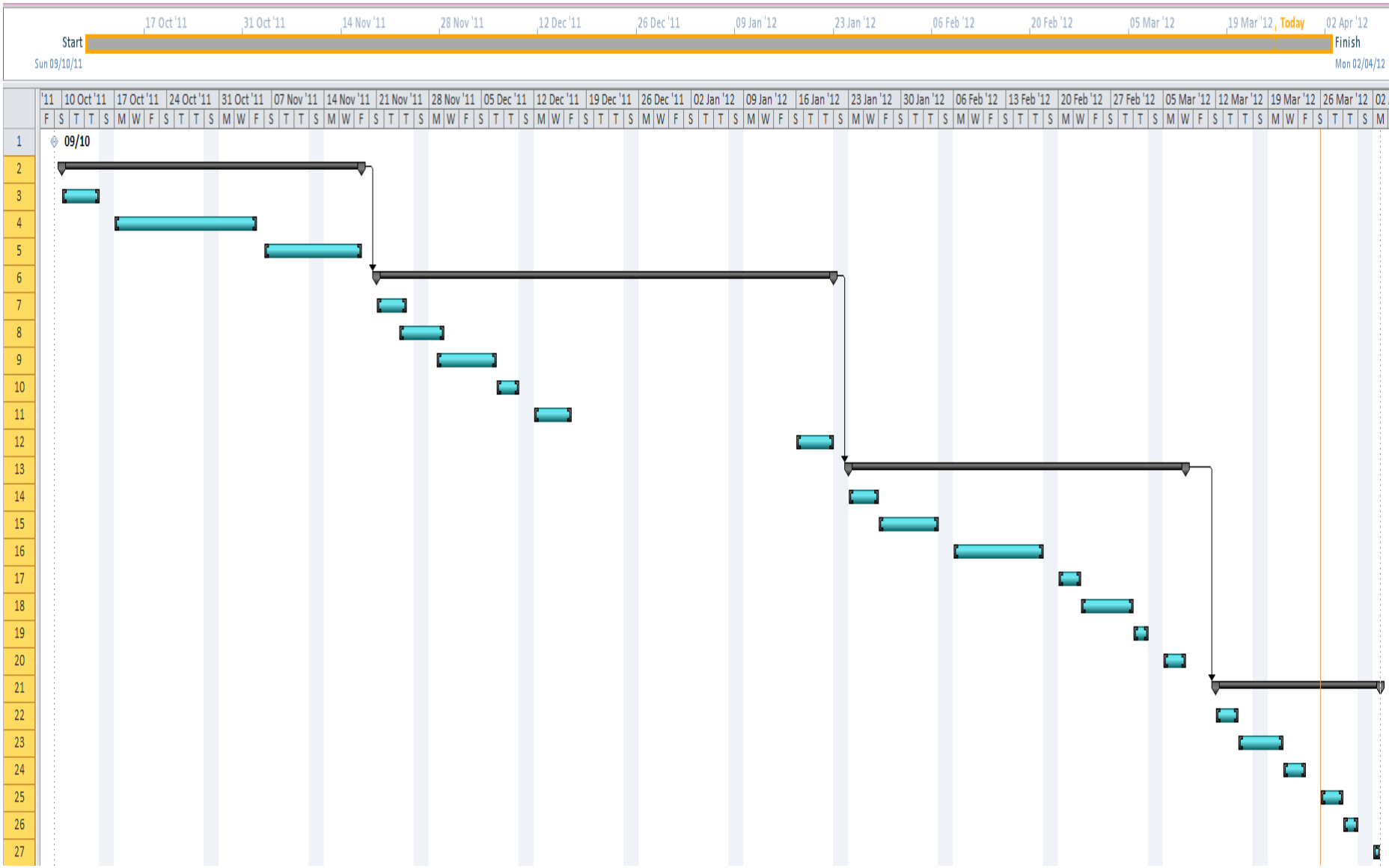


Figure 2. Gantt Chart

5. Research

5.1 Vision Research

5.1.1 Principles of Omnidirectional Vision

An omnidirectional platform consists of two key components: a camera and a mirrored fisheye, hyperbolic or conical lens, aligned perfectly so that the centre of the camera is directly underneath the central point of the lens. As a result, this can return a distorted 360-degree image of the cameras surrounding environment. For the purpose of this project a conical lens has been selected, primarily because it is the easiest type of lens to produce, which brings the cost of the project down but also because there are two key benefits over the other lens types as stated by Lin and Bajcsy [11];

1. "Curved cross section mirrors produce inevitable radial distortions. Radial distortion is proportional to the radial curvature of the mirror. We note simply that the cone has constant zero radial curvature everywhere except at its tip, which will only be reflecting the camera anyway."

2. "Radial curved mirrors produce 'fish eye' effects: they magnify the objects reflected in the center of the robot, or the sky, all of which are of minimal interest. On the other hand, they shrink the region around the horizon, thereby reducing the available spatial resolution in the area, which is of interest."

Libor Spacek [12] has gone further with this research and has concluded that the maximum useful value of the field of view angle φ of the camera is determined by (1).

$$\phi = 2 * arctan\frac{R}{R+d} \tag{1}$$

d is the distance between the lens and the camera and R is the radius of the lens. This model assumes that R is both the radius and height of the conical lens, which produces a 90-degree angle at the tip. However, a 90-degree tip is not always desired, as the tip angle affects the field of view distance range, as well as the minimum and maximum viewing distances forming the field of vision. This is because light will always reflect at the same angle it hits a flat reflective surface. The affect the angle of the lens to the horizon can have can be seen in Figure 3.

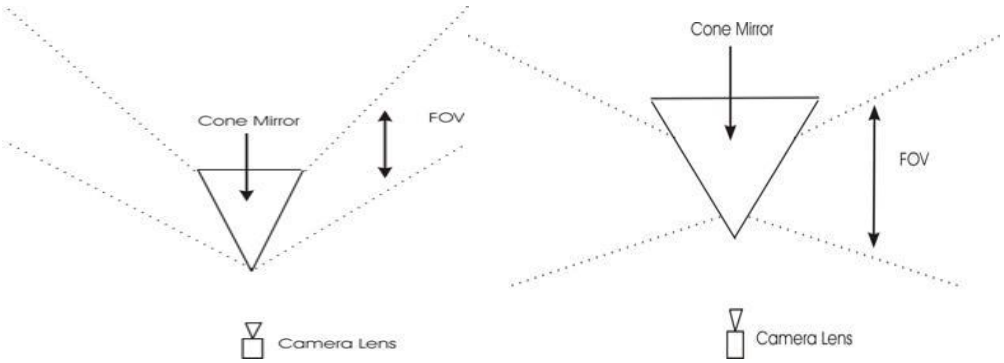


Figure 3. Effects of a large (left) and small (right) angle to the horizon on a conical lens.

If the lens angle to the horizon is too big, the image produced will be from above the lens. However, if the lens angle is to small, the range of the visible area will be too small and too close to the camera, therefore a suitable angle for the lens must be calculated to provide the required range of any system. The radius and height primarily determine the lens point angle to the horizon, but the height of the lens above the ground and the distance between the lens and the camera also play a key part in the minimum and maximum visible points. Fortunately, simple trigonometry can be used to calculate the distance range with different angled lenses as long as a camera is assumed to have a single focal point. The reflective angle of the image received from the lens is the same angle at which the light hits the lens, as the cone has constant zero radial curvature as stated by Lin and Bajcsy's first beneficial point. This is why the trigonometric calculations for a conical lens can be used.

However, by using trigonometry to calculate the angle of the lens to provide a specific distance range, the height of the lens from the ground as well as the distance between the lens and the camera become key calibration parameters.

5.1.2 Unwrapping an Omnidirectional Image from a Conical Lens

For some applications it is desired to unwrap the image of an omnidirectional system image into a panoramic image for vision processes. This is to remove the distortion of objects within the images so that the unwrapped version portrays the object of interest's actual shape.

Fortunately, due to the nature of a conical lens, vertical lines in the horizon maintain a straight edge, while horizontal lines are curved. This means that a circle can be broken up into triangular segments (Figure 4) in which all of the straight lines towards the centre of the lens are vertical lines in a rectangular unwrapped image. To produce the unwrapped image, the innermost section of the conical lens is stretched to meet the length of the outer most part. This does make the innermost part unusable due to stretching, however the resolution of the central part of the lens is already limited due to the surface area of the section and only reflects the image of the camera, which is not needed.

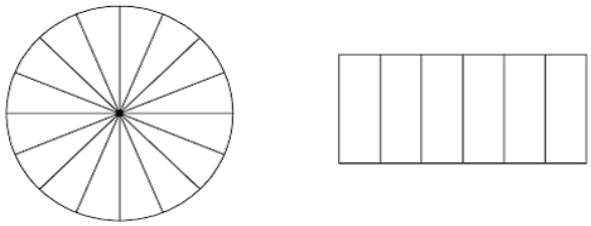


Figure 4. (Left) The lens broken up into triangular segments. (Right) The image after the central points of the lens have been stretched.

Computationally the process of unwrapping the image deploys a simple pixel repositioning process that relies on four parameters: the diameter, the central co-ordinates of the lens point x and y and a step value. The process takes the parameters into account by moving from the outside circumference to the central point of the lens, through steps along the radius. At each step, two half-circle curvatures are taken and

translated into a single straight row of pixels in the unwrapped image, with the size dependent on the implementer's choice. The size of the rows also determines how many points are taken from each half-circle, but is generally larger for bigger images to prevent blank sections in the unwrapped image, as they require a larger resolution.

The process has also been mathematically proven by Libor Spacek [12] by mapping the polar coordinates $(h_{i,}, \theta_i)$ of the input image to the rectangular image (x, y) coordinates through (2) and (3).

$$x = \frac{x_m}{2\pi} * \theta_i \tag{2}$$

$$y = \frac{y_m}{r_m} * h_i \tag{3}$$

Where (x_m, Y_m) are the required dimensions of the unwrapped image, r_m is the radius of the mirror in the input image and θ_i is measured in radians. The aspect ration of the unwrapped image is also calculated through (4).

$$\frac{x_m}{Y_m} = \frac{2\pi}{\sqrt{2}} \tag{4}$$

Then by using the polar coordinates as an intermediate step and using the center x and y coordinates of the lens (x_c, y_c), the image can be unwrapped through equations (5) and (6).

$$x_i = x_c + h_i * cos\theta_i = x_c + \frac{r_m}{y_m} * y * \cos(\frac{2\pi}{x_m} * x) \tag{5}$$

$$y_i = y_c + h_i * sin\theta_i = y_c + \frac{r_m}{y_m} * y * \sin(\frac{2\pi}{x_m} * x) \tag{6}$$

However, this is not the only method Spacek describes, he also shows how a two dimensional discrete cosine transform (DCT) can be applied to unwrap the image, which has advantages over the previous method as it produces more distinct edges. Firstly the inverse DCT can be seen in (7).

$$f(x_{i,}y_i) = \sum_{q=0}^{Q-1} \sum_{p=0}^{P-1} c(q,p).\cos\left(\frac{\pi q}{Y}(y_i + 0.5)\right) * \cos\left(\frac{\pi p}{X}(x_i + 0.5)\right) \tag{7}$$

In which $c(q,p)$ is the normalized coefficients array of dimensions (P,Q) acquired by a forward DCT process and (X,Y) are dimensions of the input image. Then by using equation (5) and (6) to replace x_i and y_i in the right hand side of (7) an unwrapping function is produced (8).

$$f(x_{i,}y_i) = \sum_{q=0}^{Q-1} \sum_{p=0}^{P-1} c(q,p).\cos\left(\frac{\pi q}{Y}\left(y_c + \frac{r_m}{y_m} * y * \sin(\frac{2\pi}{x_m} * x) + 0.5\right)\right) * \cos\left(\frac{\pi p}{X}\left(x_c + \frac{r_m}{y_m} * y * \cos(\frac{2\pi}{x_m} * x) + 0.5\right)\right) \tag{8}$$

Following on from the equations of the second DCT process, the unwrapping takes place by firstly running the DCT transformation on the conical lens image and the unwrapped image is then formed by creating pairs of co-ordinates through equation (8). This process may produce a better resolution compared to the pixel interpolation method, but as a transformation process is involved it takes longer and more computational power to produce the unwrapped image, which needs to be explored.

5.1.3 Blob Detection through Connected Component Labeling

There are two types of CCL methods, single pass and two-pass of which the later is the more accurate but slightly slower method. The two pass method involves going over an image pixel by pixel twice, the first time to record equivalences and to assign group labels to colour related pixels while the second pass finalises the group labels. Connectivity checks take place on each pass to check image pixels to the North, East, South and West of each individual pixel in an image in which four conditions are checked [13].

1. Does the West pixel have the same value?

- If yes then assign the same label to the current pixel.
- If no then check the next condition.

2. Do the pixels to the North and West of the current pixel have the same color intensity value but not the same label?

- If yes assign the current pixel the minimum label value of the North and West pixels. Then record the equivalence relationship between the North and West pixels.
- If no then check the next condition.

3. Does the pixel to the West have a different color intensity value and the North pixel the same intensity value?

- If yes then assign the label of the North pixel to the current pixel.
- If no then check the next condition.

4. Does the pixel's North and West pixels have different pixel values?

- If yes then create a new label ID and assign it to the current pixel

As a result, this produces a table of blobs assigned by label, where each pixel is also linked by equivalences. A fast method incorporates a raster-scanning algorithm to search through the image by column and then by row, as pixel access is faster this way since pointer iteration can be used as opposed to pointer repositioning, which can be processor intensive. To get the best results from this method, background removal techniques or background control methods can be used to eliminate any unwanted noise from the image prior to running algorithm.

5.1.4 Average of Synthetic Exact Filters

Average of Synthetic Filters (ASEF) [14] are simply made filters that rely on two key principles. Firstly for any given image and a second image of a response point where an object of interest is, there is an exact filter that will give the exact response when applied to the original image. The second is that through convolution theorem, convolution in he spatial domain is just element wise multiplication in the Fourier domain (9).

$$(I \otimes K)(x, y) = I'(w, v)K'(w, v) \tag{9}$$

Therefore, an image convolved with an exact filter will return the desired response if an object of interest is present in the exact same position as when the filter was created. Mathematically the creation of a perfect filter is shown in (10), (11) and (12) where G is the response image, I is the original image and K is the perfect filter.

In the spatial domain:

$$(I \otimes K)(x, y) = G(x, y) \tag{10}$$

In the Fourier domain:

$$I'(w, v)K'(w, v) = G'(w, v) \tag{11}$$

So if G and I are known:

$$K'(w, v) = \frac{G'(w, v)}{I'(w, v)} \tag{12}$$

This means that a simple division in the Fourier domain creates an exact filter. Figure 8 shows an example of this applied for eye detection purposes.

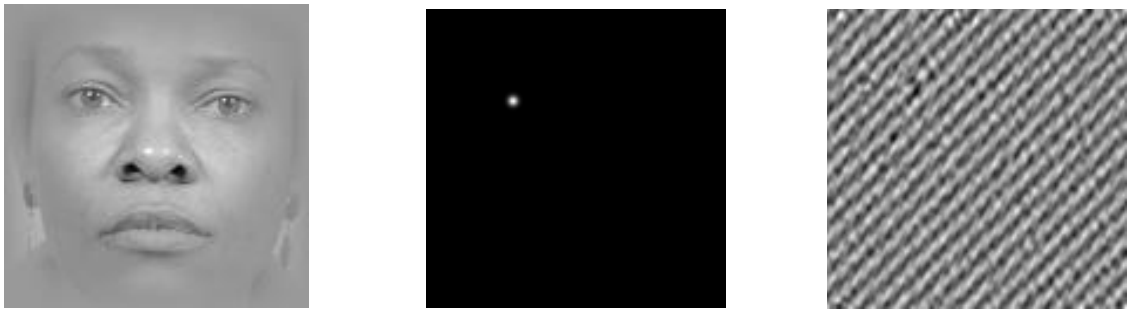


Image I Response G Exact Filter K

Figure 8. Example of an Exact Filter [14]

Then by averaging a large number of exact filters, an ASEF filter is produced that will give a response to any image with the trained object of interest in. As the Fourier transformation is a linear process, the averaging can be performed in either the Fourier or spatial domain shown in (13) and (14) where Hμ or hμ are the final ASEF Filters.

$$H'_{\mu}(w, v) = \frac{1}{N}\sum_{i=1}^{N} K_i(w, v) \tag{13}$$

$$h_u(x,y) = \frac{1}{N}\sum_{i=1}^{N} k_i(x,y) \quad (14)$$

As a result, the filter can be applied to any image that has been converted to the Fourier domain by multiplying the image by the ASEF filter in the Fourier domain to produce a response (12). The method is very simple and quick as it only relies on element-by-element multiplication and Fourier transforms, which are suitable for a mobile application with limited computational power. However, due to the nature of the process, it is restricted by the scale and rotation of the objects that are trained, therefore results will need to be analysed to see how accurate the processes is when objects are seen at different rotations to the training set and when a filter is trained with the same object at different scales and rotations.

5.1.5 Camera Options

The camera for the vision processes needed to fit a select set of criteria, which were:

- It must be small enough to fit on a mobile platform.
- It must provide enough resolution for image processes.
- It must be able to transmit visual data wirelessly without interrupting other wireless transmission sources.
- It must be able to transmit data in real time.
- It must be able to be powered from a low voltage, small power source.

There were two options for the camera; either the camera could be built from scratch into the hardware of the platform, or a wireless, off the shelf option could be used. However, as the main feature of the project is to implement vision processes the building of the camera hardware needed to be time and price justified.

The first option found was a Pinhole Camera, Receiver and USB Image Converter from [15]. The benefits of this camera were that it was small, measuring only 2x2x2cm, would transmit the image in real time directly to a PC-terminal and could be powered from a 9v battery. The only downside to the camera was that the resolution was limited to 380tv lines, which could affect image processes. Other similar micro camera packages exist on the market, however this package appeared to be one of the best on the market in terms of size, price and resolution.

In terms of building the camera system the most suitable camera choice for the project was a SEN-08667 CMOS camera [16] which provided a similar resolution to the off the shelf option. However, this device also needed to be combined with some form of wireless transmission unit, which would also require a PIC to process the data into a transmittable form. Two such options were a Bluetooth SMD module [17] or a Wireless Transceiver [18], combined with a PIC made the hardware close to the price of an off the shelf unit. Therefore as the prices were similar, the time needed to build custom hardware was not justified.

5.2 Ultrasonic Research

5.2.1 Ultrasonic Theory

The primary purpose of the Ultrasonic rangefinders is to provide obstacle avoidance on the robot and to provide a secondary distance measurement method for when an object of interest is detected.

Ultrasonic rangefinders operate by producing a pulse of high pitch sound between to 40-60KHz through a transmitter and listen for the rebounded chirp of noise through a receiver [19]. If no chirp is detected within a suitable time frame, the device assumes that there is no object present, however when a chirp is detected, the distance between the rangefinder and object can be calculated by the time it takes for the pulse to be received, as the speed of sound is constant at 343ms in air.

One significant problem with ultrasonic rangefinders is that they are prone to noise, as they can operate at the 60KHz range, indoor lighting can affect the receivers as they also work at a frequency of 60KHz. Another source of noise that may affect the response of the device is motor noise or any other noise-generating component on a robotic platform. This is because again if the frequency of noise is close to the 40-60KHz ranges then the receiver will detect the noise source as one of its own chirps. As the application in which the rangefinders will be used is ground based, it also provides another cause for concern as the chirps may end up bouncing off the ground.

If an issue such as the floor creates a problem for the receivers or the motors likewise then sound insulating materials can be incorporated into the design of the robotic platform to reduce the effects such as foam. Combined with the price of each individual device, their spanned direct vision range (Figure 5) and the simplicity to implement them, they make a great choice for the application of object detection and avoidance.

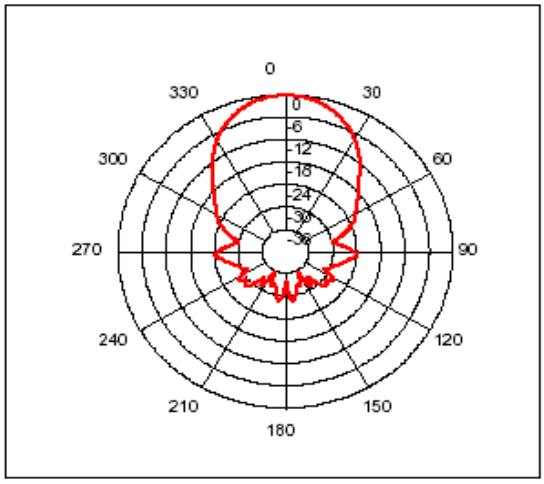


Figure 5. Typical detection range of a standard ultrasonic rangefinder [20]

5.2.2 Time Division Multiplexing

One big problem that can occur due to using multiple ultrasonic rangefinders in close proximity is Time Division Multiplexing (TDM) [21]. Ultrasonics fall victim to TDM when

a separate ultrasonic receiver receives another ultrasonics chirp. This may not be a constant problem, as it may only occur under certain environmental conditions, such as when a wall causes a rebound towards a second ultrasonic rangefinder soon after it chirps, making an object seem closer than what it is (Figure 6).

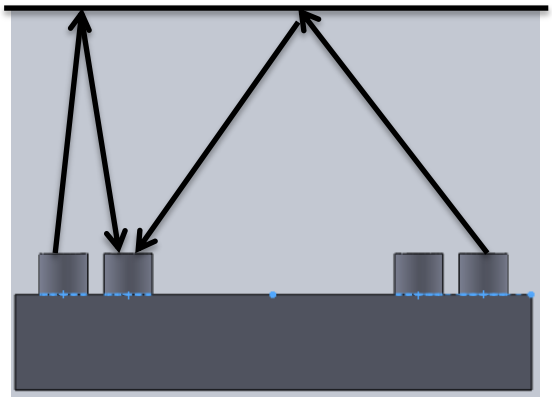


Figure 6. Ultrasonic rangefinder receiving a chirp from another ultrasonic rangefinder

A simple way to prevent TDM in ultrasonics is to trigger the ultrasonic chirps at alternating times with a substantial delay between each chirp, so that if it is detected by an alternative ultrasonic, it is not registered as the listening period of the ultrasonic will be over. This processes is slower but if TDM is not taken into consideration the mobile system will detect false objects, which will prevent smooth movement control.

5.2.3 Ultrasonic Device Comparison

Three ultrasonic devices have been considered to be used onboard the robots. The first is a custom ultrasonic unit that consists of two transmitters and one receiver provided by the University of Reading. This device is suitable for the project, as it has already been programmed to trigger the transmitters at alternative times preventing TDM. However, the accuracy of the devices has not been tested for the current application and the components used are known for being affected by noise.

The second device is a SRF05 Rangefinder [20]. Again it is already preprogrammed, but as each device consists of its own single receiver and transmitter the triggering control needs to be designed on a separate controller for when multiple devices are used to prevent TDM. Due to the price bracket of the device, the accuracy is expected to be much higher as each unit returns a high peak pulse between 100_{us} to 25_{ms} (Figure 7) before timing out, if a drop occurs between this time a controller can read this as a chirp being received, so the overall accuracy of the system using the device will depend on the sampling rate of the main control processor.

The final option, if there is too much sound interference for either of the previous devices, is a Sharp Infrared Rangefinder GP2Y0A02YK0F [22]. Since the device is light based, noise that effects ultrasonic devices will not affect the readings. The range of sight of the device is also longer. However, the device does have a short range blind spot which means it may need to be placed near the back of the platform for it to work adequately.

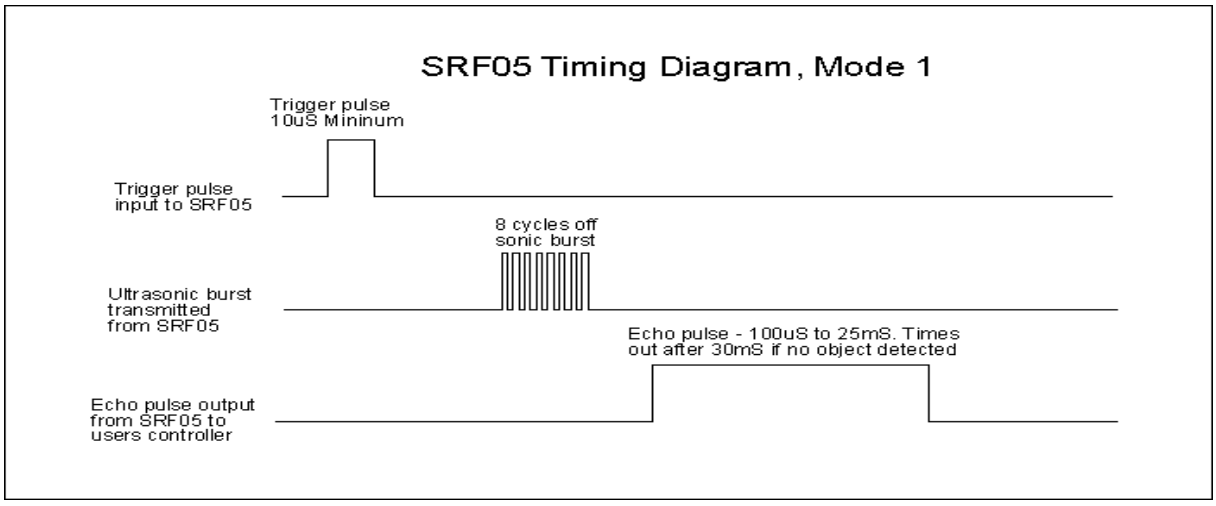


Figure 7. [20] Process of detection on an SRF05 rangefinder and other general ultrasonic rangefinders.

5.3 Existing Omnidirectional Platforms

The design of the omnidirectional platform will affect how adjustable the system is as well as the resolution of the omnidirectional image produced. The design will also affect how resistant the platform will be to vibrations caused by movement, which is vital on a mobile robot to maintain a centralised point in the returned image.

The most popular omnidirectional platform design uses a clear Perspex tube to hold a lens above the camera (Figure 8). This creates a sturdy support for the lens, reducing vibrations on the vision components. It also produces a full 360-degree field of vision, as the complete support structure is see-through. The downfall to this design is that in bright lighting conditions, glare is created and once implemented it is completely unadjustable. Heavy solid bases are also used along with this platform to increase the stability of the design, countering the weight of the reflective lens.

Figure 8. Footbot robot from the Swarmanoid project [23].

The second design is a fixed secure structure with three think steel beams to support the vision producing components (Figure 9). This however creates three thin

blind spots in the omnidirectional image but eliminates glare that would be created if using a Perspex tube. This makes the design more suited to outdoor use in low light conditions. However, the properties of the system need to be known before constructing, as there is no adjustability in the height of each component.

Figure 9. Omnidirectional vision platform designed by Dr Libor Spacek [24]

The final design (Figure 10) is similar to the previous design in the sense that it uses three support rods but it also contains adjustable plates for the camera and lens. This allows adjustments between the components as well as their height from the ground. However, the design does reduce the resistance of the platform to vibrations so relies on movement control to eliminate jerks, which could create oscillations in the lens.

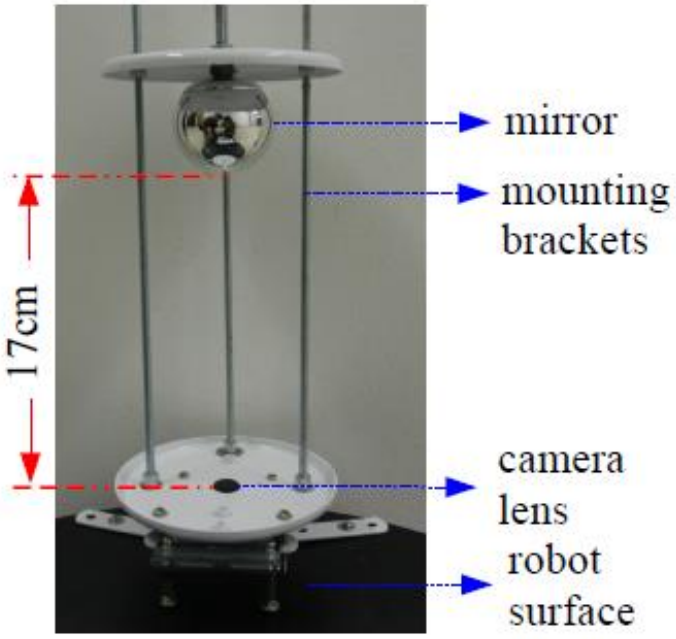


Figure 10. Design used in the process of exploring FPGA-based image processing [25]

These designs show that there are multiple ways of implementing an omnidirectional vision platform, some of which could be considered testing platforms such as the third design due to the adjustable parameters of the design.

5.4 C Sharp Library Research

C# has been selected as the main language for the project as it provides a simple interface designer, along with a standard coding environment all within Visual Studio 2010. It allows the interface to be designed simply and is also a suitable language for when additional libraries are required as not only can new c class libraries be added simply but there is also support for .dll's.

5.4.1 Camera Input Libraries

The first key process that is required within the programming environment is to feed in a live camera stream. Two libraries have been found that can perform this process; EMGU [26] the C# equivalent of OpenCV [27] and a C# Webcam Capture Library created by Philip Pearce [28].

EMGU appeared to be the stronger of the two libraries, however discussion boards online have suggested that there are problems with using different image data types in live feeds. This could present itself as a problem as there are incompatibilities with different image data types and the standard C# image types.

5.4.2 Fast Image Manipulation Libraries

EMGU prides itself as the fastest image processing library for C#, this is because unlike the standard C# image processes EMGU implements pointers automatically to gain pixel information from different segments of images which is considered unsafe in the standard image libraries of the C# programming environment.

5.4.3 Blob Detection Library's

Two mainstream image-processing libraries show promising blob detection algorithms including the use of CCL, which are AFORGE.net [29] and EMGU again. However, there are multiple c++ small libraries that have .dll versions for C# including cvBlobsLib [30]. The AFORGE.net version does come across as the stronger library for this image process as it is recommended by multiple discussion boards and through tutorials in implementation of the algorithm on the AFORGE site.

5.4.4 Fourier Transform Library's

A fast Fourier transform algorithm is the key process within the ASEF filter development and implementation on real time images, therefore a fast and accurate process is required. One way of speeding up the process is to find a library that does not rearrange the frequency response to a more human readable form.

The most popular one for c++ implementations is the Faster Fourier Transform in the West (FFTW) library [31], which comes in a .dll form for C#. What makes this algorithm unique is that it actually modifies itself for efficiency depending on the type of machine it is working upon.

The second most popular library to use Exocortex.DSP [32], developed by Ben Huston. It is now slightly out dated as the last version was released on the 6th October 2006. However, it has been recommended due its fast 2D Fourier transform algorithm.

Problems may occur due to the code being out dated when implementing the library in newer C# programming environments, in which case the code could be used as a base for developing a more up to date Fast Fourier Transform Library.

6. Hardware

A remote control tank has been provided for the purposes of this project, to build the omnidirectional vision platform upon. This remote control tank has been stripped down to the bare minimum consisting of the motors, motor control board and power connectors. All other components have been sourced and built upon the original circuitry.

6.1 Microprocessor

The microprocessor that has been selected is a PIC24H64GP506, primarily because the University of Reading has used this type of PIC with other mobile projects within the School of Systems Engineering, so a simplified daughter board has also been acquired. The daughter board also integrates a MRF24J40MA MI-WI Transceiver simplifying communication between the robot platform and a base station.

The selected PIC also provides enough A/D and I2C connectors for an ultrasonic rangefinder and two tachometers, which are key for the movement control capabilities of the robot.

6.2 Ultrasonics

A single ultrasonic rangefinder with two moveable transmitters has been selected to be used on-board the robot. As discussed in the research section 5.2.3, this type of device when controlled correctly can eliminate TDM and only requires one 3.3v power source and two lines to the processor to transmit its on-board timer values after detecting a chirp. The transmitters are positioned at a 90-degree angle to one another for left and right object detection, relative to the robot platform and are encapsulated in foam to reduce the noise from other on-board components.

6.3 Optical Tachometer

The optical tachometer selected is a custom made device designed again by the University of Reading. It is a close ranged device so needs to be positioned close to the moving tracks of the tank. The tracks must also be painted with black and white stripes at a set distance between each other so that an accurate distance calculation can be performed on-board the processors.

Each device has four pins, two of which require a 3.3v power source, one to power the device and one to set it to an active mode. Another is used as a logic line to the processor and one to ground the device. The parts for the device can be seen in Table 1, with Farnell component reference numbers and the schematic in Figure 11.

Name	Component	Value	Qty	Farnell
PL27,28,29	4WP		3	1593413
U26,27,28	AD8541		3	1333254
Q2,3,4	BC847B		3	1081232
C57,58,59	C0805	100nF	3	1759266
R43,44,45	R0805	1K	3	1469847
R49,53,57	R0805	10K	3	1612522
R58,59,60	R0805	100K	3	1469860
R48,52,56	R0805	10M	3	1469858
R46,50,54	R0805	4K7	3	1469923
R47,51,55	R0805	390	3	1576453
U20,21,22	TCND5000		3	1470584

Table 1. Component list for the Optical Tachometer.

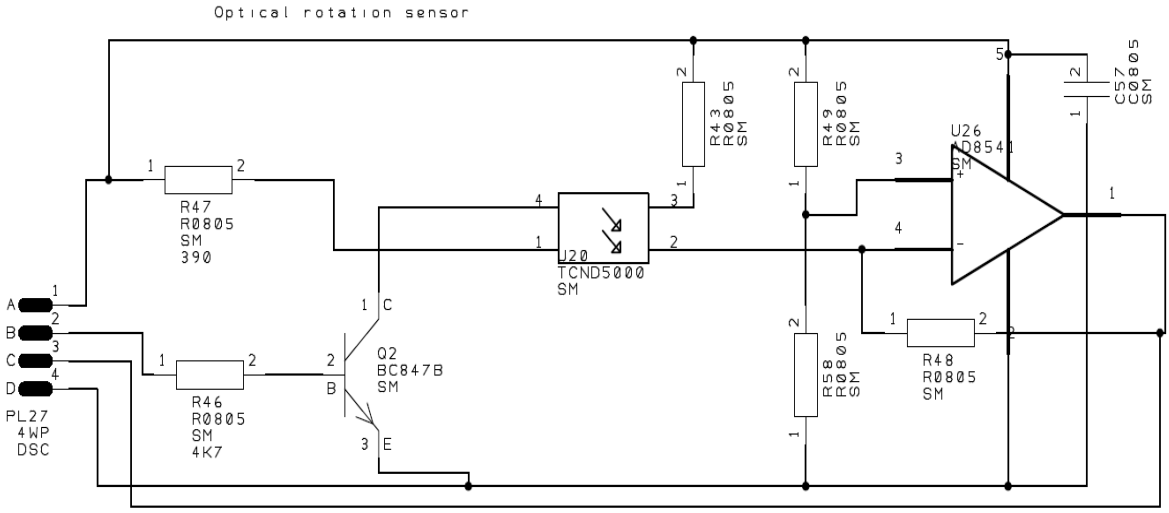


Figure 11. Hardware schematic for the Optical Tachometer.

6.4 Wireless Camera

An off the self-camera has been selected for the purpose of the project, because as discussed in the research section 5.1.5, the main aim is to explore vision processes, so the extra time required to build a custom camera unit cannot be justified when existing hardware performs the same job at a similar price. The camera that was selected came as part of a set that included a small 2x2x2cm wireless pinhole camera, wireless receiver and a USB image converter so the full process of transmitting a live stream to a base station in real time was handled.

There was one issue that needed to be explored with the camera and that is that the camera selected uses a wide-angle lens to increase its field of vision.

6.5 Power Sources

There are two power sources onboard the robot, the first being a NIMH Overland 1600mAh batteries to provide the power to the processor, motors and all other components. The second is a standard 9v battery that can be connected to the wireless pinhole camera through a 9v PP3 to 2.1mm connector, making the robot completely mobile.

The wireless camera receiver was powered from a mains connection close to the base station.

6.6 Connection Schematic

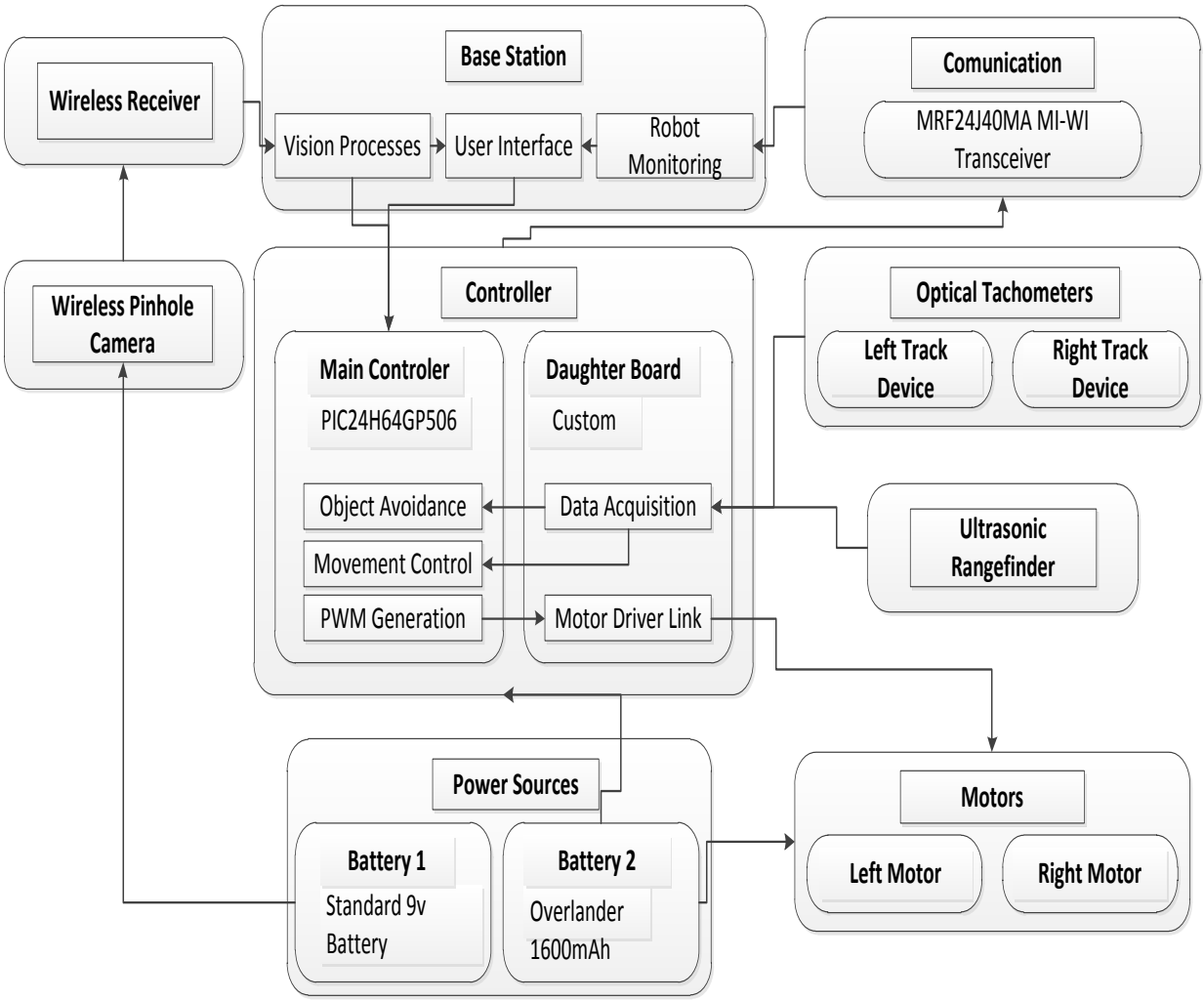


Figure 12. Full hardware schematic that also incorporates the base station software.

6.7 Base Station Hardware Specifications

The key parts of hardware within the base station terminal can be seen in Table 2. The hardware is directly responsible for the speed of image processes within the system. The base station also had a USB Mi-Wi dongle attached for the purposes of communication.

Device	Description
2x Intel Xeon X5560 @2.8GHz	Dual Processors
12Gb DDR3 SDRAM @ 1333Mhz	Memory Modules

Table 2. Key hardware found inside the base station.

6.8 Hardware Control Software

6.8.1 Ultrasonic Implementation

Through the provided daughterboard and base PIC code from the mobile tanks, implementation of the ultrasonics was fairly simple, as a I2C port had been specifically assigned for the ultrasonics, as well as the port set up for receiving the data. The only thing that had to be implemented within the PIC code was to declare the variables to store the history of the data received for averaging purposes, this can be found in Appendix 1, as well as scaling the signal returned to prevent buffer overflows when analysing the response. The code was also designed to transmit the data back to the base station upon request or when significant changes in the values had been made, which can be seen below for when a packet with a value of 16 is sent to the robot from the base station:

```
case 16:
    if (dbytes==0)     //send ultrasonics
    {
        I2S();I2send(0xB8);I2send(0);I2SR();I2send(0xb9);
        BroadcastPacketAR[0]=16; //send message ID
        for (i=1;i<=8;i++)
            //Transmit all bytes of ultrasonic data
            {BroadcastPacketAR[i]=I2GET(i!=8);}
            I2P();Broadcast=9;  // number in packet
            }
break;
```

6.8.2 Tachometer Implementation

The two tachometers to be used on each track also had pre built I2C ports on the daughter board with the ports set up similarly to the ultrasonics. The code that was implemented to handle the data returned by the tachometers can be found in Appendix 2. Using the variables of the tachometers, movement control can be implemented and monitored. The base station can also view the values of the tachometers by sending a packet with the value of 17 from the base station to the robot.

```
case 17: // Opto information
    BroadcastPacketAR[0]=35;
    BroadcastPacketAR[1] = pos2;
    BroadcastPacketAR[2] = pos2<<8;
    BroadcastPacketAR[3]= pos1;
    BroadcastPacketAR[4] = pos1<<8;
    Broadcast = 5;
break;
```

6.8.3 Object Avoidance

For movement control a simple object avoidance algorithm was implemented onboard the robot to prevent collisions with walls. The algorithm took into account the averaged values of the ultrasonic range finders, to eliminate noise and maintain a slow speed to

make the platform as stable as possible. The algorithm was based on a state machine that checked the values of the left and right ultrasonics and slowed down the track relative to where an object was detected. The full speed control function controlled by the ultrasonic algorithm can be found in Appendix 3.

7. Platform Design and Construction

The key part of the whole omnidirectional platform is the dimensions of the conical lens, as this will determine the size and position of all other components on board the platform. However, as discussed within the research section 5.1.1, the dimensions of the lens needed to be calculated to determine the visible distance range of the system.

7.1 Lens Size Calculations

The size of the lens determines key parameters that affect the field of view of the system. Therefore the path of light that reaches the camera lens needs to be analysed from the different angles that it can reach it through the reflection in the lens. A simplified model can be seen in Figure 13.

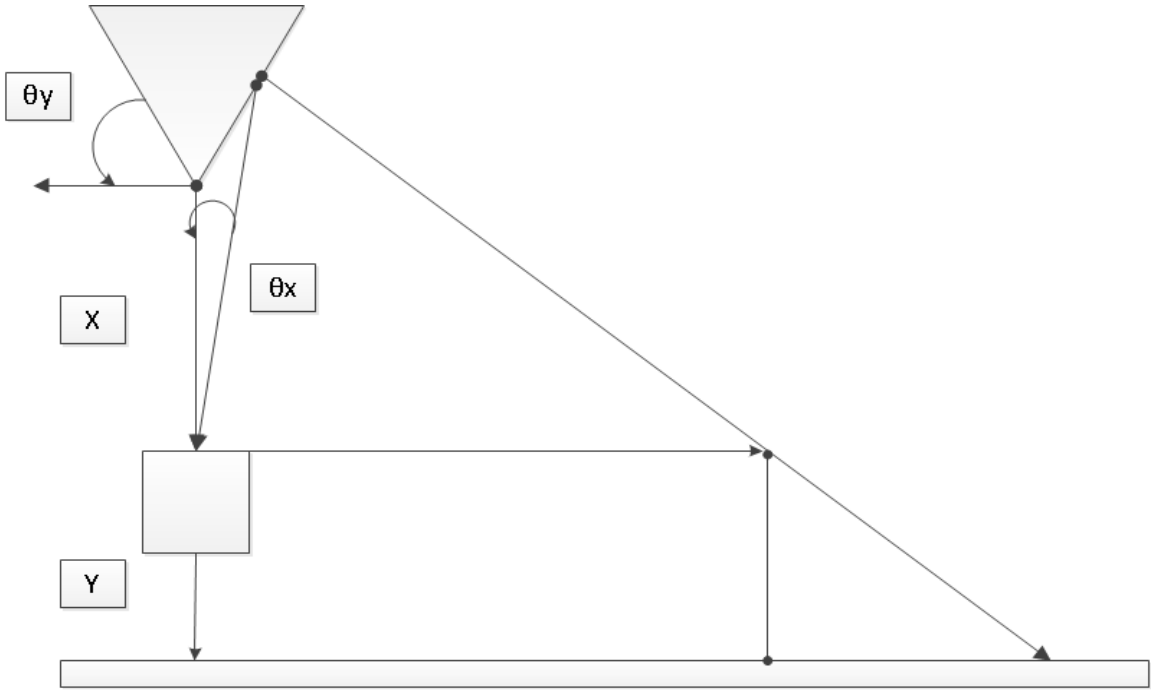
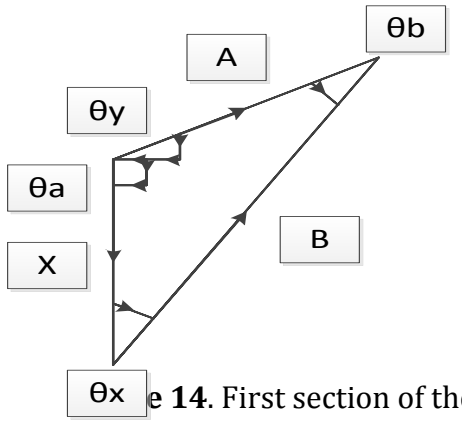


Figure 13. Demonstration of how light reflects into the camera lens.

From the diagram, it can be seen that there are four key parameters that can be adjusted; θx is the angle into the camera lens, θx is the angle of the lens in reference to the horizon, Y the height between the ground and the top of the camera and X the height between the camera lens and the point of the lens. The path of light can be broken down into four individual shapes in which trigonometry can be used to determine the distance seen at any given point in the lens. The first shape shown in Figure 14 handles the path of light from the lens to the camera.



$$\theta a = \theta y + 90$$

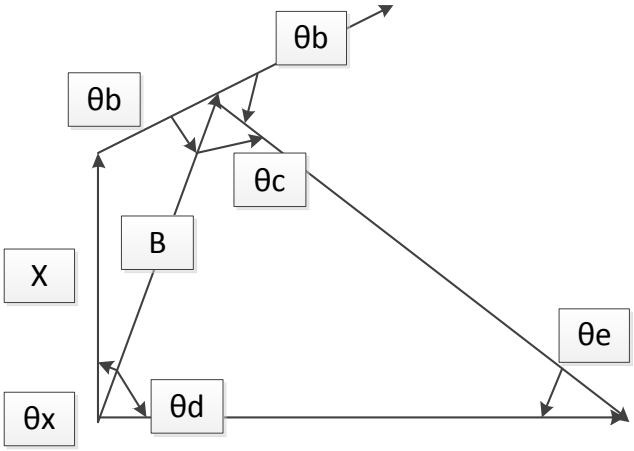
$$\theta b = 180 - \theta a - \theta x$$

$$A = \frac{X sin(\theta x)}{\sin(\theta b)}$$

$$B = \sqrt{A^2 + X^2}$$

Figure 14. First section of the distance breakdown model.

The second triangular subsection uses the fact that as the lens is reflective, as the line of light will rebound off the lens at the same angle as its entry onto the lens. Other calculations also link back to parameters found in the first triangular subsection as shown in Figure 15.



$$\theta c = 180 - 2\theta$$

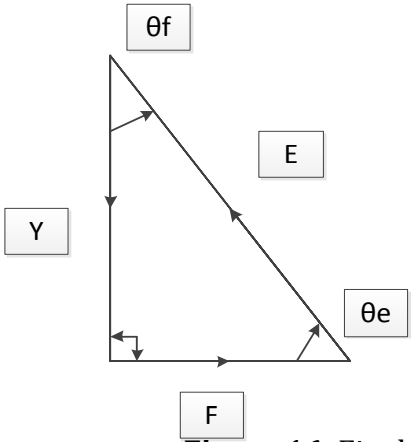
$$\theta d = 90 - \theta x$$

$$\theta e = 180 - \theta c$$

$$\mathrm{D} = \frac{B sin(\theta c)}{\sin(\theta e)}$$

Figure 15. Second section calculations in the distance calculation breakdown.

The third subsection is a rectangular piece; however there are no parameters that is related to the distance range except from the length across the ground that is equal to D previously calculated. Therefore, the final object breakdown piece is a right angled triangle and since it is a continuation of the same line from the second subsection, one of the angles is already know from θe making it solvable as shown in Figure 16.



$$\theta f = 180 - 90 - \theta e$$

$$F = \frac{Y sin(\theta f)}{\sin(\theta e)}$$

Figure 16. Final section calculation analysis.

Through this breakdown of the field of view, it can be seen that the total distance (R) that can be seen given initial values of X,Y, θx and θy is equal to F+D. However this is through a long processes of taking sub sectional shapes, by using the relations found through the shape breakdown F and D can be calculated in terms of the initial parameters as shown in (15) and (16).

$$F = \frac{Y sin(90-(180-\big(180-2*(180-(\theta y+90)-\theta x)\big)-(90-\theta x))}{Sin(180-\big(180-2(180-(\theta y+90)-\theta x)\big)-(90-\theta x))} \quad (15)$$

$$D = \frac{\sqrt{X^2+(\frac{X sin(\theta x)}{\sin(180-(\theta y+90)-\theta x)})^2}*\sin(180-2*(180-(\theta y+90)-\theta x))}{\sin(180-\big(180-2*(180-(\theta y+90)-\theta x)\big)-(90-\theta x))} \quad (16)$$

Using the two formulas and subbing in suitable values for θx and θy, a suitable range of distances can be calculated for different θy angles, which will determine the dimensions of the lens. For testing purposes a set lens radius of 35mm has been selected to fit the dimensions of the tank, so the angle of θy will only affect the height of the lens. The results of distances compared to θy can be seen in Appendix 4.

Through this table of data it has was decided that a lens with θy of 16 degrees would be used, this is because a suitable range of distance would be visible in the lens within a slightly smaller region. However, this would not affect the resolution of the image too much. The reason for this selection was that the properties of the camera were unknown at this point in the project so X and Y would need to be left adjustable to learn how the camera would best suit the needs of the system

7.2 Lens Creation

To provide the system with enough field of view as calculated in 7.1 the parameters selected to be used were: θy of 16 degrees, radius of 35mm and height of 10mm. To make the lens as accurately as possible, the lens was drilled out on a Lathe from an aluminium beam, which produced an accurate angle for θy. A thick, strong screw with a circular head was then drilled into the top of the lens to provide a handle to prevent touching the reflective section of the lens. It was then placed back in the Lathe to be sanded down with Brasso polish in a circular motion to remove all circular lines creating in in the first process. The Finished lens can be seen in Figure 17.

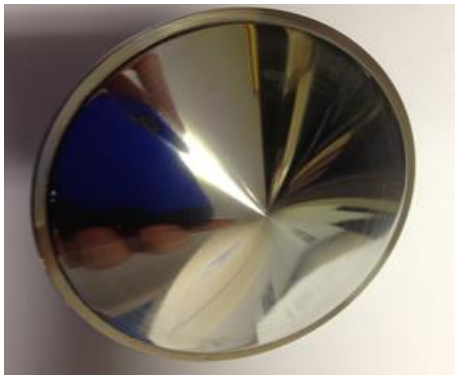


Figure 17. Reflective conical lens.

7.3 Design Ideas

Two designs were initially developed, the first used a clear Perspex tube to hold the lens above the camera, which allowed a full 360-degree range of vision and was relatively sturdy so the lens would not wobble in relation to the cameras position. However, in light intense areas the Perspex tube would produce significant glare, which could affect the vision processes of the system and once created, would be completely unadjustable.

The second design consisted of three thin but sturdy steel rods that support the lens above the camera. This design would produce three small blind spots on the lens, however it would not create glare like the Perspex tube design. The design would also allow the heights of the camera and lens to be completely adjustable which was suitable since the properties of the camera were unknown, allowing the range of distances that the systems could see to be explored. Therefore this design was used for the prototype platform.

7.4 Design of the Platform Base

The first part to the design was the baseboard that the omnidirectional vision components would rest upon. It includes; three sturdy 3mm blocked holes for the support rods to slide into, a cut out section so that a PIC processor can fit on the lower tank section and multiple drilled out holes for the purpose of screwing the board to the tank and so that components could be added the platform if needed for further additions to the project (Figure 18)

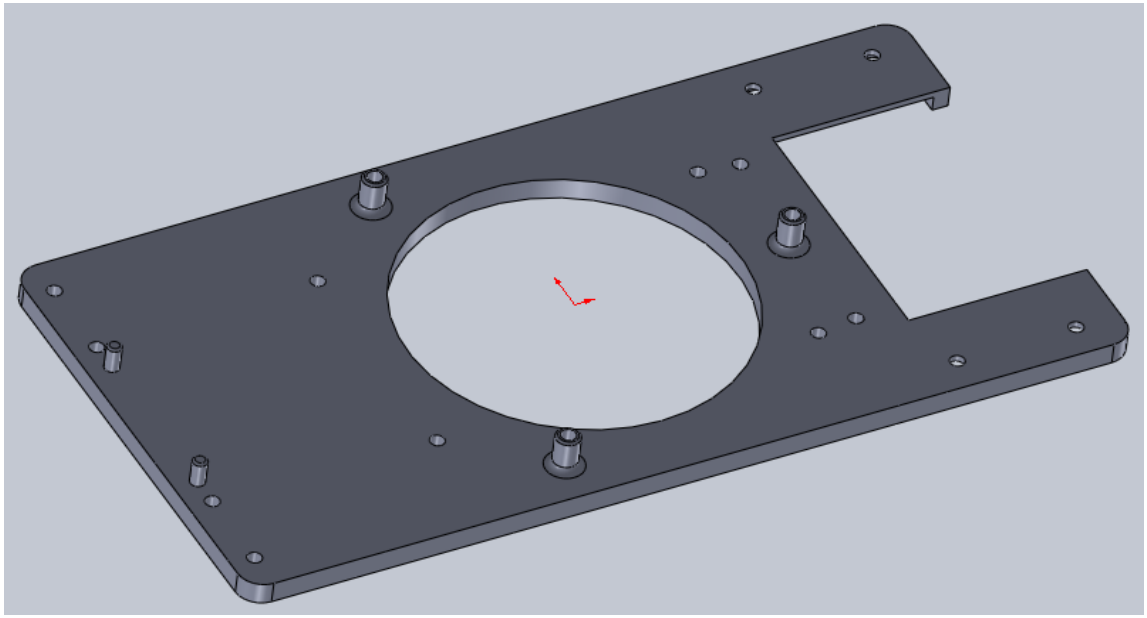


Figure 18. Baseboard design.

To secure the baseboard onto the tank, spacers were also designed to fit onto the underside of the board to raise the platform to a set height so that hardware components such as the motor control board and processor would fit under the vision platform.

7.5 Design of the Lens Holder

The lens holder (Figure 19) allows the lens to simply sit reflective side down. The lens can then be secured at any height with washers and nuts when used with 3mm steel threaded rods. Additional material has been added around the rod holes to make the part sturdier.

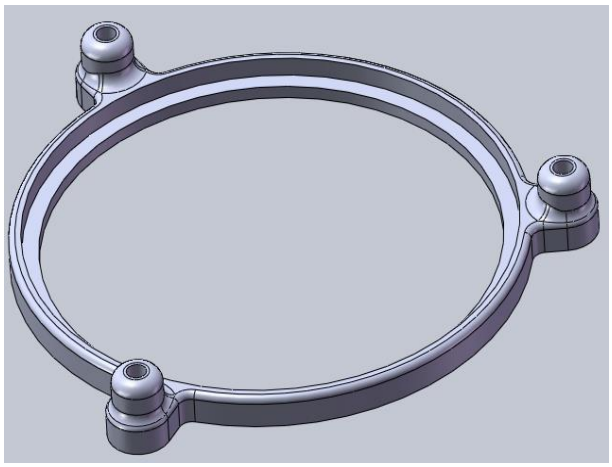


Figure 19. Lens holder design.

7.6 Design of the Camera Holder

The camera holder (Figure 20) has been designed in a similar way to the lens holder with additional material around the rod holes. A raised section in the centre of the part has been included so the camera can be slid into a central position so the central point of the camera lines up with central point of the lens. The part can also be placed at any height with the use of washers and bolts on a threaded 3mm steel rods.

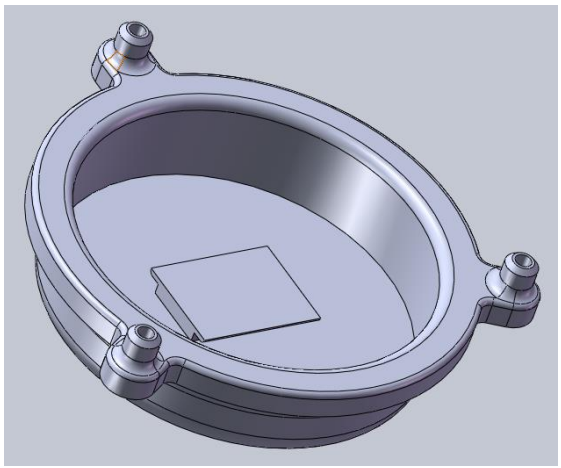


Figure 20. Camera holder design.

7.7 Additional Components

There is only one additional part that is not a vital part of the omnidirectional platform functionality and that is a simple rectangular board (Figure 21) that can be placed on the tank that allows the processor to be placed upon it easily using Velcro. This allows easy removal of the board for reprogramming. It also prevents the processor being damaged

from environmental objects as none of its edges go over the edge of the rectangular board.

Figure 21. PIC resting plate.

7.8 Final Design

The completed design containing all components linked together using Solidworks can be seen in Figure 22.

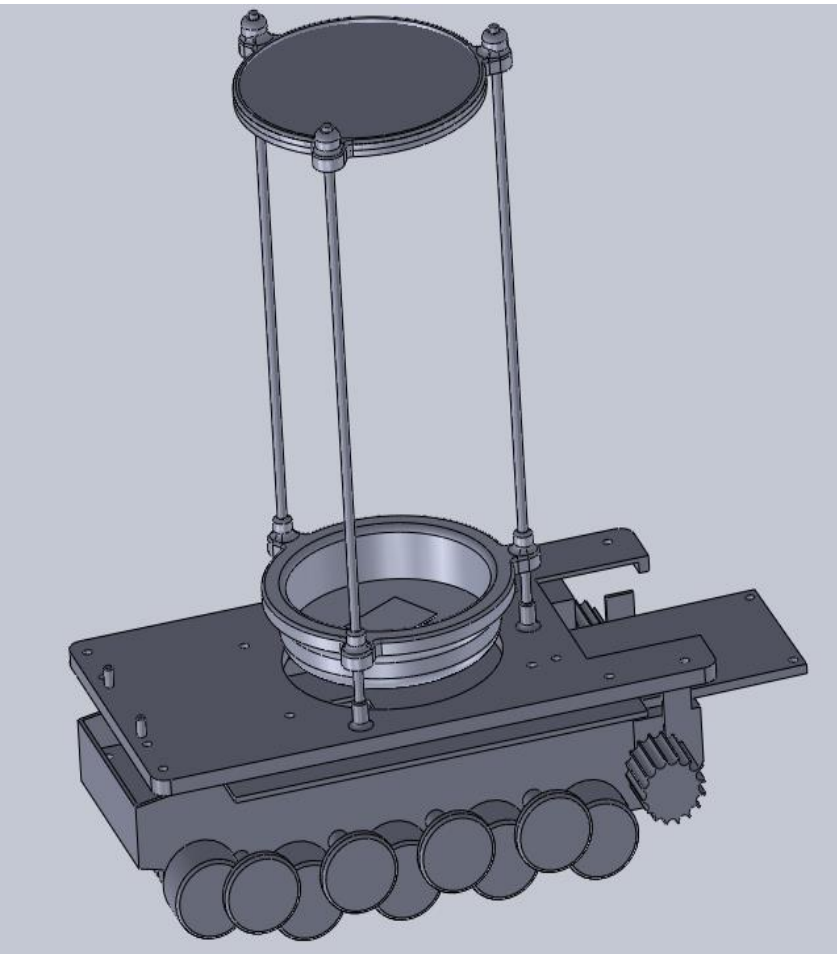


Figure 22. Final design of the omnidirectional mobile robot.

7.9 Construction

The construction of everything apart from the steel rods was made using a 3D printer to create an accurate and professional looking prototype platform (Figure 23). All hardware has been attached in the image.

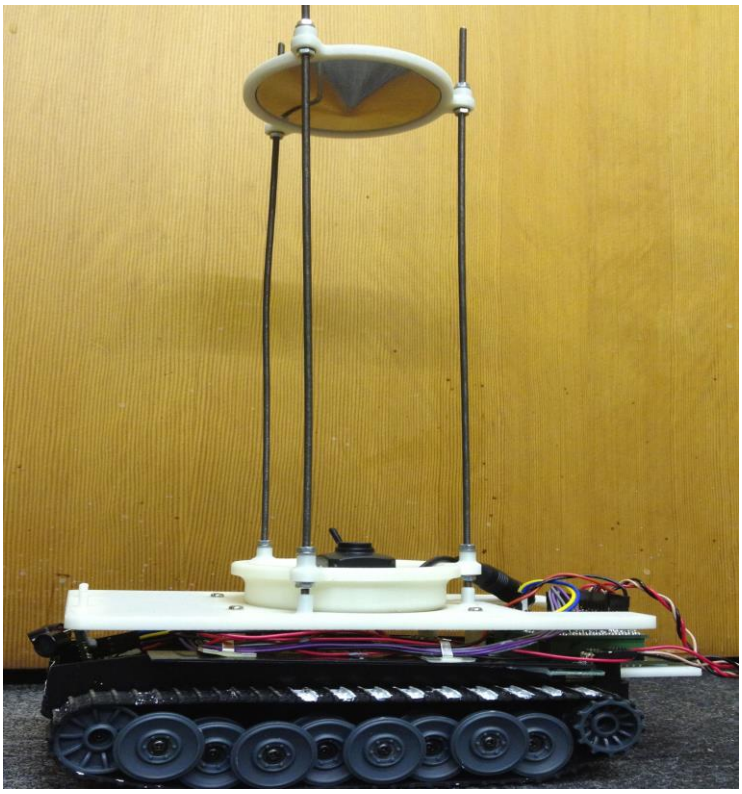


Figure 23. Final constructed mobile omnidirectional vision platform.

7.10 Sight Calibration

Following the completion of the platform, the camera and lens were set up so the camera was focusing on the central point of the lens. The lens was then raised to a height of Xmm above the camera so that the entire lens was within the image sent to the base station, which was analysed with the software that was provided with the camera. The height was significantly higher than was expected from the calculations, however, it was necessary due to the wide-angle properties of the chosen camera, something that could not of been accounted for until testing on the platform.

The minimum and maximum field of view of the lens at this height was then measured at the front, back and sides of the robot and linearised to form the diagram shown in Figure 24. This shows that the vision range of the system has a range between 120mm to 310mm but is not constant around the whole robot. However, the values could be used in conjugation with an image to determine the angle and distance an object was from the central point of the robot. This is because in an unwrapped image the height is relative to the distance and the width position is relative to the angle.

Figure 24. Linearised field of view.

8. Image Processes

The image processes section address all the functions required to provide an image for detection and identification.

8.1 Implementing a Live Feed

Attempts were made to use the EMGU library to implement the live stream from the Pinhole camera in C#. However, it was not compatible with the data type of the images being provided by the USB image converter, therefore the webcam Capture Library created by Philip Pearce was implemented.

The library was simple to implement, as it only required the WebCam_Capture.cs file provided and the declaration;

```
using WebCam_Capture;
```

The height and the width of the displayed image was handled entirely by the library so all that needed to be done within the main application is to either set a picture box to an instant of the webcam image or save the image to a bitmap variable;

```
// set image sizes at start of the program to return camera feed of certain size.
this.WebCamCapture.CaptureHeight = this.pictureBox1.Height;
this.WebCamCapture.CaptureWidth = this.pictureBox1.Width;

// Set picture box to an instance of the live feed
this.pictureBox1.Image = e.WebCamImage;

// Store the instances of the live feed to a bitmap variable
public static  Bitmap Image_Input;
Image_Input = new Bitmap(e.WebCamImage);
```

This provided a real time live feed application in C#.

8.2 Unwrapping the Image

Two versions of the unwrapping algorithm were produced; the first one used the standard drawing libraries of C# while the second used the image libraries of EMGU, primarily to see if there was a significant difference in the processing speeds of the two libraries.

Both processes require the same sized images, which are dependent on the diameter and circumference of the lens, which are provided by the user. The unwrapping process is then dealt with as a function. The image size calculations are;

```
// Load image from main program
Bitmap Original = new Bitmap(BaseStationGUI.Image_Input);

//Calculate the circumference of the lens given the diameter
int circumference = (int)Math.Round(2 * Math.PI * (Diameter / 2) + 1);
// Create the Unwrapped image with the calculated unwrapped dimensions
Bitmap Unwrap = new Bitmap(circumference + 2, (int)(Diameter / 2) + 2);
```

The process of unwrapping involves two loops to calculate half circle segments in which pixel translation takes place setting the move co-oridnate positions from the original image to the unwrapped image. These two loops are contained within a further loop that moves in steps from the edge of the radius to the central point of the lens.

```
// Move down the radius of the circle
for (double r = (Diameter / 2); r > 0; r -= 0.5)
{
     // First half of the circle
     for (double theta = Math.PI; theta > -Math.PI / 2; theta -= (2 * Math.PI /
     BaseStationGUI.thetacalc))
     {
          // Calculate Pixel cords in original image and unwrapped image
          rpix = (int)Math.Round(r * Math.Cos(theta) + centrecol);
          cpix = (int)Math.Round(r * Math.Sin(theta) + centrerow);
          newcpix = (int)Math.Round((circumference / 2) + (2 * Math.PI *
          (Diameter   / 2)) * ((Math.Abs(theta - (Math.PI / 2))) / (2 *
          Math.PI)));
          newrpix = (int)(Math.Round(r));

          // Image Translation Method
     }

     // For second half of the circle
     for (double theta = Math.PI / 2; theta <= Math.PI * 3; theta += (2 * Math.PI
     / BaseStationGUI.thetacalc))
     {
          // Calculate Pixel cords in original image and unwrapped image
          rpix = (int)Math.Round(r * Math.Cos(theta) + centrecol);
          cpix = (int)Math.Round(r * Math.Sin(theta) + centrerow);
          newcpix = (int)Math.Round((circumference / 2) - (2 * Math.PI * (Diameter
          / 2)) * ((Math.Abs(theta - Math.PI / 2)) / (2 * Math.PI)));
          newrpix = (int)Math.Round(r);

          // Image Translation Method
     }
}
```

The key parts that the two different libraries will perform differently are the image translation sections as commented in the above code.

8.2.1 Using the C Sharp Standard Libraries

The standard libraries require the use of the Color data type to store pixel values as well as the two functions GetPixel (x,y) and SetPixel (x,y) which can potentially be slow functions due to the nature of their process.

```
// Pixel Format type set to black for null reference
Color C = Color.Black;

// Get pixel value from image
C = (Original.GetPixel(cpix, rpix));

// Set pixel value in image
Unwrap.SetPixel(newcpix, newrpix, C);
```

Since the code is already in the C# standard libraries format, there is no conversion between image types and can be passed back to the main image by simply setting a public variable from the main program to the image. The full code can be found in Appendix 5.

8.2.2 Using the EMGU Library

The EMGU library has its own image formats and so the Bitmap images from the main program need to be converted to this form, which is the slowest part within the EMGU code process.

```
// Convert C# standard libray Bitmap image into EMGU Image
Image<Bgr, Byte> Original = new Image<Bgr, Byte>(OriginalBit);
```

The EMGU library uses the Color data type in the same way as the C# standard library to store the pixel value, however the .FromArgb() function needs to be used as the EMGU library stores its RGB values in a 3 channel array for each image.

```
//Create color from RGB value array
Color_set = Color.FromArgb(Original.Data[rpix - 1, cpix - 1, 0],
Original.Data[rpix - 1, cpix - 1, 1], Original.Data[rpix - 1, cpix - 1, 2]);
```

This also means that the pixel RGB values also needed to be set in the unwrapped image individually;

```
// Set pixel value in unwrapped image
Unwrap.Data[newrpix, newcpix, 2] = Color_set.R;
Unwrap.Data[newrpix, newcpix, 1] = Color_set.G;
Unwrap.Data[newrpix, newcpix, 0] = Color_set.B;
```

The full code can be found in Appendix 6.

8.2.3 Comparing the results

Both algorithms produced the same image at the same scale of resolution (Figure 25). What can be noticed from the unwrapped image is that there is some blurring of the image towards the top of the image, which is where the point of the lens would be. This is because the image at the tip of the original image has been stretched out; the tip of the lens also has a small surface area that adds to this problem, but the segment of image missing is only of the camera, which does not need to be seen.

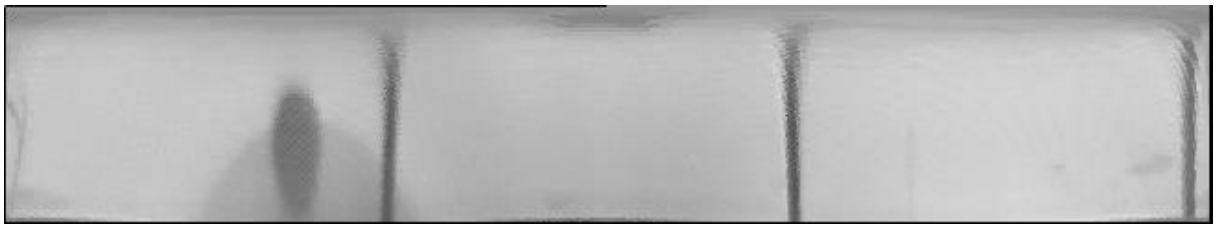


Figure 25. Result of the unwrapping algorithm

Comparing the operational times of the two versions of the unwrapping code, the standard libraries could unwrap each image at a rate of one image on average every 6.71 seconds which is far too slow for a real time application. The EMGU algorithm however could unwrap one image on average every 0.68 seconds which is significantly faster but is still not fast enough for a real time application. The only way around this speed difference is to move the robot at a slow speed so that when a later algorithm detects an object, the object is still present within the range of the robot for an action to be performed upon it.

8.3 Implementing a Reverse Threshold

The reverse threshold function is to be used when the environment has been specifically set to have a bright background with dark objects. The algorithm works by checking the intensity of each pixel in a greyscale version of the unwrapped image. If the intensity is greater than 128, the pixels value is set to 0, however if the pixel is less than or equal to 128, the pixel is set to 255.

The algorithm has been developed using the EMGU library due to the speed difference observed in the unwrapping algorithms and can be seen in Appendix 7. The results can be seen in Figure 26.

Figure 26. Result of the reverse threshold algorithm within a controlled environment

8.4 Blob Detection using AFORGE.net

The Aforge library allows a simple implementation of a CCL algorithm to detect blobs of pixels with size restrictions. This is all done through its BlobCounter class. The first phase to the process is to set the min and maximum restraints on any detected blobs.

```
// Set Blob size restraints
blobCounter.FilterBlobs = true;
blobCounter.MinHeight = 10;
blobCounter.MinWidth = 20;
blobCounter.MaxWidth = 130;
blobCounter.MaxHeight = 130;
```

The second phase is to process the image and to acquire all the information of every object.

```
// Process image
blobCounter.ProcessImage(ThersholdFlip.ToBitmap());

// Get info
Blob[] blobs = blobCounter.GetObjectsInformation();
```

Finally a list of all edge points can be made, which can either be drawn onto a resulting image, or converted into corners that can then be drawn on to a resulting image.

```
//Create a list of all key points and determin the corners of the object
List<IntPoint> edgePoints = blobCounter.GetBlobsEdgePoints(blobs[i]);
List<IntPoint> corners = PointsCloud.FindQuadrilateralCorners(edgePoints);
```

Through this library and the restrictions on the size of the blobs, the support rods of the platform can be eliminated by making sure that the width of the blobs identified are bigger than the width of the rods. The results of the reverse threshold algorithm can be seen in Figure 27 and the completed code in Appendix 8.

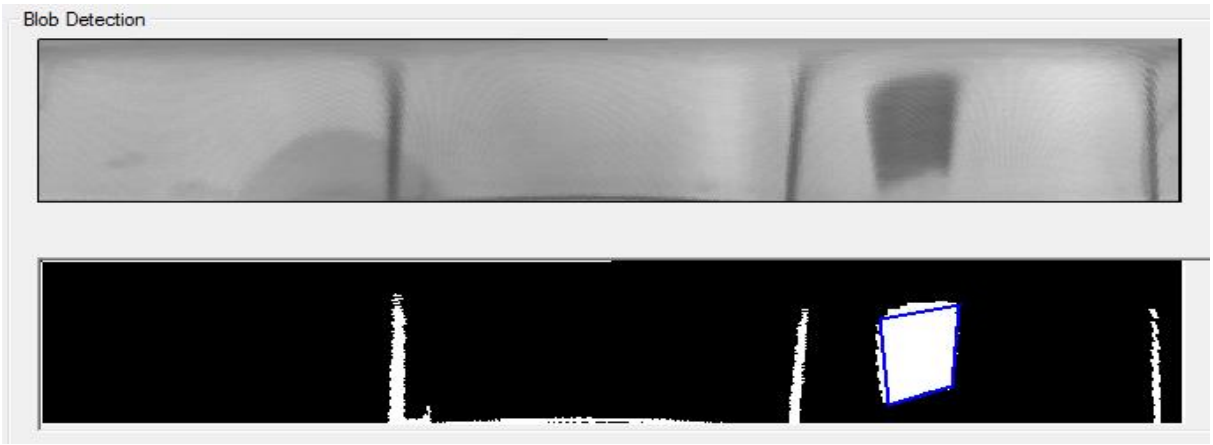


Figure 27. Results of the Aforge.net CCL Blob detection.

8.5 Real-time Stream Implementation

All previous functions discussed in this section work on single images, therefore to implement the full blob detection process in a real time application, the real-time video stream needed to be broken down into individual frames and passed to each function

individually. However, this means that only one image can be handled at any one time until it has been passed through all functions, which either means frames will be lost while waiting for the process to finish on each image, or all frames are passed but there will be a significant delay between each image processing phase reducing the frame rate.

To overcome this issue while maintaining a constant frame rate, the functions discussed have been designed within the constraints of a class and a threadpool, to run the image processes simultaneously so a smooth real-time feed is returned with a minor delay caused by the processing time.

```
// allow Threading to be implemented in the program
using System.Threading;

// create a list of classes so each new image can be initialised as class
List <Image_Process> IU = new List<Image_Process>();

// Create the thread pool with a maximum number fo simulatonios threads
ThreadPool.SetMaxThreads(17,17);

// create a class for when a new frame is acquired and then add the initialisation
function of the class to start the image processes on the frame.
Image_Process tmp = new Image_Process();
IU.Add(tmp);
ThreadPool.QueueUserWorkItem(new WaitCallback(IU.Last().Class_Process));
```

The design of the operation of the application can be seen in Figure 28.

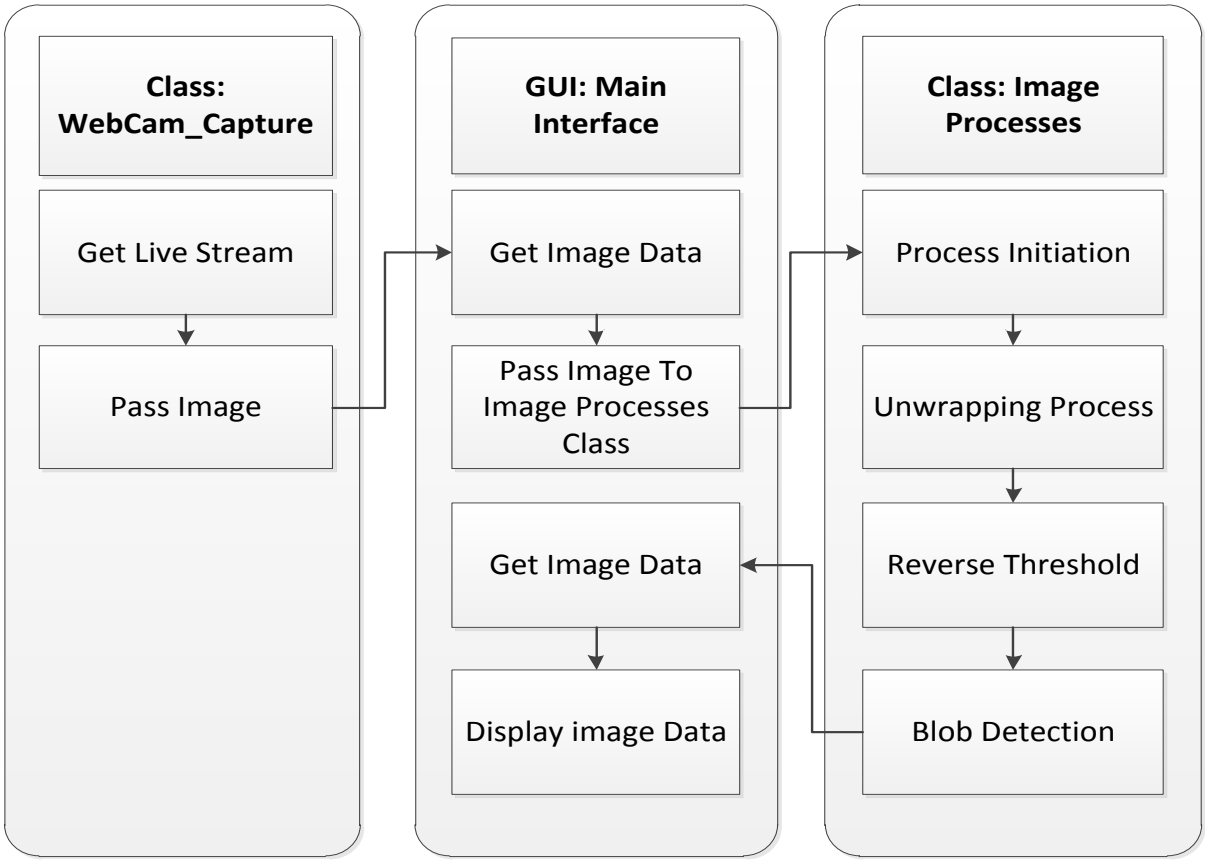


Figure 28. Application Processing Path

The threadpool created a problem in the live stream acquiring process, as multiple threads attempted to access the live stream image simultaneously. C# cannot allow simultaneous access to a single variable so a mutex function was implemented within the data gathering section of the class which prevents other functions and operations from accessing any variables included within it by creating an access queue.

```
// Start access waiting queue
BaseStationGUI.mut.WaitOne();

// Get the data from the main application
diameter = BaseStationGUI.Diameter;
centercol = BaseStationGUI.centrecol;
centerrow = BaseStationGUI.centrerow;
OriginalBit = new Bitmap(BaseStationGUI.Image_Input);

//Release variables for other operations to use
BaseStationGUI.mut.ReleaseMutex();
```

The same process was used within the main GUI code for when the class returned images. The final issue that occurred with the implementation of the live feed was dealing with memory. Since the classes were part of a list, the deconstruction of any individual object proved difficult as the C# implemented garbage collector cannot deal with lists that are currently in use. To overcome this issue a public Boolean variable was created within the image processes class that was set to true once the image processes were completed. A timer was then used within the main application to read the Boolean variables of all objects within the list of image process class members, if it was true then the instance of the class was set to null, freeing any used memory. The final GUI interface can be seen in Figure 29.

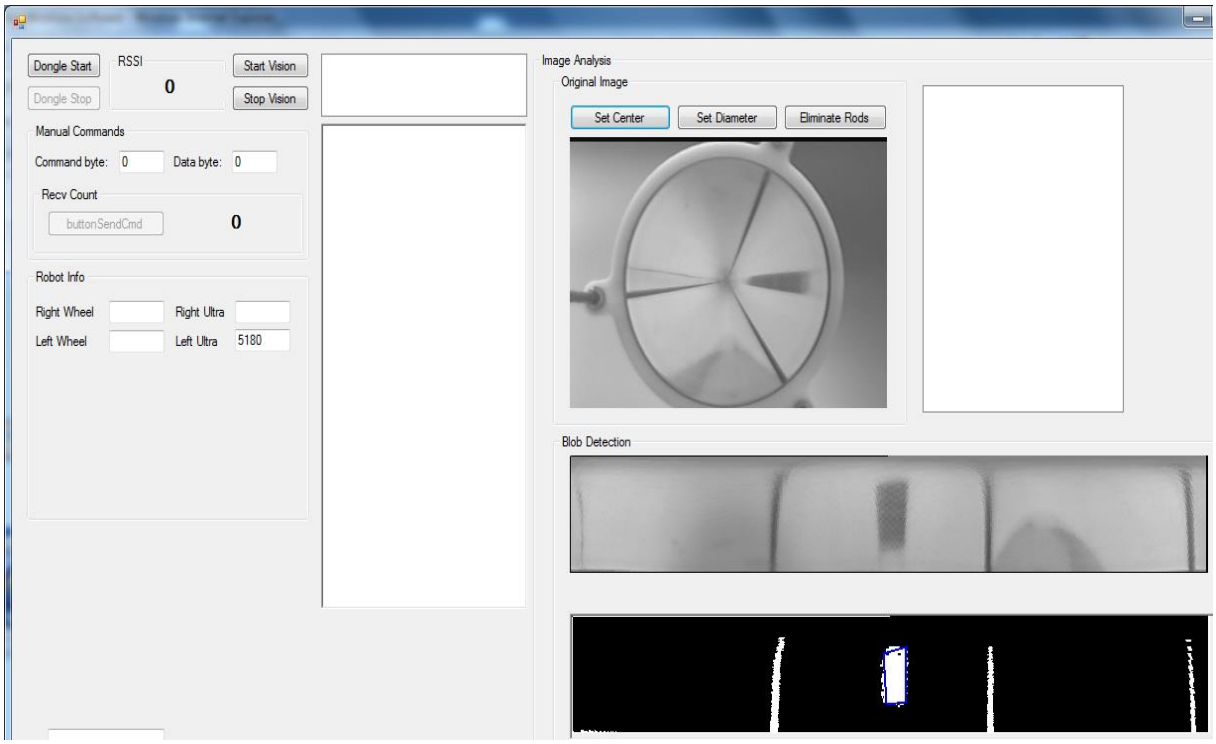


Figure 29. GUI Interface

9. ASEF Development

9.1 ASEF Design and Implementation

There are three key sections to the ASEF filter process; creating image information files, training the filter and applying the filter, all of which must provide images and data in a form that each other section will be able to use.

9.1.1 Creating Image Information Files

The first section of the ASEF processes needed to be able to take an image and allow a user to click on a point of interest. The point of interest could then be saved to a text file to be used by other applications within the ASEF processes. The peseudocode can be seen below.

1. Load images locations into a string list **locations**
2. For each image address in **locations**
   2.1. Load image
   2.2. Display Image
   2.3. Wait for user to click on point of interest
   2.4. Store **location** address to a string Info
   2.5. Append X point click Co-ordinate to string **Info**
   2.6. Append y point click Co-ordinate to string **Info**
   2.7. Push string Info into a list of strings **Data**
3. For each string in **Data**
   3.1. Write line of text to a text file
4. Close Application

This application would only rely on one folder of images to run and would output a text file in the form of:

Image Address" "X Co-ordinate" "Y Co-ordinate

This way any other application can easily determine each bit of data by the spaces between each bit of information. However, it does mean that there is a long user processes required for determining all points of interest for each individual image in a data set.

9.1.2 Training Process

The training process was designed to run two key functions; the first to store each image and its information gained from a data file created by the first application in a structure and to train an ASEF filter from the data which would save an ASEF filter as a text file. However, as some parts of the algorithm would be required in the testing application, the key processes were designed within a class. The class design can be found in Appendix 9. The pseudocode for the training process is as follows:

Main Application:
1. Create list of structures from class structure template
2. Open text file
3. Read each line of text file
   3.1. Create new structure element in list
   3.2. Store first word in line to location variable in structure
   3.3. Store the second and third word to the point variable in structure
4. Create new class for training purposes
5. Create thread for training and pass class structure with training data to function **TrainImages**();
6. On timer interrupt check for image updates in class and display images to picture boxes

Class Functions:

TrainImages:

1. For each structure in passed list
   1.1. Pass data of individual structure to **TrainImagePrep**
2. Save AverageFilter created using **SaveFilter**
3. Exit Function

TrainImagePrep:

1. Load the image from address to a global variable
2. Convert the image to a greyscale version and store in a second global variable
3. If average filter does not exist
   3.1. Create average filter with null values with same row and column values as the loaded image
   3.2. Set a flag to determine that an average filter exists
4. Pass Greyscale image to **TrainImage**
5. Exit Function

TrainImage:

1. Pass point of interest to **CreateIdeal**
2. Pass greyscale original image to **ImageToFourier**
3. Pass ideal response image to **ImageToFourier**
4. Create PerfectFilter variable with same dimensions as Original Image
5. For all pixels in perfect filter
   5.1. PerfectFilter pixel = Ideal Fourier images pixel / Original Fourier images pixel
6. For all pixels in average filter
   6.1. AverageFilter pixel = (current value * scalar) + (scalar * PerfectFilter pixel)
7. Exit Function

ImageToFourier: Converts image into Fourier transformation using Libraries
FourierToImage: Convert Fourier image into a real domain image using Libraries

SaveFilter:

1. Open file stream to user provided text file
2. Output height of AverageFilter to the first line
3. Output width of AverageFilter to the second line
4. For all pixels in AverageFiler
   4.1. Output to newline as complex value
5. Exit Function

The primary function in the class is the TrainImage function which takes care of the perfect filter creation and the average filter creation. As the SaveFilter function saves the ASEF filter in its Fourier form there is no need to convert it in later applications.

9.1.3 Applying the filter

Applying the filter can be done either on a set of images or the live stream implemented previously, however the ASEF class needed two additional functions to first load the ASEF filter required from file and to apply the filter to an image provided and return a response image:

LoadFilter:

1. Open file stream to user provided text file
2. Read in first line and set data to AverageFilter height
3. Read in Second line and set data to AverageFilter width
4. For each line in text file
   4.1. Separate data into real and imaginary values
   4.2. Set AverageFilter pixels imaginary value relative to line number = imaginary import
   4.3. Set AverageFilter pixels real value to relative line number = real import
5. End function

TestImage:

1. Convert original image to Fourier size
2. Convert image to grayscale version
3. Convert to Fourier form using **ImageToFourier**
4. Create image Return of same dimensions as original
5. For all pixels in the image Return
   5.1. Pixel = Original Fourier pixel * AverageFilter pixel
6. Convert Return image back to Real Domain using **FourierToImage**
7. End Function

Through these functions, any image, as long as it has the same dimensions as the filter image, can be applied. Therefore the testing application needed to pass individual images through to the TestImage function after an average filter has been loaded into the class. Therefore the pseudocode for the main testing application is:

1. Decide on live stream or image sample
2. If (image sample)

2.1. Load images into list from users defined folder
2.2. For all images in list
    2.2.1. Pass image to **TestImage**
    2.2.2. If (median of response is less thatn pre set value)
        2.2.2.1. Search for brightest point in image
        2.2.2.2. Return point as point of interest
2.3. If (Live stream)
    2.3.1. For every frame of the live stream
        2.3.1.1. Pass Image to **TestImage**
        2.3.1.2. If (median of response is less that a pre set value)
            2.3.1.2.1. Search for brightest point
            2.3.1.2.2. Return point as object of intrest
3. End Analysis

This way the results of the ASEF filters could be analysed in real time to see how they functioned and if they were functioning correctly through picture boxes displaying the response.

9.1.4 Implementation

To implement the design of the ASEF system two key libraries have been used, primarily in the ASEF class. The two libraries were ExoCortex.DSP, which provided Fourier transform functionality, and EMGU for the fast conversion of images to gray scale. ExoCortex also provided a separate image class called CImage, which checked to make sure an image was of a suitable size for a Fourier transform to take place as well as stored each element of an image in a one dimensional array. This made pixel manipulation faster as only one loop was necessary to go through all the data in the image, as opposed to a double nested loop that would of gone through the rows and columns of the original image data type. The full ASEF class code can be found in Appendix 10.

Further functionality was also included within the testing algorithm to include a filter comparison view on single images as well as a peak highlighting algorithm to make the peaks of a response image more visible to a user.

9.2 Data Set Gathering

To test the capabilities of the ASEF filter, three simple objects were selected; a square, circle and triangle. Nine different filters were created as there are three different image types; the unprocessed original image of the camera, the unwrapped image and reversed threshold image.

A modified version of the user interface was developed from the blob detection algorithms discussed in 8.4 to acquire multiple test sets of images, such that when a button is pressed the current displayed within the application is saved to specific location. The unwrapped parameters have also be fixed to the same size for all images as if the size of the image changes the ASEF process will not be able to create an accurate ASEF filter. The types of images that were gathered and used can be seen in Figure 30, Figure 31 and Figure 32.

As the ASEF filter requires a large data set to learn an object, 100 sample images have been taken for each shape and image type for the purposes of training and a further 100 more for testing purposes. The object of intrest in each data set is also moved around within the image.

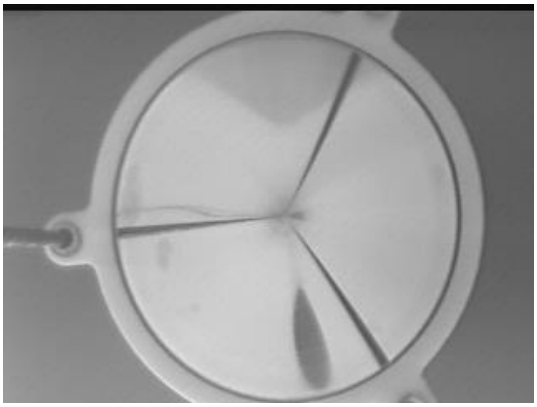


Figure 30. Typical non-preprocessed image of the conical lens.

Figure 31. Unwrapped image in greyscale.

Figure 32. Inverse threshold image.

A further three data sets have been created of 100 mixed images from the previous three testing data sets of original, unwrapped and threshold images, this is for the purposes of analysing the potential of identification as opposed to just detection testing as all three objects of interest is present within the data set.

9.3 Results of the ASEF Filters

Each filter was applied to a set of test images to test how accurate it was on the shapes it was trained on, as well compared to other filters trained on different shapes to see how good each filter was at detecting and identifying specific objects.

9.3.1 Circle Trained Filters

All three filters trained on the circular object show a response to the central point of a circle, however the intensities of the response of the filters were different for each type of image as can be seen in Figure 33, Figure 34 and Figure 35. The strongest response is from the threshold image, followed by the unwrapped image and then the original image. A response was expected from the unwrapped and threshold image, as the

unwrapping process of the original image reduces the distortion that can be seen on the object in the original image, however the original image filter has managed to learn some features of the circular object when distorted producing a response. This means that object detection can take place on the original image without the need of unwrapping the image, which takes up valuable computational power.

The accuracy rate of the original image is 93% as opposed to the threshold and unwrapped image which both have an accuracy of detection of 96%. It should also be noted that the median of each filter is very low, which is associated with a strong response, therefore all filters are suitable to use for the purpose of circular object detection.

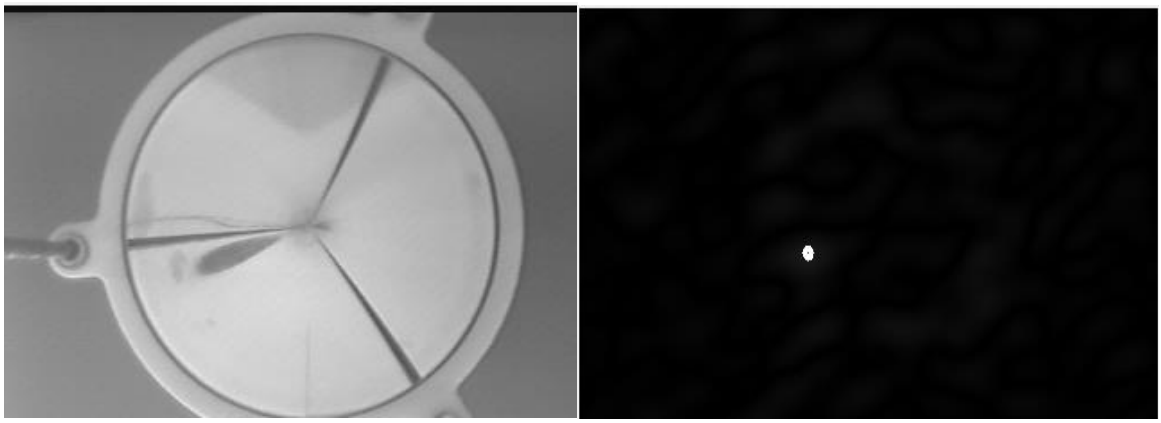


Figure 33. Response to the original image containing a circle.

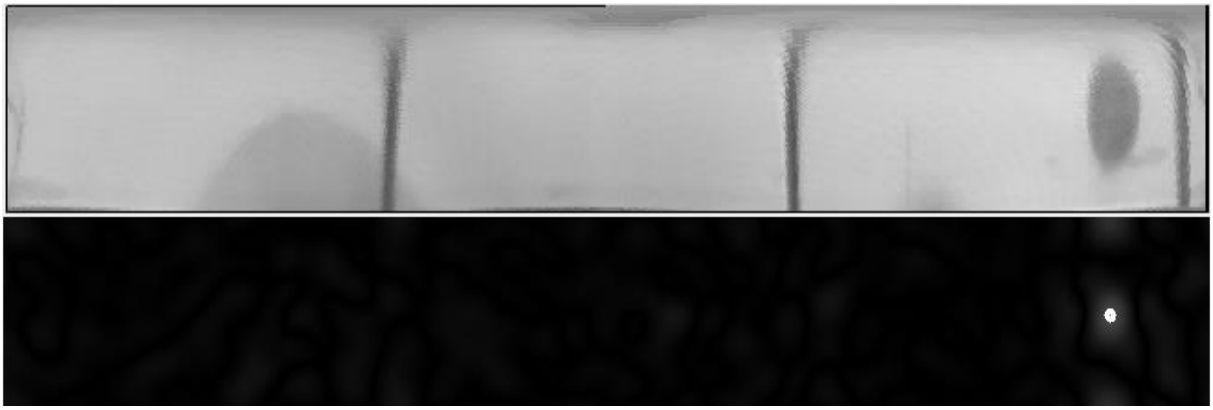


Figure 34. Response to the unwrapped image containing a circle.

Figure 35. Response to the inverse threshold image containing a circle.

9.3.2 Square Trained Filters

Similar results were seen for the square filters, as can be seen in Figure 36, Figure 37 and Figure 38. The accuracy results where 92% for the original image, 94% for the unwrapped image and 95% for threshold image. The images in which detection failed were when the square object was at a 45-degree angle to the horizon. Again, all filters work at a suitable accuracy for the purpose of detecting square like objects as the median is still low, except from in the original image filter response, this means that the response is not as strong but the brightest peak in the results is still consistent with the objects location.

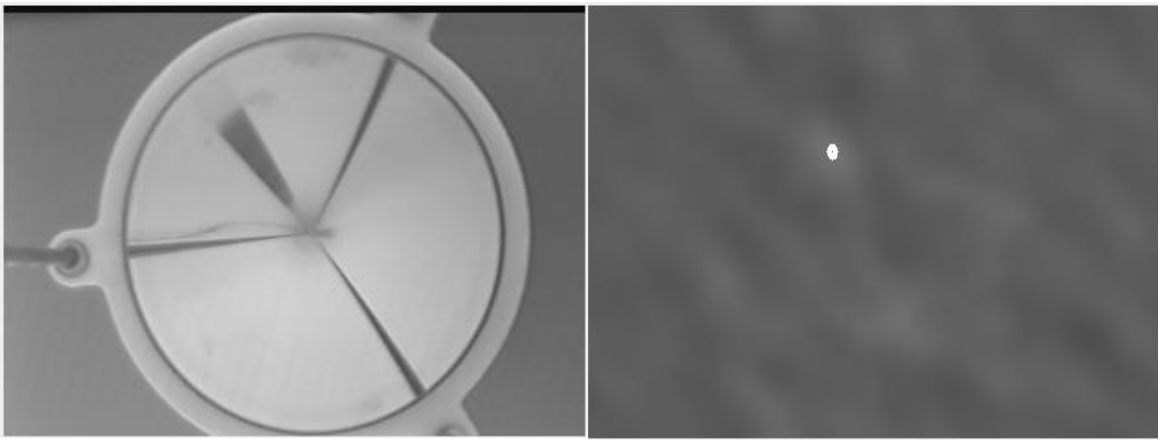


Figure 36. Response to the original image containing a square.

Figure 37. Response to the unwrapped image containing a square.

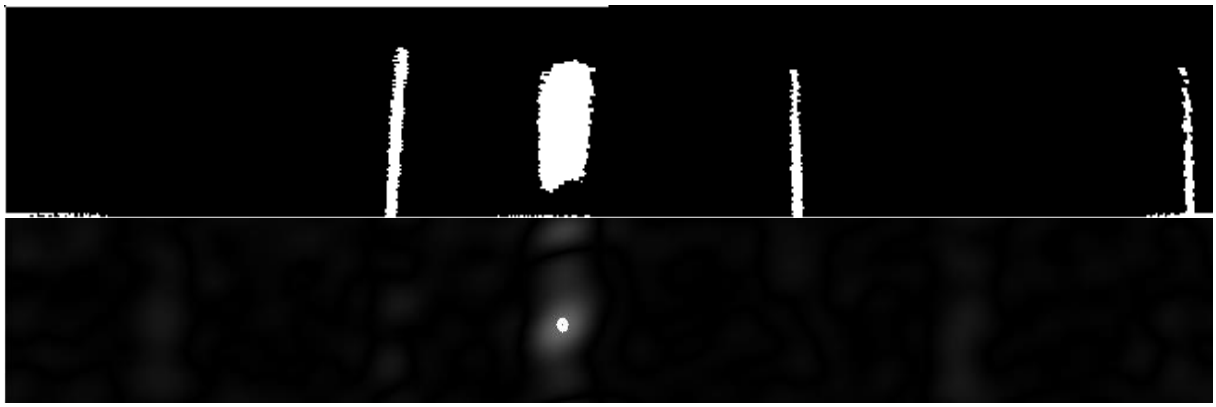


Figure 38. Response to the inverse threshold image containing a square.

9.3.3 Triangle Trained Filters

The triangular filters also provide accurate responses show in Figure 39, Figure 40 and Figure 41 at 92% for the original image, 94% for the unwrapped image and 93% for the threshold image. The images in which detection failed were when the triangular object was at a major angle to the horizon, which shows that rotation of objects can make them undetectable. The median in the original and unwrapped image is not as low as the threshold image, which again shows that the response to the object is not as strong as what it could be but the bright peaks of the response are still consistent with the objects position.

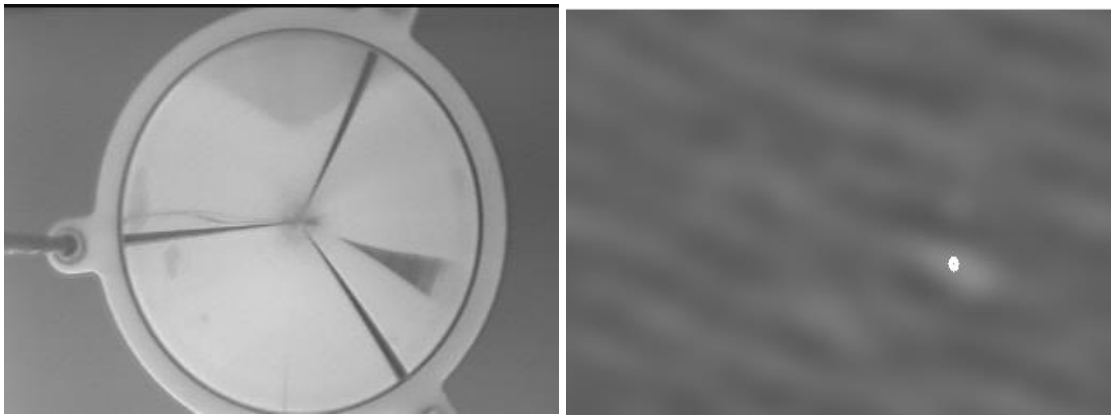


Figure 39. Response to the original image containing a triangle.

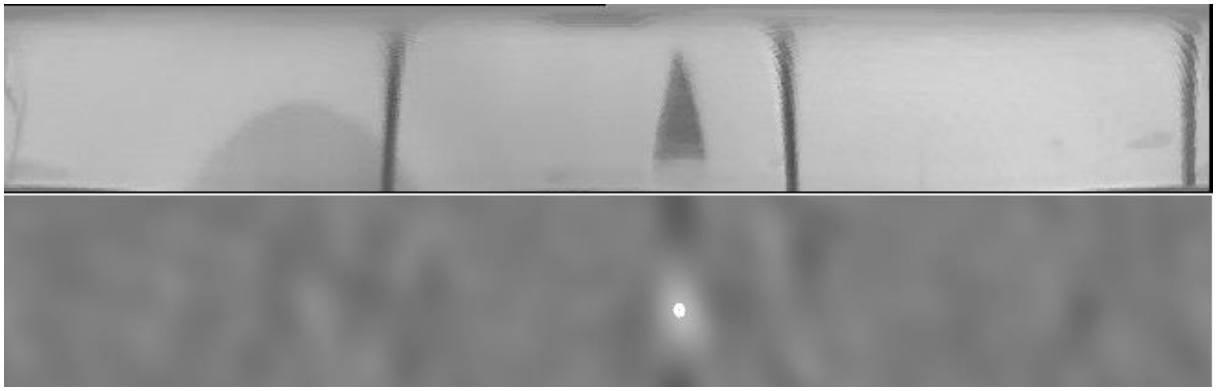


Figure 40. Response to the unwrapped image containing a triangle.

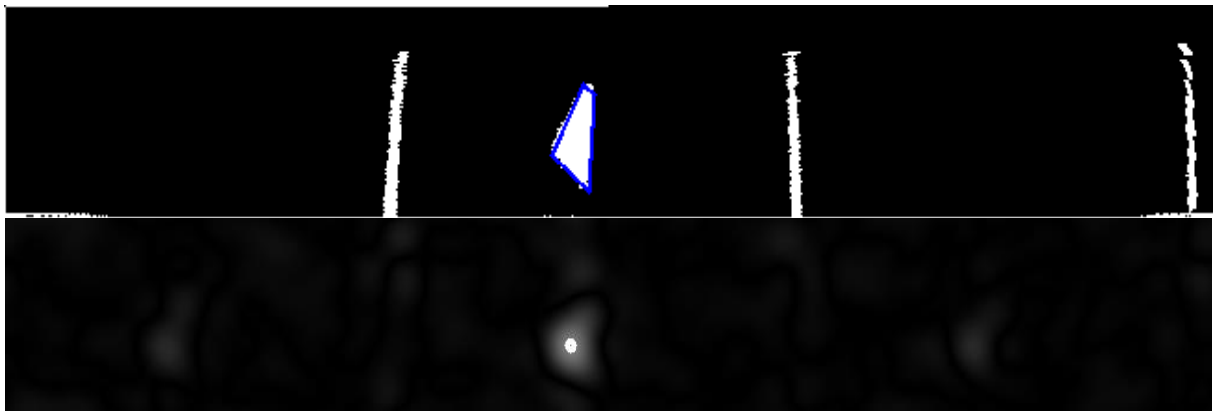


Figure 41. Response to the inverse threshold image containing a triangle.

9.3.4 Comparison of Original Image Filters

The results to different objects in the original image can be seen in Figure 42, which show that except when there is a major distortion due to an object being too close to the central point of the lens, all three filters will pick up on any object. This shows that identification of an object cannot be done through the original image displaying the full conical lens.

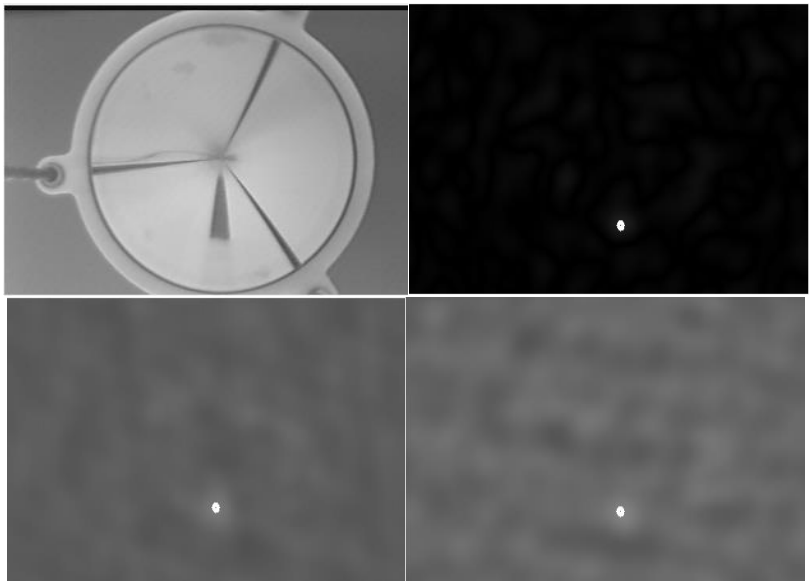


Figure 42. All three original image filters detecting a square object.

9.3.5 Comparison of Unwrapped Image Filters

The results of the unwrapped filters again show (Figure 43) that all filters can detect all object types; there is no indication from the response images that identification between the simple objects can be achieved.

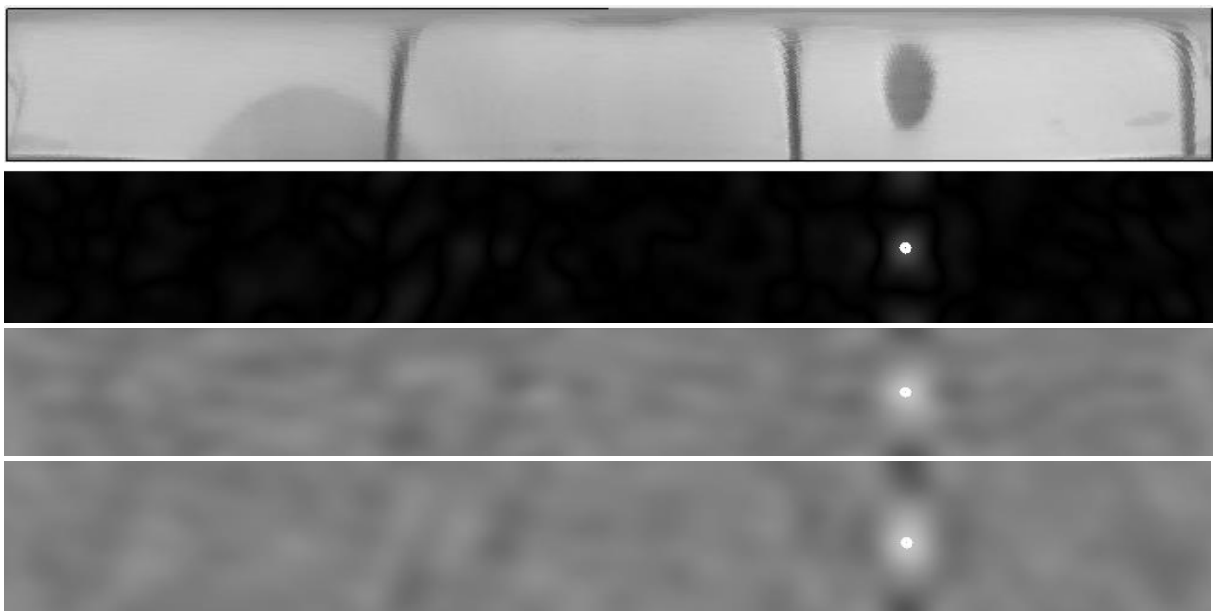


Figure 43. Unwrapped image and three corresponding filter responses in the order of circle, square and triangle.

9.3.6 Comparison of Threshold Image Filters

The final test on the threshold image filters concludes that identification in any image type using the trained filters is not possible as can be seen in Figure 44 as all filters provide a response to any object.

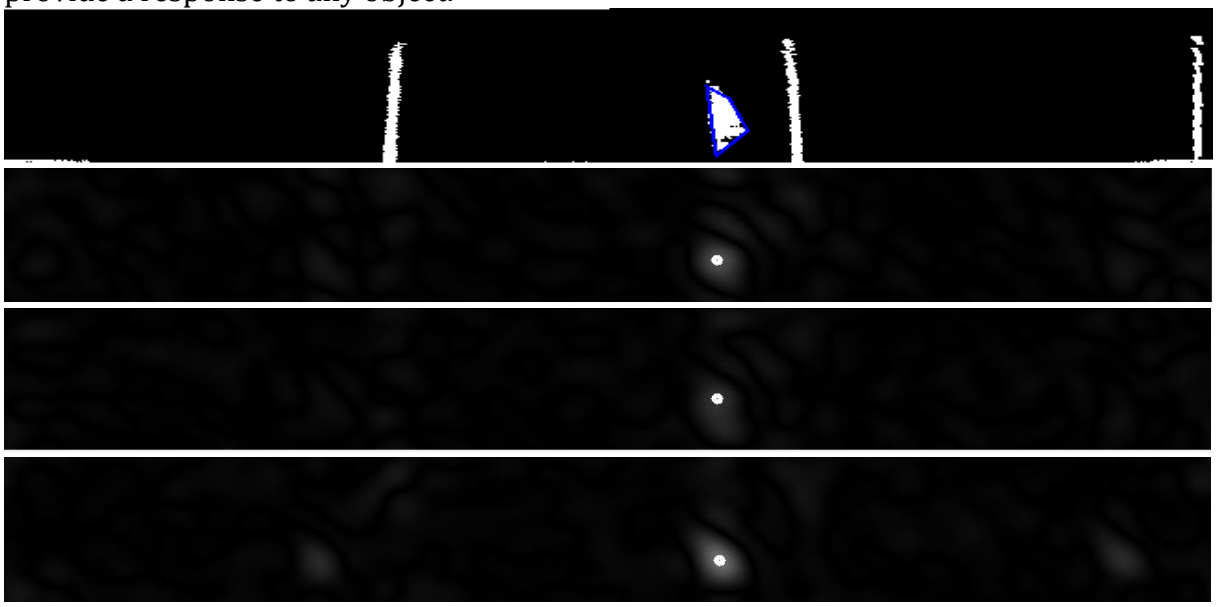


Figure 44. Inverse threshold image and three corresponding filter responses in the order of circle, square and triangle.

9.3.7 Real Time Application

The process of applying an ASEF filter takes approximately 0.16 seconds per frame, therefore if the filter was only applied to the original image for the purpose of object detection the operation would run at a frame rate of 6.25 per second. If the operation is run on the unwrapped image, the operation takes 2.4 seconds and 3.2 on the threshold image due to stacking processes. Therefore a suitable frame rate has been achieved on the basic original image, even if it is not at a real time rate.

10. Overall Results

As of the 10/04/2012 two successful object detection processes have been developed and proved to work significantly well, one of which can operate at a rate of 6.25 frames per second. Further work is however needed in fine-tuning the ASEF algorithm to see if it is capable of object identification as opposed to just detection of simple objects. This may be due to the resolution of the images that have been trained, or due to the orientation of the objects that have been present within the training images.

The design of the omnidirectional platform has proven suitable for testing purposes, however now that the system has been explored in terms of the position of camera and lens of the omnidirectional system, a more fixed and rigid design could be incorporated such as a clear Perspex tube to provide more stability when the platform is moving as well as a heavier base to also increase stability.

Within the time frame of the project, movement control for stability has not been implemented and through a simple motor test upon the platform, it was seen that it is necessary with the current platform to reduce the vibrations in the lens support structure. This is to prevent toppling of a top-heavy robot due to the weight of the lens and to maintain a still lens so that the central point in the vision algorithms is consistent with the real world lens central point.

The overall speed results of frame processing could be considered slow if a frame rate of 15fps was considered vital. Therefore in terms of producing an image process that can identify specific objects in real time the project has failed, however the vision processes explored are still relatively fast compared to other techniques and with better processing hardware and better library manipulation, the speed could still potentially reach a real world rate. The results of the ASEF filter on the original non pre-processed image also can be considered a success as any work in image identification within the field of omnidirectional vision runs image analysis on the unwrapped image.

11. Conclusion and Future work

The results have shown that the platform constructed is suitable for the testing process of an omnidirectional platform and can allow different hardware in the form of lens and camera to be attached to the platform. However, analysis of the platform while moving has shown that at high speeds it is unstable, due to the heavy lens positioned above the platform, which causes vibrations in the support rods. This as result affects the central point of the lens in the camera image, which then affects the unwrapping algorithm. Therefore, for future work in this area, as the platform and hardware used has been calibrated, a clear Perspex tube could be incorporated into the design to add support to the lens while increasing the weight of the base to increase ground stability. This will provide the basics for the project to advance with movement.

The results of the Connected Component Labelling blob detection method has shown that it can be used in controlled environments. However, the processing time of the algorithm within the system is delayed due to the unwrapping of the conical image and the inverse threshold process upon the image. This is primarily due to the number of image conversions within the code and the way in which elements of an image are accessed, therefore to improve the performance of this algorithm, an array based image storage method could be used to speed up element access, as well as designing a custom library incorporating all aspects of the CCL detection processes so that it is optimised to the project. The processes works correctly, it just needs to be adjusted to work in real time.

The ASEF filter has shown that it is highly suited to the process of object detection in an omnidirectional system, mainly because it can detect objects on the original non-preprocessed image so no pre-processing of the live video stream needs to take place before applying the algorithm, increasing the speed of detection. However, the filter has not been successful in differentiating between different objects in terms of identifying them as a specific shapes, something that it should have been able to do. The reason behind why it has not been able to has not been discovered, but could be due to the resolution of the image or the orientation of the objects in the training images not being

constant. However, this is an exploration area for future work, either to adapt the ASEF filter or find another suitable fast method for identifying different shapes of objects.

Therefore, as a whole, the project has implemented an ASEF filter in a novel way that could benefit omnidirectional robotic systems and has provided a good insight into vision processes for the author. Even though there is further work that can be done on the project, the results so far have provided a good foundation for other projects within the School of Systems Engineering.

12. Acknowledgements

The author acknowledges the work and guidance of the following individuals: Alex Angell, Joe Ball, Jonathon Boyle, Murray Evans, Tim Holt, Paul Minchinton, Dr Richard Mitchell, Max Parfitt and Brian Roantree.

13. Reference

[1] NiceVision. (2005, Mar.) Unattended Baggage Detection. [Online]. http://www.voiceproducts.com/media/files/NiceVision/NiceVision_Baggage_Detection_Brochure.pdf

[2] Kirokazu Kato, Seiji Inokuchi Atsushi Nakazawa, "Human Tracking Using Distributed Vision Systems," in *Fourteenth International Confrence on Pattern Recognition*, vol. 1, 1998, pp. 593-596.

[3] Kejian Yang,Feng Gao, Jun Li Hongxia Wang, "Normalization Methods of SIFT Vector for Object Recognition," in *Distrubuted Computing and Applications to Business, Engineering and Science (DCABES)*, Wuhan, China, 2011, pp. 175-178.

[4] Y.B. Fan, T.X. Zhang and H.S. Sang H.L. Zhao, "Stripe-based connected components labelling," *Electronics Letters*, vol. 46, no. 21, pp. 1434-1436, October 2010.

[5] Pyke Tin, Hiromitsu Hama Thi Thi Zin, "Building Multislit-HOG Features of Near Infrared Images for Pedistrian Detection," in *Innovative Computting, Information and Control (ICICIC)*, Osaka, Japan, 2009, pp. 302-305.

[6] Jibo Wei and Zhoawen Zhang, "A novel optical signal detection and processing method for swarm robot vision system," in *Robotics, Intelligent Systems and Signal Processing*, vol. 2, Changsha, China, 2003, pp. 1081-1085.

[7] Young-Ki Jung and Yo-Sung Ho Sung-Yeol Kim, "High-Resolution Depth Map Generation by Applying Stero Matching Based on Intial Depth Infomration," in *3DTV Conference: The True Vision - Capture, Transmission and Display of 3D Video*, Gwangju, 2008, pp. 201-204.

[8] Yuli Liu and Zuoliang Cao, "Omnidirectional Vision Tracking and Positioning for Vehicles," in *Natural Computation*, Tianjin, pp. 183-187.

[9] Elizabeth Fish. (2012, Feb.) Nano Quadrotors Swarm in Formation, Will Take Over Earth With Grace. [Online]. http://www.pcworld.com/article/249150/nano_quadrotors_swarm_in_formation_will_take_over_earth_with_grace.html

[10] Richard Harth. (2011, Dec.) ASU Biodesign Institute Sensing the Deep Ocean. [Online]. http://www.biodesign.asu.edu/news/sensing-the-deep-ocean-

[11] Ruzena Bajcsy and Shih-Schon Lin, "Single-View-point omnidirectional catadioptric cone mirror imager," *Patern Analysis abd Machine Intelligence*, vol. 28, no. 5, pp. 840-845, may 2006.

[12] Libor Spacek. Omnidirectional Catadioptric Vision System with Conical Mirrors. [Online]. http://essexrobotics.essex.ac.uk/TIMR03/bestpaper/spacek.pdf

[13] H Samet, "Efficent component labeling of images of arbitary dimension represented by linear bintrees," *Pattern Analysis and Machine Intelligence*, vol. 10, no. 4, pp. 579-586, July 1988.

[14] Bruce A. Draper and J.Ross Beveridge David S. Bolme, "Average of Synthetic Exact Filters," in *Computer Vision and Pattern Recognition*, Colarado, USA, 2009, pp. 2105-2112.

[15] Spy Camera CCTV. (2007) Wireless Pinhole Hidden Camera, USB Receiver, PC Plug and Play Spy Cam. [Online]. http://www.spycameracctv.com/spycamera/usb-receiver-cctv-wireless-pinhole-covert-hidden-spy-camera-system

[16] SparkFun. CMOS Camera - 640x480. [Online]. http://www.sparkfun.com/products/8667

[17] SparkFun. Bluetooth SMD Module - RN-41. [Online]. http://www.sparkfun.com/products/8497

[18] SparkFun. RFM22-S2 SMD Wireless Transceiver. [Online]. http://www.sparkfun.com/products/9581

[19] Scott Alexander, "MEng Project: Flying Brain," in *Scarp Procedings* , Reading, UK, 2011.

[20] Robot Electronics. SRF05 - Ultra-Sonic Ranger Technical Specification. [Online]. http://www.robot-electronics.co.uk/htm/srf05tech.htm

[21] Hauke Schmidt and Roland Klinnert Wolfgang M. Grimm, "Interference Cancelation in Ultrasonic Sensor Arrays by Stochastic Coding and Adaptive Filtering," in *IEEE International Conference on Itelligent Vehicles*, Stuttgart, Germany, 1998, pp. 376-389.

[22] SparkFun. (2012, Apr.) Infrared Proximity Sensor Long Range - Sharp GP2Y0A02YK0F. [Online]. http://www.sparkfun.com/products/8958

[23] Travis Deyle. (2011, Aug.) Hizzok: Swarmanoids: Foot-Bots, Hand-Bots, and Eye-Bots Cooperate to Win "Best Video" at AAAI 2011. [Online]. http://www.hizook.com/blog/2011/08/12/swarmanoids-foot-bots-hand-bots-and-eye-bots-cooperate-win-best-video-aaai-2011

[24] Dr Libor Spacek. (2008, Mar.) Omnidirectional Vision. [Online]. http://cswww.essex.ac.uk/mv/

[25] Daniel Monoz Arbeleda, Janier Arias Garcia, Carlos Lianos Quitero and Jose Motta Jones Yudi Mori, "FPGA-based image processing for omnidirectional vision on mobile robots," in *24th symposium on Integrated circuits and systems design*, New York, USA, pp. 113-118.

[26] EMGU. EMGU. [Online]. http://www.emgu.com/wiki/index.php/Talk:Tutorial

[27] OpenCV. Open CV. [Online]. http://opencv.willowgarage.com/wiki/

[28] Philip Pierce. (2003, July) C Sharp Webcam Capture. [Online]. http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1339&lngWId=10

[29] AForge. AForge.net. [Online]. http://www.aforgenet.com/

[30] OpenCV. cvBlobLib. [Online]. http://opencv.willowgarage.com/wiki/cvBlobsLib

[31] FFTW. Fastest Fourier Transform in the West. [Online]. http://www.fftw.org/

[32] Ben Houston. Exocortex.DPS. [Online]. http://www.exocortex.org/dsp/

14. Appendix 1 – Ultrasonic PIC Code

```
void Ultrasound_value_sort(){

	int i;
	LeftUltraPrev2 = LeftUltraPrev1;
	RightUltraPrev2 = RightUltraPrev1;
	LeftUltraPrev1 = LeftUltra;
	RightUltraPrev1 = RightUltra;
	LeftUltra= 0x00;
	RightUltra = 0x00;
	I2S();I2send(0xB8);I2send(0);I2SR();I2send(0xb9);

	for (i=1;i<=8;i++)
	{
		UltraRec[i]=I2GET(i!=8);
	}

	I2P();

	LeftUltra = UltraRec[1]<< 8;
	LeftUltra = LeftUltra + UltraRec[2] ;
	RightUltra = UltraRec[3]<< 8;
	RightUltra = RightUltra + UltraRec[4];

	UltraLeftAverage = LeftUltraPrev2+LeftUltraPrev1 + LeftUltra / 3;
	UltraRightAverage = RightUltraPrev2+RightUltraPrev1+RightUltra /3;

	BroadcastPacketAR[0] = 18;
	BroadcastPacketAR[1] = UltraLeftAverage;
	BroadcastPacketAR[2] = UltraLeftAverage << 8;
	BroadcastPacketAR[3] = UltraRightAverage;
	BroadcastPacketAR[4] = UltraRightAverage << 8;
	Broadcast = 5;
}
```

15. Appendix 2 – Tachometer PIC Code

```
void __attribute__((interrupt,auto_psv))  _DMA3Interrupt(void)
{
 if (hist3<0+OPTOoffset)
        {
        if (adDMAbuf[6]<hist3)
                {
                        hist3=10+OPTOoffset;pos3+=lastdir3;
                }
        }
        else
        {
        if (adDMAbuf[6]>hist3)
                {
                        hist3=-10+OPTOoffset;pos3+=lastdir3;
                }
        }
 if (hist2<0+OPTOoffset)
        {
        if (adDMAbuf[4]<hist2)
                {
                        hist2=10+OPTOoffset;pos2+=lastdir2;
                }
        }
        else
        {
        if (adDMAbuf[4]>hist2)
                {
                        hist2=-10+OPTOoffset;pos2+=lastdir2;
                }
        }
 if (hist1<0+OPTOoffset)
        {
        if (adDMAbuf[5]<hist1)
                {
                        hist1=10+OPTOoffset;pos1+=lastdir1;
                }
        }
        else
        {
        if (adDMAbuf[5]>hist1)
                {
                        hist1=-10+OPTOoffset;pos1+=lastdir1;
                }
        }

_DMA3IF=0;
}
```

16. Appendix 3 – PIC Movement Control

```
void Ultrasound_avoid(){


        if (UltraLeftAverage < 160 && UltraRightAverage < 160){
                if(UltraLeftAverage < UltraRightAverage){
                        speed1 = 127;
                        speed2 = 40;
                }
                if (UltraLeftAverage >= UltraRightAverage){
                        speed1= 40;
                        speed2= 127;
                }
                CheckFlag = 1;
        }

        if (UltraLeftAverage < 160 && CheckFlag !=1){
                speed1 = 20;
                speed2=127;
        }

        if (UltraRightAverage < 160 && CheckFlag !=1){
                speed1 = 127;
                speed2 = 20;
        }

        if (UltraRightAverage >=160 && UltraLeftAverage >=160 && CheckFlag !=1){
                speed1 = -60;
                speed2 = -60;
        }
        if (speed1<0)
        {
                OC2CON=0;OC2R=OC2RS=0;
                OC5RS=abs(speed1)*16;
                if (!OC5CON) OC5CON=6;
//                                                              LATD|=0x0010;
                lastdir1=1;
        }
        if (speed1>0)
        {
                OC5CON=0;OC5R=OC5RS=0;
                OC2RS=abs(speed1)*16;
                if (!OC2CON) OC2CON=6;
//                                                              LATD|=0x0002;
                lastdir1=-1;
        }
        if (speed2<0)
        {
                OC4CON=0;OC4R=OC4RS=0;
                OC3RS=abs(speed2)*16;
                if (!OC3CON) OC3CON=6;
```

```
        //                                                      LATD|=0x0004;
                lastdir2=1;
        }
   if (speed2>0)
        {
                OC3CON=0;OC3R=OC3RS=0;
                OC4RS=abs(speed2)*16;
                if (!OC4CON) OC4CON=6;
//                                                      LATD|=0x0008;
                lastdir2=-1;
        }

        CheckFlag = 0;
}
```

17. Appendix 4 – Comparison of different lens sizes

θy	Lens Height	Lens Radius	Third Side	Y	X	Θx = 5	Θx = 10	Θx = 20	Θx = 30	Θx = 40	Θx = 50	Θx = 55	Θx = 60
5.00	3.06	35.00	35.13	41.00	20.00	18.10	25.74	42.58	63.15	90.91	133.73	166.59	214.79
6.00	3.68	35.00	35.19	41.00	20.00	20.40	28.18	45.50	67.01	96.71	144.10	181.82	239.33
7.00	4.30	35.00	35.26	41.00	20.00	22.75	30.70	48.56	71.13	103.07	155.99	199.86	269.90
8.00	4.92	35.00	35.34	41.00	20.00	25.17	33.29	51.77	75.54	110.11	169.77	221.60	309.09
9.00	5.54	35.00	35.44	41.00	20.00	27.64	35.98	55.14	80.30	117.94	185.97	248.35	361.23
10.00	6.17	35.00	35.54	41.00	20.00	30.20	38.77	58.71	85.45	126.72	205.30	282.12	434.07
11.00	6.80	35.00	35.66	41.00	20.00	32.83	41.67	62.50	91.07	136.67	228.82	326.16	543.16
12.00	7.44	35.00	35.78	41.00	20.00	35.57	44.70	66.54	97.22	148.06	258.11	386.06	724.74
13.00	8.08	35.00	35.92	41.00	20.00	38.41	47.88	70.86	104.00	161.25	295.63	472.41	1087.56
14.00	8.73	35.00	36.07	41.00	20.00	41.37	51.23	75.51	111.55	176.72	345.51	607.89	2175.39
15.00	9.38	35.00	36.23	41.00	20.00	44.47	54.77	80.55	120.00	195.18	415.17	851.47	
16.00	10.04	35.00	36.41	41.00	20.00	47.73	58.52	86.02	129.57	217.61	519.46	1419.33	
17.00	10.70	35.00	36.60	41.00	20.00	51.16	62.51	92.01	140.50	245.52	693.00	4257.28	
18.00	11.37	35.00	36.80	41.00	20.00	54.79	66.78	98.62	153.13	281.25	1039.70		
19.00	12.05	35.00	37.02	41.00	20.00	58.65	71.38	105.95	167.95	328.72	2079.03		
20.00	12.74	35.00	37.25	41.00	20.00	62.77	76.34	114.15	185.61	394.98			
21.00	13.44	35.00	37.49	41.00	20.00	67.18	81.74	123.43	207.04	494.12			
22.00	14.14	35.00	37.75	41.00	20.00	71.95	87.65	134.01	233.68	659.03			
23.00	14.86	35.00	38.02	41.00	20.00	77.11	94.15	146.24	267.76	988.40			
24.00	15.58	35.00	38.31	41.00	20.00	82.73	101.37	160.56	313.00	1975.58			

- All distances are in mm and all angles are in Degrees.

18. Appendix 5 – Unwrapping function using C# standard libraries

```
public void Unwraped_Process()
{
     // Get image variables from main program
     int Diameter = BaseStationGUI.Diameter;
     int centrecol = BaseStationGUI.centrecol;
     int centrerow = BaseStationGUI.centrerow;

     // Get image from main program
     Bitmap Original = new Bitmap(BaseStationGUI.Image_Input);

     // calculate the circumference of the lens given the diameter
     circumference = (int)Math.Round(2 * Math.PI * (Diameter / 2) + 1);

     // create image of correct size for the unwrapped image
     Bitmap Unwrap = new Bitmap(circumference + 2, (int)(Diameter / 2) + 2);

     // move counter r along the radius of the lens
     for (double r = (Diameter / 2); r > 0; r -= 0.5)
     {

          // first half of the circle

          // variable to store value of pixels
          Color C = Color.Black;

          for (double theta = Math.PI; theta > -Math.PI / 2; theta -= (2 * Math.PI /
          BaseStationGUI.thetacalc))
          {
               // null pixel value is set to color black
               C = Color.Black;

               // calculate pixel positions of original image
               rpix = (int)Math.Round(r * Math.Cos(theta) + centrecol);
               cpix = (int)Math.Round(r * Math.Sin(theta) + centrerow);

               // calculate new pixel positions in unwrapped image
               newcpix = (int)Math.Round((circumference / 2) + (2 * Math.PI *
               (Diameter / 2)) * ((Math.Abs(theta - (Math.PI / 2))) / (2 * Math.PI)));
               newrpix = (int)(Math.Round(r));

               // Make sure pixel position is valid and get pixel value
               if (rpix < Original.Height && cpix < Original.Width)
               {
                    if (rpix > 0 && cpix > 0)
                    {
                         C = (Original.GetPixel(cpix, rpix));
                    }
               }

               // Set pixel value in unwrapped image
               Unwrap.SetPixel(newcpix, newrpix, C);
           }

           // for second half of the circle

          for (double theta = Math.PI / 2; theta <= Math.PI * 3; theta += (2 * Math.PI
          / BaseStationGUI.thetacalc))
          {
               // null pixel value is set to color black
               C = Color.Black;
```

```
                // calculate pixel positions of original image
                rpix = (int)Math.Round(r * Math.Cos(theta) + centrecol);
                cpix = (int)Math.Round(r * Math.Sin(theta) + centrerow);

                // calculate new pixel positions in unwrapped image
                newcpix = (int)Math.Round((circumference / 2) - (2 * Math.PI *
                (Diameter / 2)) * ((Math.Abs(theta - Math.PI / 2)) / (2 * Math.PI)));
                newrpix = (int)Math.Round(r);

                // Make sure pixel position is valid and get pixel value
                if (rpix < Original.Height && cpix < Original.Width)
                {
                      if (rpix > 0 && cpix > 0)
                      {
                            C = (Original.GetPixel(cpix, rpix)
                      }
                }

                // Set pixel value in unwrapped image
                Unwrap.SetPixel(newcpix, newrpix, C);

                }
      }

      // Send image back to main program
      BaseStationGUI.Returned_Unwrap = Unwrap;
      BaseStationGUI.recv = true;
}
```

19. Appendix 6 – Unwrapping Algorithm in EMGU

```
public void Unwraped_Process()
{

     // convert image from standard library to EMGU library format
     Image<Bgr, Byte> Original = new Image<Bgr, Byte>(OriginalBit);

     // calculate the circumference of the lens given the diameter
     circumference = (int)Math.Round(2 * Math.PI * (diameter / 2) + 1);

     // create image of correct size for the unwrapped image
     Unwrap = new Image<Bgr, Byte>(circumference + 2, (int)(diameter / 2) + 2);

     // variable to store value of pixels
     Color Color_set;

     for (double r = (diameter / 2); r > 0; r -= 0.5)
     {
          // first half of the circle

          for (double theta = Math.PI; theta > -Math.PI / 2; theta -= (2 * Math.PI /
          BaseStationGUI.thetacalc))
          {
               // null pixel value is set to color black
               Color_set = Color.Black;

               // calculate pixel positions of original image
               rpix = (int)Math.Round(r * Math.Cos(theta) + centercol);
               cpix = (int)Math.Round(r * Math.Sin(theta) + centerrow);

               // calculate new pixel positions in unwrapped image
               newcpix = (int)Math.Round((circumference / 2) + (2 * Math.PI *
               (diameter / 2)) * ((Math.Abs(theta - (Math.PI / 2))) / (2 * Math.PI)));
               newrpix = (int)(Math.Round(r));

               // Make sure pixel position is valid and get pixel value
               if (rpix < Original.Height && cpix < Original.Width)
               {
                    if (rpix > 0 && cpix > 0)
                    {
                         Color_set = Color.FromArgb(Original.Data[rpix - 1, cpix - 1,
                         0], Original.Data[rpix - 1, cpix - 1, 1], Original.Data[rpix
                         - 1, cpix - 1, 2]);
                    }
               }

               // Set pixel value in unwrapped image
               Unwrap.Data[newrpix, newcpix, 2] = Color_set.R;
               Unwrap.Data[newrpix, newcpix, 1] = Color_set.G;
               Unwrap.Data[newrpix, newcpix, 0] = Color_set.B;

          }

          // for second half of the circle
          for (double theta = Math.PI / 2; theta <= Math.PI * 3; theta += (2 * Math.PI
          / BaseStationGUI.thetacalc))
          {
               // null pixel value is set to color black
               Color_set = Color.Black;
```

```
                // calculate pixel positions of original image
                rpix = (int)Math.Round(r * Math.Cos(theta) + centercol);
                cpix = (int)Math.Round(r * Math.Sin(theta) + centerrow);

                // calculate new pixel positions in unwrapped image
                newcpix = (int)Math.Round((circumference / 2) - (2 * Math.PI *
                (diameter / 2)) * ((Math.Abs(theta - Math.PI / 2)) / (2 * Math.PI)));
                newrpix = (int)Math.Round(r);

                // Make sure pixel position is valid and get pixel value
                if (rpix < Original.Height && cpix < Original.Width)
                {

                     if (rpix > 0 && cpix > 0)
                     {
                          Color_set = Color.FromArgb(Original.Data[rpix - 1, cpix - 1,
                          0], Original.Data[rpix - 1, cpix - 1, 1], Original.Data[rpix
                          - 1, cpix - 1, 2]);
                     }
                }

                //Check pixel values for the set function are greater than 0
                if (newrpix - 1 < 0 || newcpix - 1 < 0)
                {
                     newcpix = 0;
                     newrpix = 0;
                }

                // Set pixel value in unwrapped image
                Unwrap.Data[newrpix, newcpix, 2] = Color_set.R;
                Unwrap.Data[newrpix, newcpix, 1] = Color_set.G;
                Unwrap.Data[newrpix, newcpix, 0] = Color_set.B;

           }

     }

     // send back to main program using EMGU's convert image function ".ToBitmap"
     BaseStationGUI.Returned_Unwrap = Unwrap.ToBitmap();
     BaseStationGUI.recv = true;
}
```

20. Appendix 7 – Reverse Threshold Algorithm

```
public void Reverse_UnwrappedImage()
{
     // convert image to grayscale in EMGU
     Image <Gray, Byte> OriginalGray = Unwrap.Convert<Gray, Byte>();

     // create result image with correct sizes
     ThersholdFlip = new Image<Gray, Byte>(OriginalGray.Width, OriginalGray.Height);

     // variables for pixel value checks
     Gray GrayValue;
     int GrayIntensity = 0;

// go through image pixels firstly by the height of the image (vertically)
     for (int i = 0; i < OriginalGray.Height; i++)
     {
          // Go through the image secondly by the width of the image (horizontally)
          for (int j = 0; j < OriginalGray.Width; j++)
          {
               // get the pixel from the image in char form
               GrayValue = OriginalGray[i, j];

               // convert char to an int between 0 and 255
               GrayIntensity = (int)GrayValue.Intensity;

               // check value of the int and set the value in the results image to
               either 0 or 255 depending on the check
               if (GrayIntensity > 127)
              {
                    ThersholdFlip[i, j] = new Gray(0);
               }
               else
               {
                    ThersholdFlip[i, j] = new Gray(255);
               }
          }
     }

// send image back to main program
BaseStationGUI.Reverse_Threshold = ThersholdFlip.ToBitmap();
BaseStationGUI.recx = true;

}
```

21. Appendix 8 – Blob Detection Algorithm

```
public void Blob_Search()
{
     // create blobCounter class
     BlobCounter blobCounter = new BlobCounter();

     // eliminate noisy blobs by setting size restrictions

     blobCounter.FilterBlobs = true;
     blobCounter.MinHeight = 10;
     blobCounter.MinWidth = 20;
     blobCounter.MaxWidth = 130;
     blobCounter.MaxHeight = 130;

     // process image
     blobCounter.ProcessImage(ThersholdFlip.ToBitmap());

     // get info
     Blob[] blobs = blobCounter.GetObjectsInformation();

     // create Graphics object to draw on the image and a pen

     Graphics g = Graphics.FromImage(test);
     Pen bluePen = new Pen(Color.Blue, 2);

     // draw all blobs
     for (int i = 0, n = blobs.Length; i < n; i++)
     {
          // add all points of an edge of a blob to a list
          List<IntPoint> edgePoints = blobCounter.GetBlobsEdgePoints(blobs[i]);

          // create a list points od a square that connects for corner points
          List<IntPoint> corners = PointsCloud.FindQuadrilateralCorners(edgePoints);

          // draw the individual blob on the result image
          g.DrawPolygon(bluePen, PointsListToArray(corners));
          blobfound = true;

     }
}
```

22. Appendix 9 – ASEF Class Design

```
public class ASEFFilter
{
     // Set of images created and used throughout the training processes that need to
     be global
     Image<Bgr, Byte> OriginalImage = null;
     Image<Gray, Byte> OriginalGray = null
     CImage OriginalFourier;
     Image<Gray, Byte> Ideal;
     CImage IdealFourier;
     CImage PerfectFilter;
     CImage AverageFilter;
     CImage ReturnTestFourier;
     CImage ReturnTestReal;

     // Structure used for storing the information required for training
     public struct ASEFInput
     {
          public string filename;
          public List<Point> centres;
     }

     // Set of functions to separate each image, pre processes the image into a usable
     form and run the training process
     public void TrainImages(List<ASEFInput> input, string SaveLoc)
     private void TrainImagePrep(ASEFInput input)
     private void TrainImage(List<Point> POI, string filename)
     private Bitmap ResizeImageText(string filename)
     private Bitmap ResizeImageImage(Bitmap image)

     // create the ideal image by creating a gaussium point from the x and y cords
     provided
     private void CreateIdeal(List<Point> centres, string filename)

     // Fourier Transform and Inverse Fourier transform functions
     private CImage ImageToFourier(Image<Gray, Byte> Original)
     private void FourierToImage(CImage cimage)

     // Apply the ASEF filter created an image
     public void TestImage(Bitmap input)

     // Store and load the ASEF filter
     public void SaveFilter(string SaveLoc)
     public void LoadFilter(string LoadLoc)
}
```