Guide to Star Tracker Program

Table of Contents

[1 Obtaining Image 2](#_Toc35257750)

[2 Getting ‘BLOB’s from image 3](#_Toc35257751)

[2.1 Thresholding 3](#_Toc35257752)

[2.1.1 Global Image Thresholding 3](#_Toc35257753)

[2.1.2 Adaptive Thresholding 3](#_Toc35257754)

[2.1.3 Choosing 4](#_Toc35257755)

[2.1.4 Implementation 5](#_Toc35257756)

[3 The implementation of the Pyramid Method 6](#_Toc35257757)

[4 Searching the database 7](#_Toc35257758)

# 1 Obtaining Image

# 2 Getting ‘BLOB’s from image (image\_processing)

## 2.1 Thresholding

To derive the ‘relative’ brightness of stars and their positions, the background must first be removed.

### 2.1.1 Global Image Thresholding

This is when the whole image is sampled for a histogram, everything bellow a certain value will be set to 0.

It is really fast but less effective for gradient backgrounds.

Otsu’s method is common for doing this.

It expects a foreground and a background which will be two heaps in the histogram and cuts everything out from the first heap.

It does not work when there are no obvious heaps

‘Global percentage’ is when you make a brightness histogram and find what brightness band exceeds this. Everything below should be set to 0.

### 2.1.2 Adaptive Thresholding

This is when you take a sampled average of the pixels around a pixel and if it is less then the average, it will set it to 0.

It is very slow however it can be far more accurate.

This works in 3 methods; the mean, the median, the mean min-max: .

The mean min max works best for this.

|  |  |  |
| --- | --- | --- |
| Method | Pros | Cons |
| Otsu’s | Fast. | Only works when the background brightness is always bellow the foreground brightness. And there is a distinct foreground and background. |
| Global Percentage | Fast.  It will cut the background. | Cuts out some of the foreground.  May need calibrating. |
| Adaptive Thresholding | Usually more successful for varying backgrounds. | Very slow on large images. |

### 2.1.3 Choosing

Looking at the tested sample, this would be the worst case. Percentage threshold worked well once it was calibrated, however, adaptive thresholding did not manage even when turning the sample area up high. With this image, it is a very large file size making percent threshold more favourable.

After testing with an actual photo taken with the camera we are considering using, the percentage threshold had to be changed to 99.99% otherwise it would not work.

Adaptive thresholding worked at 10 sample area and 99.9% however it was taxing on the computer.

It is likely that using percent threshold would be the most ideal, however, it will need to be calibrated initially.



|  |  |  |
| --- | --- | --- |
|  | Percent Threshold | Adaptive Threshold |
| When too low | 99% | Sample area = 5, Exclude: 99.9% |
| When too high | 99.99% | Does not work. |
| When ideal | 99.5% | Does not work. |

### 2.1.4 Implementation

#### Adaptive Threshold:

This method copies the image, it will then loop through every pixel. For each pixel, it will find the min and max pixels around it in the specified radius. It will calculate the average by:

If the value is smaller than T, it will be set to 0.

#### Percentage Threshold:

This requires two methods:

* generateHistogram

This will create an array of values corresponding to the number of pixels fitting a specific colour range.

* percentThreshold

This will count through all the color bands in the array from ‘generateHistogram’ and when it gets to a value greater than the percentage specified, it will return this value.

Later in the code, you will need to request a baseline brightness for blob detection, enter this value and everything bellow will be ignored.

## 2.2 Blob Detection

For detecting the blobs, I used the grass fire method: <http://what-when-how.com/introduction-to-video-and-image-processing/blob-analysis-introduction-to-video-and-image-processing-part-1/>.

This will start at the first pixel exceeding a certain brightness, it will then add all the surrounding pixels above the brightness around it to a queue and eventually check them all until the queue is empty. While it does this, it sets the pixels to 0 so it won't be used in the future.

This implementation in the program uses several methods:

* findBlobs

This moves from the top to the bottom of the image wiping from left to right.

If a pixel exceeds the background brightness, it will call ‘spreadGrassFire’.

* spreadGrassFire

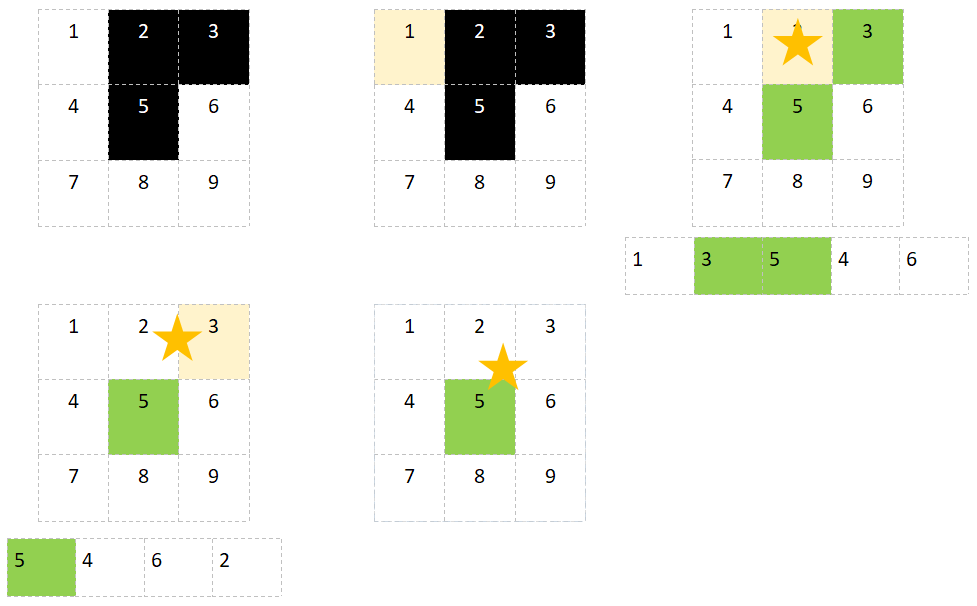
This will have an input pixel location.

At every pixel location, it will check that it is brighter than the background.

If it is, it will recalculate the centroid:

It will set the current pixel to 0.

It will then find add all the pixels around it and repeat the process.



# 3 The implementation of the Pyramid Method (star\_tracker)

## 3.1 Advised Method

The pyramid method requires 4 stars.

The brightest star is found (pilot or **r**).

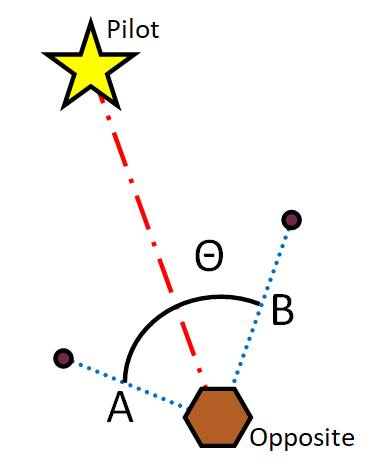
The furthest star in the selection is then found (opposite or **j**).

The angle between A (**i**) and B (**k**) is found.

This is repeated over many sets.

The sets are then compared to a database.

The most likely location is returned.



## 3.2 Implementation

Due to the processing power of a microcontroller, the cost for accuracy is processing time. This means using a high-resolution camera may be beneficial, however, it is completely impractical to run on a microcontroller.

With low-resolution camera’s, there is the problem that not many stars will be recognised, some stars may overlap and the accuracy of the angle will never be exact.

There were many theoretical solutions we decided on:

* Stars will not be recognised:

It is desirable to have 20 to 30 degrees FOV with high powered cameras, this slightly amplifies the brightness of the stars and has a more accurate positioning.

Cameras of low resolution are not usually designed good enough for the dark and will have noise.

Because of this, we decided to increase the FOV to ~600, this allowed us to have far more high magnitude stars in the shot allowing the camera to pick them up stars easier over the noise.

* Some stars will overlap:

When stars are close to each other, they will be recognised as one star. This may make two dull stars register as a pilot.

To counteract this, we decided to use not just the angle but also a distance ratio (mentioned later).

* Accuracy of angles:

The ideal method finds sets of stars close to each other in the frame, this works well as the number of pixels approach infinity. However, with limited pixels, the distance between pixels becomes larger. Because of this, it is ideal to try to keep the stars as apart as possible, with a larger distance from each other, the more accurate the angle will be.

To do this we decided to use the brightest stars in the image which are naturally scattered.

# 4.0 Searching the database (database)

This requires finding close matches and the probability of each being a star in the image.

## 4.1 Searching

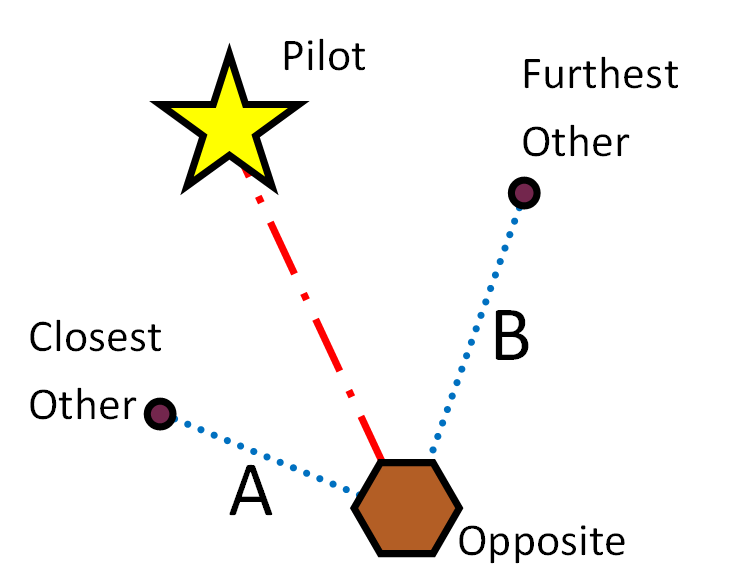
This part of the program will search through the database to find close matches. To increase the chances of the correct match, these are:

* Close Angle

If the angle of the opposite star is similar, include it in the list.

* Close Ratio

This is based on the relative distance of the stars that are not the pilot. The ratio is the (futhest distance from opposite) / (closest distance from opposite).



Since distance and angle are unique from each other, this allows the program to narrow down the search.

## 4.2 Probability

This is to find the most likely element of the database to be a match. It works in a way of probability which is an arbetory number calculated, the higher the number the more likely. It is done by:

# 5. Creating the database