DEPARTMENT OF COMPUTER SCIENCE

Penguin Head Counting using Real-time Object Recognition

Naiwen Zhang

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering

September 2010 | CSMSC-10

Declaration

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science in the Faculty of Engineering. It has not been submitted for any other degree of diploma of any examining body. Except where specifically acknowledged, it all the work of the Author.

Naiwen Zhang, September 2010

Abstract

This thesis presents a vision system that allows for real-time head counting of African penguins in wildlife video. African penguins (*Spheniscus demersus*), living on the south coast of Africa, have broad white "C" shaped markings on their black heads and pink glands above the eyes. We modeled this characteristic pattern on the penguin head using machine learning algorithms on large sets of real-world data. The finished classification system takes a bird-view video stream of African penguins walking by in their natural habitats as input. Then, by using computer vision techniques, the system detects the penguin heads in the video and tracks them until they disappear from view. Based on a combination of detection and tracking algorithms [25], the system counts the number of penguins passing the field of view. The system outputs a video, in which the detected penguin heads are marked and the number of penguins is displayed.

The underlying technique used for detecting penguin heads in video streams is a real-time object detection framework introduced by Viola and Jones [1] applied to a novel task.

- I collected 3,366 negative samples and 18,569 positive penguin head samples by labeling the penguin heads manually in more than six thousand original images taken in Bristol Zoo Gardens, see pages 29-31.
- I trained five classifiers (front, front side, side, side back, back) with cascade structure for multi-poses classification (five penguin head poses) utilizing the AdaBoost learning algorithm [1] [4], see pages 31-34. I tested the performance of the five classifiers by using 662 positive test samples and 200 negative test samples. The average GAR of the first four classifiers reaches 91.38% while the average FAR per window is 1.6570E-07, see 33-34.
- I extended the pose space to a full 360 degrees coverage (at given camera elevation) exploiting the physiological symmetry of African penguins and integrated multiple detections, see pages 34-37.

The implementation of tracking and counting moving penguin heads is based on feature selection and feature tracking.

- I used a corner detector proposed by Jianbo Shi and Carlo Tomasi [17] to choose good features for tracking (see pages 38-39) and used a pyramidal implementation of the Kanade-Lucas-Tomasi Feature tracker [19] [22] [23] to track the selected features from frame to frame (see pages 40-41).
- By combining the object detection and features tracking [25] and adapting it to the application at hand, I implemented penguin head tracking so that it works robustly for wildlife video, see pages 41-45.
- Based on the penguin detection and penguin tracking, I implemented the real-time penguin head counting and achieved an average counting accuracy of 91.96%, see pages 46-54.

Acknowledgements

First of all, I would like to express the deepest appreciation to my supervisor, Dr. Tilo Burghardt for giving me the opportunity to work in such an interesting area, and for his guidance and support throughout this project.

Secondly, I would like to thank the markers to this project, Dr. Andrew Calway and Dr. Erik Reinhard. Their questions and comments during the presentation reminded me of the inadequacies of my work, thereby making this thesis stronger.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of the project.

Table of contents

1. Introduction and motivation

1.1 General Introduction

In recent years, biometric recognition technologies, as important applications of image processing, have attracted increasing attention from various fields of society. At the beginning, they were mainly developed to help identify human beings. Human faces recognition and human fingerprints recognition are the most two representative applications. However, these technologies can also help to recognize, identify and track wild animals in their natural habitats, such as African penguins (*Spheniscus demersus*).

This provides an opportunity for automatic remote monitoring these birds without disturbing them. An interdisciplinary research group from Bristol, together with the Animal Demography Unit at the University of Cape Town (South Africa), has introduced a biometric identification system to help monitoring these species [7] [9][25]. The system can identify individuals from near frontal habitat images. However, accurate penguin counting is not implemented yet.

As the current counting methods are very primitive, we need to use new techniques for counting penguins. This thesis presents a vision system that allows for real-time head counting of African penguins in wildlife video. More specifically, this system should be capable of detecting penguin heads in a bird-view video stream and tracking them from frame to frame. By the combination of detection and tracking algorithms [25], the system counts the number of penguins passing the field of view.

This project will make use of both, appearance based detection and motion based tracking. As in the domain of human surveillance, patterns in video not only refer to that in the spatial structure of image intensities but also refer to the temporal patterns that arise due to motion. Thus, counting by tracking within the field of view can make use of both types of information in order to resolve ambiguities.

The project aims at implementing automated penguin counting without the need for intrusive and time-consuming methods, and achieving high counting accuracy by using techniques that originated in computer vision and biometrics. It can make a small contribution towards the long term goal of integrating the use of computer vision and emerging technologies into the biological sciences.

1.2 Motivation and Context

A recent report shows that the African penguin population has declined rapidly over the last century, falling from several millions in the early 20th century to around 180 000 at the start of the 21st century [5]. At the present time, scientists are trying to find reasons for this phenomenon, in order to reverse this trend. “The marine management of the Western Cape of South Africa produces an annual census and demography information of the endemic African penguin population in order to support conservational policies. The animal demography unit (ADU) of the University of Cape Town uses this information to study the species [14] ”.

1.2.1 Current counting methods

According to Nic Huin's Falkland Islands penguin census 2005/06 report [6], there are three commonly used methods for penguin counting in their colonies.

The first one is called 'Direct Method' [6]. Observers manually count nests with birds or eggs using tally counters. For getting a better accuracy rate, each colony is usually counted by two or three observers. However, this method is very time consuming and it is only suitable for small sized colonies. The second one is 'Photographic Method' [6]. After thousands of digital high resolution color bird-view photographs were taken, a Bezier tool in CorelDraw8 was used to highlight the nests on the screen. Observers usually have to count the nests more than twice to get higher accuracy. However, this method is not suitable for African penguins. Because African penguins' nests are usually built in the bush, it is very difficult to see from a bird-view photograph. The third one is 'Area and Density Method' [6]. This method is suit for large colonies. As the name suggests, after measured the area of a colony, researchers estimated the density of nests in that colony. Then, by multiplying the area with the density, they get an approximate number of nests.

All the above three methods require considerable time and effort but they are still not accurate enough. Moreover, some counting methods are very intrusive which are possibly harmful to the birds. Consequently, there is a need to find out a non-invasive and more accurate method to make the counting for census purposes more efficient.

1.2.2 Previous work on relating subjects

One major contribution for non-invasive observation of wild animals is the Bristol Penguin Recognition Project [7][9][25]. Their system is capable of autonomously monitoring, recognizing and identifying individual African penguins in their natural environment without actually tagging them (See Figure1).

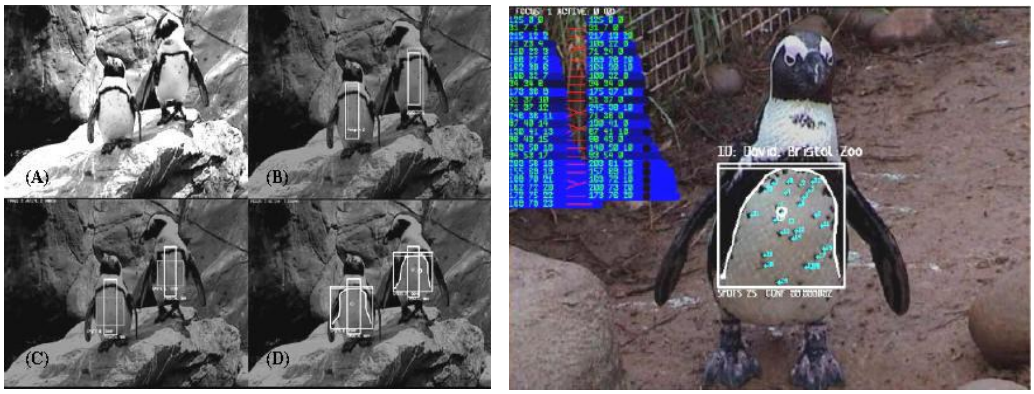


Figure 1: Individual identification of African penguins. The pictures are from the paper 'Automated Visual Recognition of Individual African Penguins (2004) [7][9] '

Before the development of this technology, penguins had to be captured and fitted with metal flipper bands or transponders for identification. It is suspected that both of them have negative effects on welfare and performance. For example, the performance of transponders is

restricted by the distance between reader and penguin. Moreover, transponders, being fitted internally are possibly harmful to the penguins while metal flipper bands will affect the normal activities of penguins [9].

The researchers of Bristol Penguin Recognition Project [7] [9] noticed that the black spots pattern on every adult penguin's chest is unique, which can be used as s natural marking for individual identification. By using computer vision techniques, they developed a real-time monitoring system in 2004. The system is capable of finding and tracking the frontal African penguin's chest in video stream and identifying the penguin by using the unique chest pattern. The whole process mainly consists of three parts [7] [9].

1) Detect and locate the penguin chest in the video stream.
2) Extract the outline of the chest and the pattern of black spots on the chest.
3) Compare the spots pattern a with the database records to achieve identification.

However, this penguin recognition system cannot achieve penguin counting. The reason is obvious. For capturing the whole chest of penguins, camera should be placed in front of the penguins. Looked from that direction, penguins might occlude each other which will make the counting task even more difficult.

2. Aims and objectives

The aim of this project is to implement automated real-time penguin counting in wildlife video and to achieve high counting accuracy. By using techniques that originated in computer vision and human biometrics, this project could help to make the counting for census purposes more efficient.

The specific project objective is to develop an automatic set of computer vision models and algorithms that is capable of processing video of African penguins walking by and detecting, tracking and counting the penguin heads. More specifically, the finished classification system should be capable of:

1) Importing/processing a bird-view video stream of African penguins walking by in their natural habitats (filming at Bristol Zoo Gardens).
2) Detects the penguin heads in the video by using computer vision techniques.
3) Tracks the detected penguin heads until they disappear from view.
4) Count the number of penguins passing the field of view by using a combination of detection and tracking algorithms [25].
5) Output a video, in which the detected penguin head is marked and the number of penguins is displayed.

3. Theoretical Basis

3.1 Object detection

The background research for object detection covers three parts. The first is the feature representation used to characterize the penguin head's appearance. The second is a machine learning algorithm based on AdaBoost [4] [1]. It selects critical visual features to train efficient classifiers. The third is a cascading structure [1] used to combine the boosted classifiers. A cascade of classifiers can greatly improve the detection speed. The high detection speed enables it applied to real-time applications.

3.1.1 Feature representation

3.1.1.1 Requirements

When we detect an object in an image or video, we need to rely on its features which can help to distinguish the object from background. There are many feature representations in image processing, such as Haar-like features, Color features, Gaussian features [11], Gabor features [12] etc. Each of them has advantages and disadvantages. The aim of this project is to detect penguin heads in video stream. Therefore, the feature representation we choose must have the following properties.

1) For achieving real-time detection, the features should be computed extremely rapidly.
2) For detecting penguin heads, the features should be capable of modeling complex objects.
3) The penguins are black and white. Therefore, the features should be capable of modeling edges and regions of high-contrast.
4) The features should be tolerant to illumination changes [10].

Based on the above requirements, Haar-like features are the best choice for this project. The new image representation – integral image, allows the Haar-like features to be computed very fast [1]. Haar-like features has various types and they can vary with different positions and scales. Thus, they are capable of modeling complex objects. Moreover, Haar-like features are good at modeling high-contrast regions. The following section will describe this feature representation in detail.

3.1.1.2 Haar-like feature

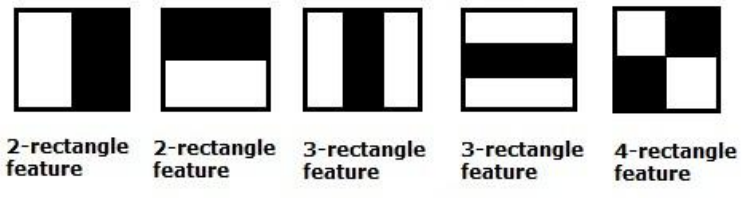


Figure 2: Some rectangle features

The name of Haar-like features comes from Haar wavelets [13] because of their intuitive similarity. They were first used by Papageorgiou et al. to replace using image intensities for face detection [2]. Later, in the study of real-time object detection, Viola and Jones expanded the features on this basis [1]. They used three kinds of features, namely two-rectangle feature, three-rectangle feature and four-rectangle feature (See Figure 2). The value of the rectangular feature is the difference between the sum of the pixels within white rectangle(s) and the sum of the pixels within black rectangle(s). By changing the position and scale of the feature template, the exhaustive set of rectangle features is very large.

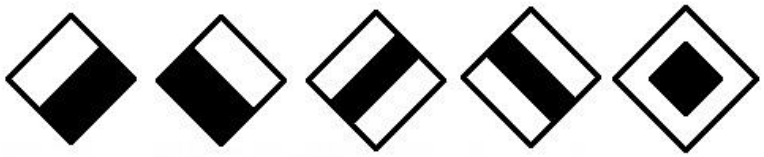


Figure 3: Some tilted rectangle features

One year later, Lienhart and Maydt published a paper called "An Extended Set of Haar-like Features for Rapid Object Detection" [3]. In this paper, they introduced a new set of 45° rotated Haar-like features (tilted Haar-like features) (See Figure 3). The new set of features can improve the detection performance by providing a better description of the characteristics of the object [3]. Use the words of Lienhart and Maydt, "Their basic and over-complete set of Haar-like feature is extended by an efficient set of 45° rotated features, which add additional domain-knowledge to the learning framework and which is otherwise hard to learn" [3].

3.1.1.3 Integral image

Because the training samples are usually over ten thousand and the number of rectangular features is very large as well. There is a need to find a more efficient way to compute the value of rectangular feature instead of calculating every pixel within the rectangle(s).

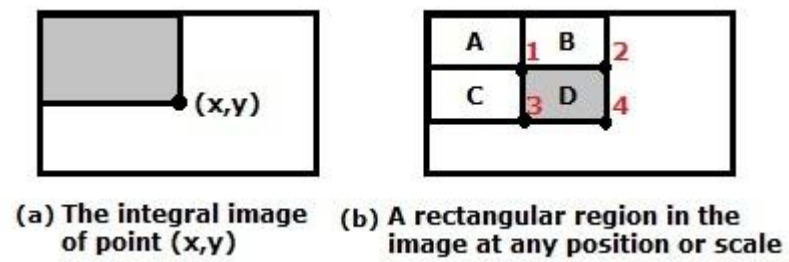


Figure 4: Integral image

Viola and Jones introduced a new image representation method - integral image (Summed Area Table) in the paper "Robust Real-time Object Detection" [1]. The value of the integral image at point (x, y) is the sum of all pixels located on the up-left region (ranging from point (0, 0) to point (x, y)) (See Figure 4 (a)). ii represents the integral image while $i(x, y)$ represents the original image. Then, $ii(x, y)$ can be calculated by [1]:

$$ii(x, y) = \sum_{x' \le x, y' \le y} i(x', y')$$

The value of any rectangle feature can be calculated by only using the integral image of each endpoint of this rectangle. For example, the sum of all pixels within rectangle D can be computed by calculating four integral images of point 1,2,3,4 respectively (See Figure 4(b)).

$$sum(D) = ii(4) - ii(2) - ii(3) + ii(1)$$

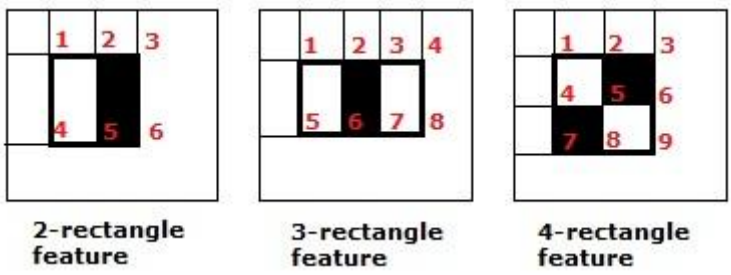


Figure 5: Calculate rectangle features using integral image

Recall the simple Haar-features shown in Figure 2, a two-rectangle feature can be computed by using 6 integral images. A three-rectangle feature can be computed by using 8 integral images and a four-rectangle feature can be calculated by using 9 integral images (See Figure 5). Therefore, regardless of position and scale, any one of these Haar-like features can be computed very fast and in constant time [1].

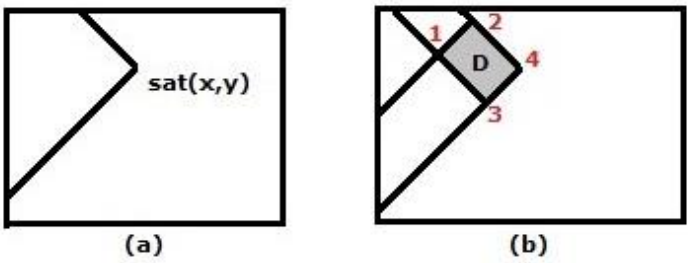


Figure 6: Summed area table

When Lienhart and Maydt proposed a set of tilted Haar-like features, they also used the summed area table representation to achieve high computation speed [3]. $sat(x, y)$ represents the modified integral image and $i(x, y)$ represents the original image (See Figure 6 (a)). $sat(x, y)$ can be calculated by [8] :

$$sat(x, y) = \sum_{\substack{x' \le x \\ x \le x - |y - y'|}} i(x', y')$$

Then, the sum of all pixels within region D (see Figure 6 (b)) can be computed by

$$sum(D) = sat(4) - sat(2) - sat(3) + sat(1)$$

Therefore, like Viola and Jones' basic set of Haar-like features, Lienhart and Maydt's tilted Haar-like features can also be computed very fast and in constant time [3].

3.1.1.4 Discussion

Although Haar-like features can vary with different positions, scales, and patterns, they still have limitation. They are very good at modeling the objects in regular rectangular shape. However, for modeling a complex object, several overlapping Haar-like features are needed to achieve a better modeling performance. Obviously, the more features used, the more computation time needed. There are many feature representations in image processing. A useful one for this project is the color feature. The black and white penguin is colorless while the background is colored. Using color features can decrease the number of false positives.

3.1.2 Supervised learning method

3.1.2.1 Requirements

As mentioned before, we use Haar-like features to characterize the target object's appearance. However, if we only use one feature a time to detect the object, the result is not satisfactory. Therefore, a better method is to use a combination of several features together for the detection. Moreover, as we know, by changing the position and scale of the feature template, the exhaustive set of rectangle features is very large. However, only very few of them are key features which can be used to distinguish the target object. We want to select these key features and combine them together to yield an effective classifier. Therefore, we need to find an algorithm which can solve these two issues by learning from a large set of sample images. Based on the above requirements, AdaBoost learning algorithm is the best choice for this project. The following section will describe the learning algorithm in detail.

3.1.2.2 AdaBoost algorithm

AdaBoost (short for adaptive boosting), as a machine learning algorithm, was first formulated by Freund and Schapire in 1995 [4], and then, applied to human face detection by Viola and Jones in 2001 [1]. Viola et al. proposed a variant of AdaBoost algorithm in the study of real-time object detection to train an extremely effective classifier. The following flow chart (See Figure 7) describes the training process.

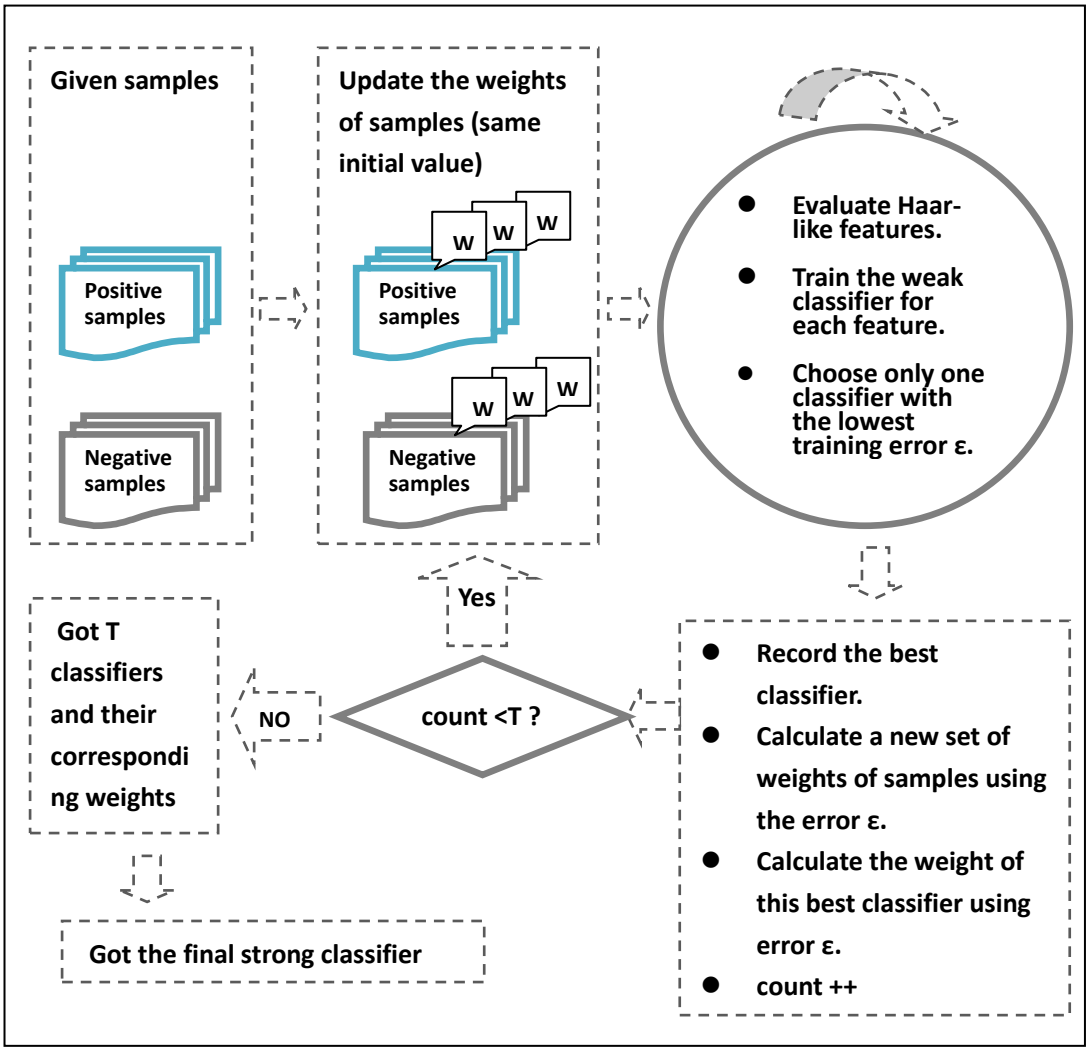


Figure 7: The flow chart for the training process

The training process is an iterative process. There are three key points in this process.

Firstly, Viola et al. made an analogy between the weak classifier and the feature. Then, they restricted each weak classifier to using only one feature. Therefore, the process of finding the best weak classifier is actually the process of finding the key feature.

Secondly, in each round of training, the AdaBoost learning algorithm only chooses the best weak classifier with the lowest error, which means it discards the vast majority of features. The best weak classifier only contains one feature which can best separate the positive and negative samples. For example with frontal penguin head (See Figure 8). Obviously, if we use feature 1 to separate the positive and negative samples, the number of misclassified samples will be very large, which means the training error is high. In contrast, if we use feature 2, the number of misclassified samples will be much smaller, which means the training error is low. Therefore, finding the classifier with the lowest error is actually finding the best weak classifier and the key feature.

Thirdly, the AdaBoost learning algorithm trains different weak classifiers using different training sets obtained by adjusting the corresponding weight of every sample. And then, it

combines these weak classifiers (trained by different training sets) together to form a strong classifier.

More specifically, the corresponding weight of each sample indicates its importance in the training set. Initially, the weight of every sample is the same. Using this distribution, the AdaBoost algorithm trains the first classifier. Then, it updates the weight of every sample by increasing the weight of incorrectly classified sample and decreasing the weight of correctly classified sample. As a result, the incorrectly classified samples are exposed, and so, the next classifier can focus on those samples. Then, the AdaBoost algorithm calculates the weight of the first classifier according to its training error. The weight indicates the importance of this weak classifier. Obviously, the less error it made, the greater weight it got, and vice versa. Under a new sample distribution, the AdaBoost algorithm trains the second classifier. And so on, after T cycles, T classifiers are trained and their corresponding weights are calculated. Then, the final strong classifier is the weighed linear combination of the T weak classifiers.

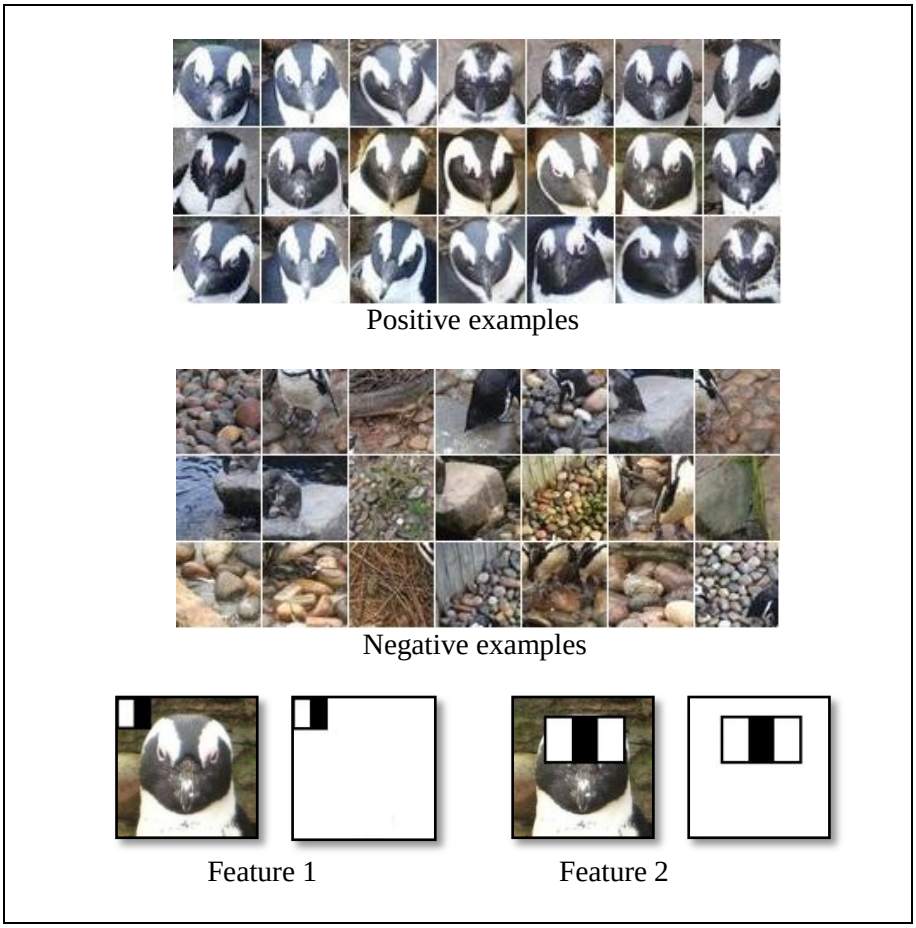


Figure 8: Example with frontal penguin head

The complete AdaBoost leaning algorithm according to Viola and Jones [1] is as follows.

Firstly, a set of samples $(x_1, y_1), \ldots\ldots (x_n, y_n)$ should be provided. $y_i = 0$ for negative samples. $y_i = 1$ for positive samples.

Secondly, initial weights are assigned to all samples.

$w_{1,i} = \frac{1}{2m}$ for $y_i = 0$ (negative), where m is the number of negative samples.

$w_{1,i} = \frac{1}{2l}$ for $y_i = 1$ (positive), where l is the number of positives samples.

Then, loop starts. For t = 1. . . . .T: `

- Normalize the weights to get a probability distribution w_t.

$$w_{t,i} = \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}$$

- Train a weak classifier h_j for each feature f_j.

$$h_j(x) = \begin{cases} 1 & if\ p_j f_j(x) < p_j \theta_j \\ 0 & otherwise \end{cases}$$

where θ_j is a threshold and p_j is a parity indicating the direction of the inequality sign.

- Evaluate the error ε_j of classifier h_j

$$\varepsilon_j = \sum_i w_i \left| h_j(x_i) - y_i \right|$$

- Choose the classifier h_t with the lowest error ε_t
- Update the weights of samples

$$w_{t+1,i} = w_{t,i} \left(\frac{\varepsilon_t}{1-\varepsilon_t}\right)^{1-e_i}$$

where $e_i = 0$ if sample x_i is classified correctly, $e_i = 1$ if sample x_i is classified incorrectly.

Finally, we got a strong classifier, a weighed linear combination of these T weak classifiers.

$$h(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & otherwise \end{cases}$$

where $\alpha_t = log \frac{1-\varepsilon_t}{\varepsilon_t}$

3.1.2.3 Discussion

Although AdaBoost learning algorithm can greatly improve the classification accuracy, it also has disadvantages. There are three main disadvantages:

1) The good performance of AdaBoost relies on a large quantity of good samples. If the training samples are insufficient, the performance might be poor.

2) Recall the training process (see Figure 7), for choosing the classifier with the lowest error, every feature has to be computed in each round of training. Hence, the training process will take a long time.

3) It is susceptible to noise [15].

3.1.3 Cascading

3.1.3.1 Requirements

Although the fast feature computation by using integral image and the efficient learning algorithm by using AdaBoost can greatly reduce the time for object detection. It is not fast enough for a real-time application. To further enhance the detection speed without losing high detection rate, Viola and Jones [1] introduced an algorithm for building a cascade of classifiers. The original meaning of the word "cascade" is a series of small waterfalls. As the name suggests, a cascade of classifiers is composed of a succession of boosted classifiers. It aims to use fewer features to reject more negative sub-windows at the earliest stage, thereby dramatically reducing computation time.

3.1.3.2 Cascade structure

The classifiers at earlier stages are relatively simpler than those at later stages and the former uses more prominent features for classification than the latter. Thus, the classifiers at earlier stages can quickly distinguish those regions which have large differences with the target object. It is very easy to understand. Recall the Adaboost training process described in the last section, in each round of training, AdaBoost only chooses the best weak classifier that contains a single feature which can best separate the positive and negative samples. And as the cycle continues, after reweighted the set of positive and negative samples, the subsequent classifiers focus more on those misclassified samples. That means, after those prominent features have been found by the previous classifiers, the subsequent classifiers face more tough tasks than the previous ones. Therefore, they have to find more inconspicuous features to separate the positives and negatives correctly.

During the object detection, every sub-window goes through a series of classifiers (a cascade of classifiers) in turn. Once a sub-window is rejected by the classifier of a certain stage, it will not enter later stages (see Figure 9). That means the classifiers at later stages only deal with the sub-windows which have passed through all the previous stages.

To give a visualized example (see Figure 10), a classifier can be likened to a sieve. Positive sub-window can be likened to water while negative sub-window can be likened to sand. After each round of screening, sand lefts on the sieve while water goes through it.

In an image, object-like regions are in the minority while background regions are in the majority [1]. Therefore, the classifier at the earliest stage can exclude a large number of background regions, which means only the promising sub-windows can reach subsequent layers. Then, the more complex classifiers at later stages are reserved for these sub-windows, and they use more inconspicuous features to exclude the negative sub-windows. Therefore, the cascade structure can dramatically reduce the computation time.

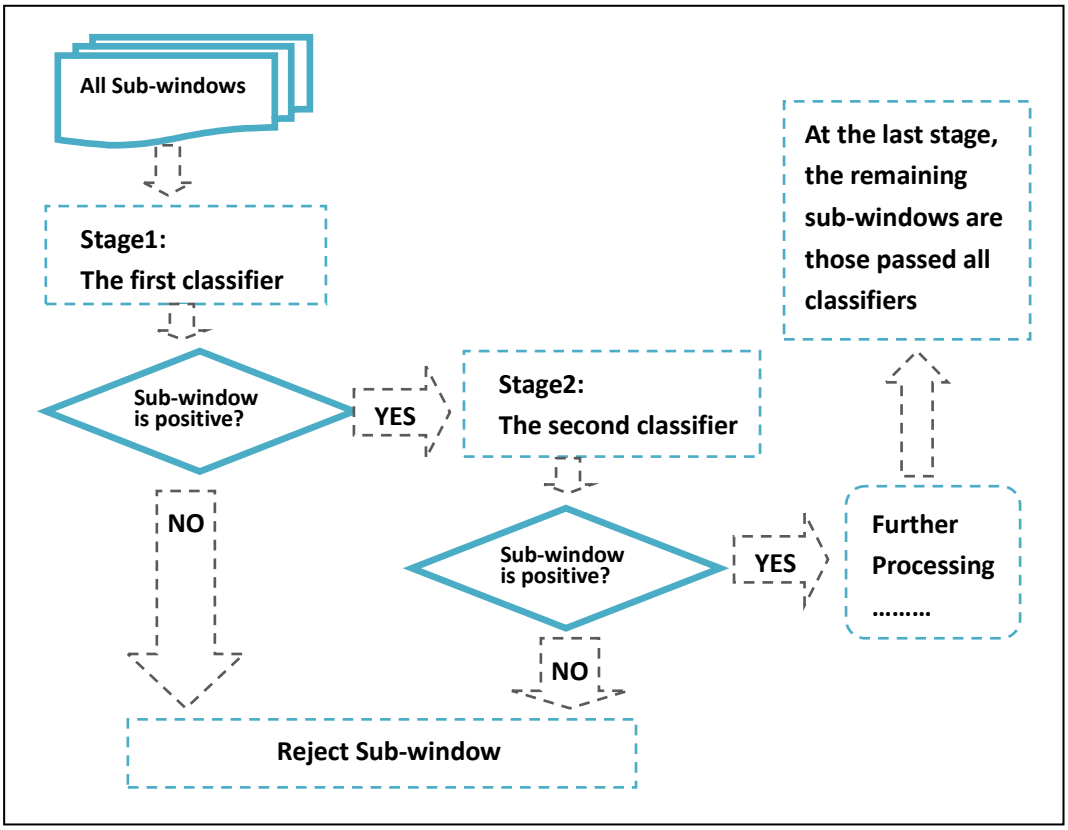


Figure 9: The structure of the cascade according to Viola and Jones [1]

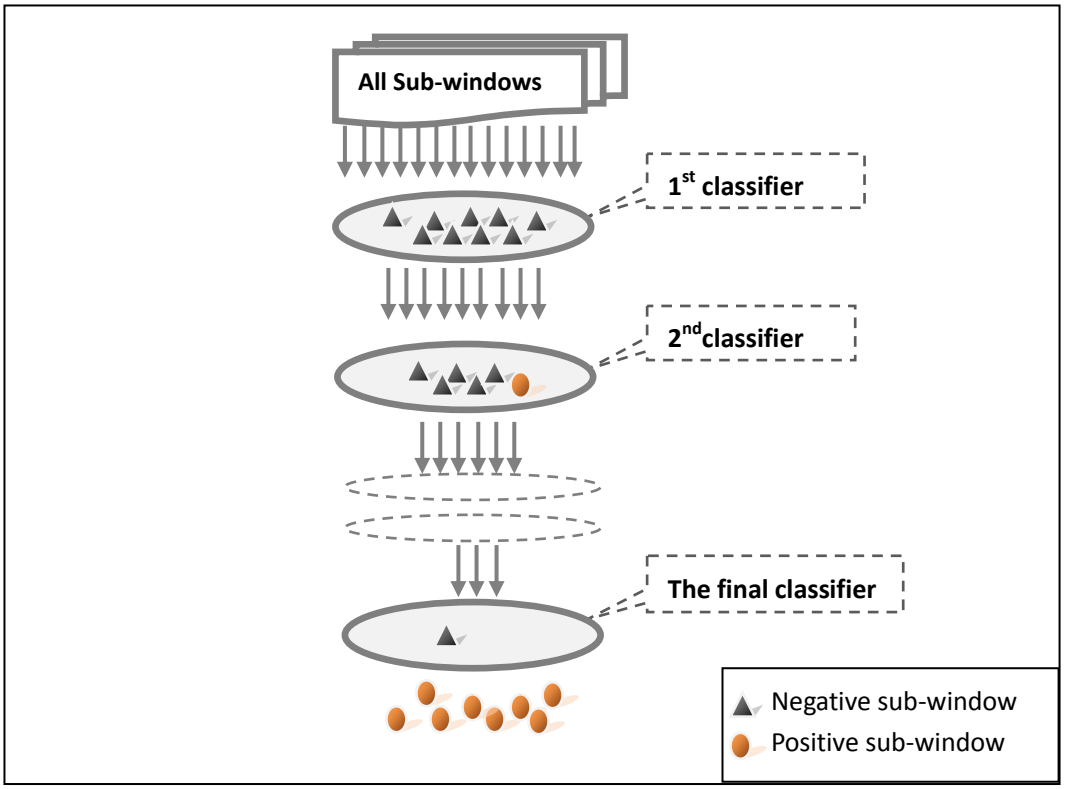


Figure 10: The schematic which vividly describes the cascading process

3.1.3.3 Training a cascade of classifiers

After analyzed the cascade structure, this section will focus on how to train an effective cascade of classifiers. There are two terms need to explain. The first one is "False Positive Rate", which is also known as false acceptance rate (FAR). False Positive, as the name suggests, means incorrectly mark a negative result as a positive result.

$$False\ Positive\ Rate\ = \frac{number\ of\ false\ positives}{number\ of\ negatives}$$

The second one is "True Positive Rate", which is also known as genuine acceptance rate (GAR) or detection rate.

$$True\ Positive\ Rate = \frac{number\ of\ true\ positives}{number\ of\ positives}$$

Training an effective cascade of classifiers is actually finding the optimum with fewer false positives and a higher detection rate (GAR) while maintaining a high detection speed. There is a tradeoff between FAR and GAR. Recall that there is a threshold when training the weak learner. By increasing the threshold, the false positives decrease while the detection rate decreases as well. By decreasing the threshold, the detection rate grows while the false positives also increase. There is another tradeoff between the detection performance and the computation time. Obviously, using more features for classification will get better detection performance (higher GAR and lower FAR) but more computation time will be needed. Using fewer features for classification will save computation time but will get low detection accuracy [1].

In Viola and Jones cascade training algorithm, the user must select the minimum GAR and maximum FAR per layer and the target FAR. The training algorithm according to Viola and Jones is as follows [1].

Before the training starts, the user has to input three values. The first one is the maximum acceptable FAR per layer (f_i). The second one is the minimum acceptable GAR per layer (d_i). The third one is the target overall false positive rate (F_{target}) (See Figure 11). Besides, a set of positive samples and a set of negative samples should be provided.

```
E:\PENGUIN\frontaltraining>haartraining -data e:\penguin\froantaltraining\haarca
scade -vec frontal.vec -bg frontal_neg.txt -nstages 20 -nsplits 2 -minhitrate 0.
999 -maxfalsealarm 0.5 -npos 2282 -nneg 4380 -w 24 -h 24 -nonsym -mem 512 -mode
ALL
```

Figure 11: OpenCV Haar Training

Training starts. Initialize $F_0 = 1.0\ ; D_0 = 1.0\ ; i = 0$

$$F = \prod_{i=1}^{k} f_i$$

where k is the number of classifiers, f_i is the false positive at the ith layer

$$D = \prod_{i=1}^{k} d_i$$

where k is the number of classifiers, d_i is the detection rate at the ith layer

Then, a loop starts. While $F_i > F_{target}$: (if the F_{target} is not met yet, it adds another layer to the cascade)

- $i \leftarrow i + 1$
- $n_i = 0;\ F_i = F_{i-1}$
- While $F_i > f \times F_{i-1}$
  - ✧ $n_i \leftarrow n_i + 1$
  - ✧ Use set of positive and negative samples to train a classifier with n_i features using AdaBoost.
  - ✧ Evaluate current cascaded classifier on validation set to determine $F_i\ and\ D_i$
  - ✧ Decrease threshold for the i th classifier until the current cascaded classifier has a detection rate of at least $d \times D_{i-1}$
- $N \leftarrow \emptyset$
- If $F_i > F_{target}$ then evaluate the current cascaded detector on the set of non-face images and put any false detections into the set N

3.1.3.4 Discussion

Although the cascade structure can greatly improve the detection speed, it has some drawbacks [16].

Firstly, recall that during the detection process, once a sub-window is rejected by one classifier, it will not enter later stages. That means, when lots of negative sub-windows are excluded, some positive sub-windows might also be rejected just because they do not meet the criteria of one classifier. That will increase the False Negative Rate.

Secondly, recall that there are two tradeoffs during the training process. One is between FAR and GAR while the other one is between detection performance and computation time. The users have to try different input values for the minimum GAR and maximum FAR per layer and value for the target FAR to find the optimization result. The retraining process will be very time-consuming.

3.2 Object tracking

The background research for object tracking mainly covers two challenges.

The first one is how to choose features that are best suited for tracking. In this project, we used Jianbo Shi and Carlo Tomasi‘s corner detection [17]. The second challenge is how to track the selected features from frame to frame. We used the Kanade–Lucas–Tomasi (KLT) Feature Tracker [22] for this project in order to track the corner points selected by Shi and Tomasi [17].

3.2.1 Find Good features to track

3.2.1.1 Requirements

If we want to track an object in a video stream, we need to get enough motion information on that object. However, motion information cannot be got from every part of an image. For instance, the motion along an edge might be ambiguous but the motion along the direction that perpendicular to the edge can be seen clearly (See Figure 12). For this reason, we have to select some "good features" which contain enough motion information for supporting us to track an object.

Good features can be selected based on texturedness or cornerness by different measures [17]. For example, one measure, proposed by Moravec in 1980, is to use high standard deviations in spatial intensity profile [21]. However, these measures sometimes are arbitrary. Because they assumed the selected features are "good features", but in fact the features might not be good enough for a tracking algorithm [19]. In 1994, Jianbo Shi and Carlo Tomasi proposed a new approach - corner detection, for selecting good features that can be tracked well. They based the definition of good features on their way for tracking rather than an arbitrary assumption [17] .

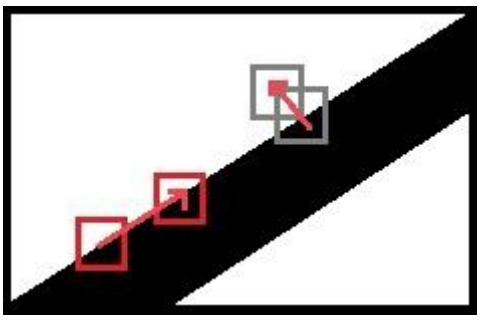


Figure 12: Edge

3.2.1.2 Corner detection

A corner can be defined as where two edges meet or a point that has high intensity changes in both two directions (horizontal and vertical) (See Figure 13).

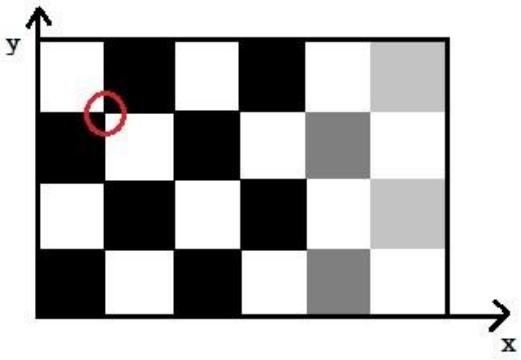


Figure 13: Corner

Harris corner detector [18] and Shi and Tomasi corner detector are the two most commonly used corner detectors and these two corner detectors are very similar. The Harris corner detector uses a function to calculate two eigenvalues and gets a score from that. Then, it uses the score to check if the window can be marked as a corner. Jianbo Shi and Carlo Tomasi eliminated the process of using function to get the score and only used the two eigenvalues to determine whether a window is a corner or not [20].

As mentioned before, a corner point has high intensity changes in both x and y directions (horizontal and vertical). For each pixel (x, y), both Harris corner detector and Shi-Tomais corner detector calculate the intensity variation matrix Z (a symmetric and positive semi-definite 2 x 2 coefficient matrix).

$$Z = \begin{bmatrix} \left(\frac{\partial I}{\partial x}\right)^2 & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \left(\frac{\partial I}{\partial y}\right)^2 \end{bmatrix}$$

The eigenvalues of Z are denoted by $\lambda 1$ and $\lambda 2$. Different eigenvalues can represent different features (See Figure14). A good example can be seen from Figure 15.

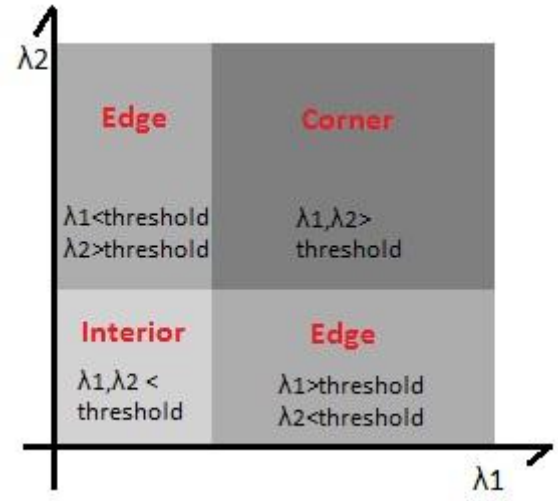


Figure 14: Different eigenvalues for different feature regions

The partial enlarged screenshot on the top left corner shows an edge feature which has large $\lambda 1$ but small $\lambda 2$.

The partial enlarged screenshot on the bottom right corner shows a low texture region which has small $\lambda 1$ and $\lambda 2$.

The partial enlarged screenshot on the top right corner shows a high textured region- a corner feature which has large $\lambda 1$ and $\lambda 2$.

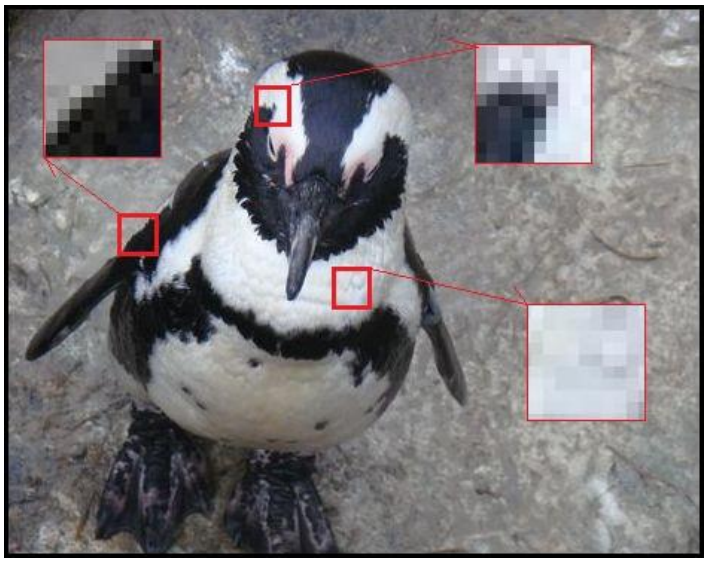


Figure 15: Different feature regions on a penguin

The Harris corner detector calculates the score by using the function like this:

$$score = det\ (Z) - k(trace(Z))^2$$

$$det(Z) = \lambda 1\ \lambda 2$$

$$trace(Z) = \lambda 1 + \lambda 2$$

As we know, if we want to track a window, the matrix Z must be well conditioned and above the image noise level as well [17] . Features with high eigenvalues can be above the image noise level and those with similar eigenvaluse can be well conditioned [17] [19]. Therefore, *λ1* and *λ2* must be both large and in a similar order of magnitude. According to this principle, Shi and Tomasi calculate the score by:

$$score = min\ (\lambda 1, \lambda 2)$$

They accept the window to be a corner if

$$score > threshold$$

In fact, Shi and Tomasi corner detector is mainly based on Harris corner detector but they changed the selection criteria and produced better results in corner detection. OpenCV implements both Harris corner detector and Shi-Tomais corner detector.

3.2.2 Track the good features

3.2.2.1 Requirements

After we selected good features by using the Shi and Tomasi corner detector, we want to track these good features from frame to frame. Therefore, the most challenging thing is how to find one frame's features in another frame.

Optical flow is an important method in dynamic images analysis. As we know, an image comes from the 3-D projection. When an object moves, the brightness patterns of the object's corresponding points in the image also moves. The apparent motion of brightness patterns in

the image is optical flow.

The points on the moving object will appear in different locations at different times. Therefore, we can track the moving object by tracking the points on it. Assuming the brightness of the same point on the moving object always be the same, we can get an image constraint equation [26]. For example, the image at time t is called image A and the image at time $t + \delta t$ is called image B. According to this assumption, the brightness intensity of a point at time $t + \delta t$ and at location $(x + \delta x, y + \delta y)$ in image B must equal to the brightness intensity of the same point at time t and at (x, y) in image A [26].

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t)$$

If the movement between time t and time $t + \delta t$ is small enough, we can get a new equation by using Taylor series to this image constraint equation [26].

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + H.O.T$$

If the movement is small enough, $H.O.T$ is nearly 0. Then, by combining the above two equations, we can get [26]:

$$\frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t = 0$$

Or

$$\frac{\partial I}{\partial x}\frac{\delta x}{\delta t} + \frac{\partial I}{\partial y}\frac{\delta y}{\delta t} + \frac{\partial I}{\partial t}\frac{\delta t}{\delta t} = 0$$

V_x, V_y $(\frac{\delta x}{\delta t}, \frac{\delta y}{\delta t})$ are the x and y components of the optical flow of $I(x, y, t)$ respectively.

$$\frac{\partial I}{\partial x}V_x + \frac{\partial I}{\partial y}V_y + \frac{\partial I}{\partial t} = 0$$

I_x, I_y, I_t are the derivatives of the image at point (x, y, t) in x, y, t direction respectively.

$$I_x V_x + I_y V_y + I_t = 0$$
$$I_x V_x + I_y V_y = -I_t$$

Obviously, the above equation cannot be solved because there are two unknowns in it. That means at least another one equation by using more additional constraint is need for calculating the optical flow. Therefore, we need to find an optical flow method which introduces additional conditions. Lucas-Kanade optical flow method is a good choice for solving this problem.

3.2.2.2 Lucas-Kanade optical flow method

KLT tracker is the short for Kanade-Lucas-Tomasi feature tracker which is based on Lucas–Kanade optical flow method. This method, proposed by Bruce D. Lucas and Takeo Kanade in 1981 is a two-frame differential method for optical flow estimation [22] .

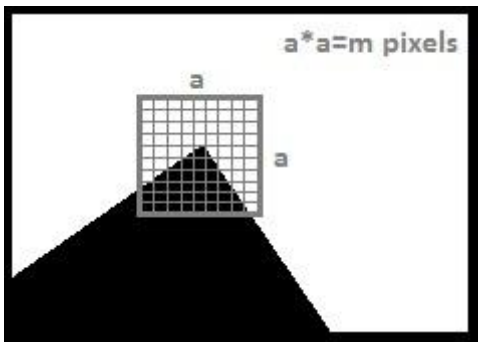


Figure 16: A small patch around the corner feature

If we want to track a corner feature from frame to frame, compared with using one single point to represent the corner, using a small patch around the corner point is more stable. The vertex of the corner is the central pixel of the small patch. If the time between frames is very short and the movement of the object is not obvious, The Lucas–Kanade optical flow method assumes that within a small patch in an image, the optical flows of all pixels are the same (see Figure 16). If there are m pixels within that small patch ($a \times a = m$), we can get [27]:

$$I_{x1}V_x + I_{y1}V_y = -I_{t1}$$
$$I_{x2}V_x + I_{y2}V_y = -I_{t2}$$
$$\cdots\cdots\cdots\cdots$$
$$I_{xm}V_x + I_{ym}V_y = -I_{tm}$$

We can write the equations into a matrix [27]

$$\begin{pmatrix} I_{x1} & I_{y1} \\ I_{x2} & I_{y2} \\ \cdots & \cdots \\ I_{xm} & I_{ym} \end{pmatrix} \begin{pmatrix} V_x \\ V_y \end{pmatrix} = - \begin{pmatrix} I_{t1} \\ I_{t2} \\ \cdots \\ I_{tm} \end{pmatrix}$$

Obviously, the equations are more than unknowns. By using least squares principle, we can get the optical flow[27]:

$$\begin{pmatrix} V_x \\ V_y \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} (I_{xi})^2 & \sum_{i=1}^{m} I_{xi}I_{yi} \\ \sum_{i=1}^{m} I_{xi}I_{yi} & \sum_{i=1}^{m} (I_{yi})^2 \end{pmatrix}^{-1} \begin{pmatrix} -\sum_{i=1}^{m} I_{xi}I_{ti} \\ -\sum_{i=1}^{m} I_{yi}I_{ti} \end{pmatrix}$$

For highlighting the corner points, they use Gaussian function to give more weight to those points which are closer to the central pixel of the patch [27].

$$\begin{pmatrix} V_x \\ V_y \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{m} w_i (I_{xi})^2 & \sum_{i=1}^{m} w_i I_{xi} I_{yi} \\ \sum_{i=1}^{m} w_i I_{xi} I_{yi} & \sum_{i=1}^{m} w_i (I_{yi})^2 \end{pmatrix}^{-1} \begin{pmatrix} -\sum_{i=1}^{m} w_i I_{xi} I_{ti} \\ -\sum_{i=1}^{m} w_i I_{yi} I_{ti} \end{pmatrix}$$

3.2.2.3 Discussion

Described above is the standard Lucas-Kanade optical flow method [22] [27]. Although the performance of this method is good, it still has limitation.

Firstly, recall the new image constraint equation by using Taylor series.

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\delta x + \frac{\partial I}{\partial y}\delta y + \frac{\partial I}{\partial t}\delta t + H.O.T$$

Lucas-Kanade made an assumption that if the movement is small enough, $H.O.T$ is nearly 0. However, this is not exact. For improving accuracy, $H.O.T$ should be taken into account. In implementation, this can be solved by using many iterations of Newton-Raphson method [23]. This extended version is called iterative Lucas-Kanade algorithm.

Secondly, the original Lucas-Kanade optical flow method only works for small motion. If the object moves very fast, which means the optical flow vector between the two nearby instants frames is large, the original Lucas-Kanade optical flow method might fail. Therefore, pyramids were used to solve this problem [23].

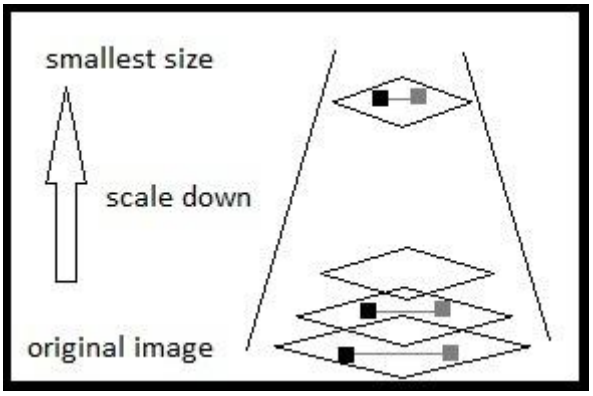


Figure 17: Lucas-Kanade with pyramids

As we can see from Figure 17, with the whole image scales down, the resolution of the image was reduced. Therefore, the displacement of the image contents between two nearby instants frames scales down as well. This extended version is called Pyramidal Lucas-Kanade algorithm (Multi-resolution Lucas-Kanade algorithm) [23].

OpenCV implements the Pyramidal iterative version of the Lucas-Kanade optical flow [24] .

4. The visual system design

Currently, the African Penguin population census is mainly done by manually counting nests. Other methods are difficult to implement, since penguins look almost identical to the naked eye; and they roam freely in the colony. Therefore, we want to use new techniques that originated in computer vision and biometrics to solve this problem.

Firstly, the similarity of penguins' appearance is conducive to allow computer to find their common characteristics and recognize them. African Penguins have broad white "C" shaped markings on their black heads that do not change from season to season during their adult life. We can model this characteristic pattern on the penguin head using machine learning algorithms on large sets of real-world data. Since juvenile penguin has a different look with adult penguin, this system does not apply to juvenile penguins (See Figure 18).

Figure 18: (left) Adult African penguin (right) Juvenile African penguin

Secondly, counting moving objects in a defined field of view is a well constrained vision task. Penguins walk along in predictable patterns between their nest site and the beach at some point in the day. Thus, if the camera has been set up on established walkways, it is fairly certain that most of the population will be captured by the camera.

Figure 19 shows three options of the location of the camera. If the camera is placed at location 1, obviously, the penguins will occlude each other. That would make the head detection task even more difficult. If the camera can be placed at location 3, the occlusion can be avoided. However, in penguins' natural habitats, it is hard to place a camera at that position. Therefore, a compromise location is location 2.

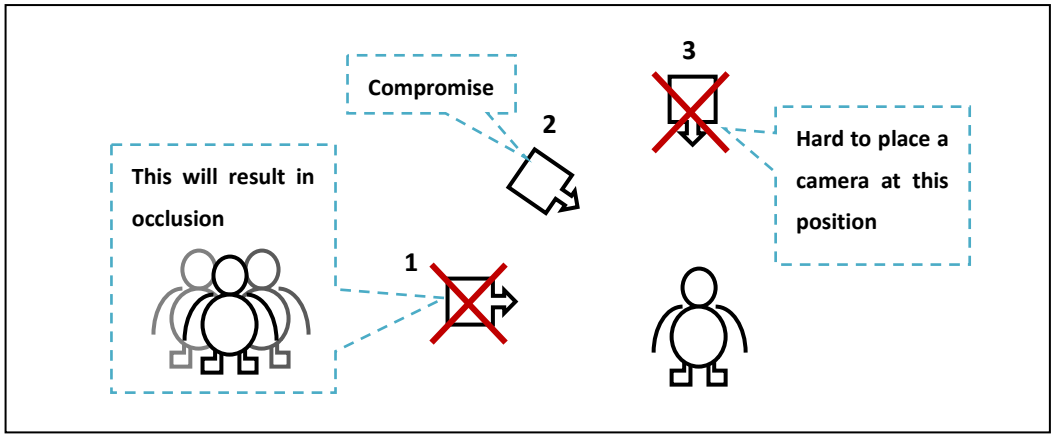


Figure 19: The choices of setting up a camera

In order to detect penguin heads in different poses, five sets of positive samples were collected for training five classifiers (detectors). The five detectors cover the 5 penguin poses (front, front side, side, side back, back) (See Figure 20).

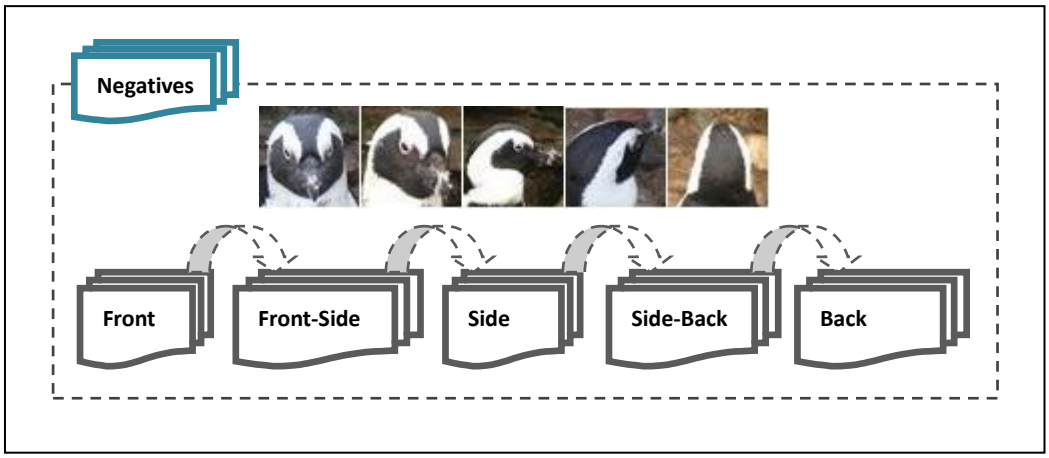


Figure 20: Five poses of the penguin head.

4.1 System running processes

The flow chart shown in Figure 21 describes the main process of the penguin head counting system.

After imported a video stream of African penguins walking by, the system captures the first frame of the video. By using the trained detectors, the system detects penguin heads in the first frame. After integrated multiple detections it saves the final detection results into a set of interest models. The system finds four strong corner points for each interest model and saves them into a set of feature points. After that, system enters a loop.

In each loop, it captures the next frame from the video stream and tracks the feature points of the last frame to get a new set of features points in the current frame. At the same time, the system detects penguin head in the current frame again and saves the detection results into new interest models. By combining the tracked points with new interest models, the system gets new evaluated models and updated feature points. During this period, the counting is

implemented as well and the counting result, updated feature points, new evaluated models will be displayed on the result window in real-time. The loop will continue until it has gone through every frame in the video.

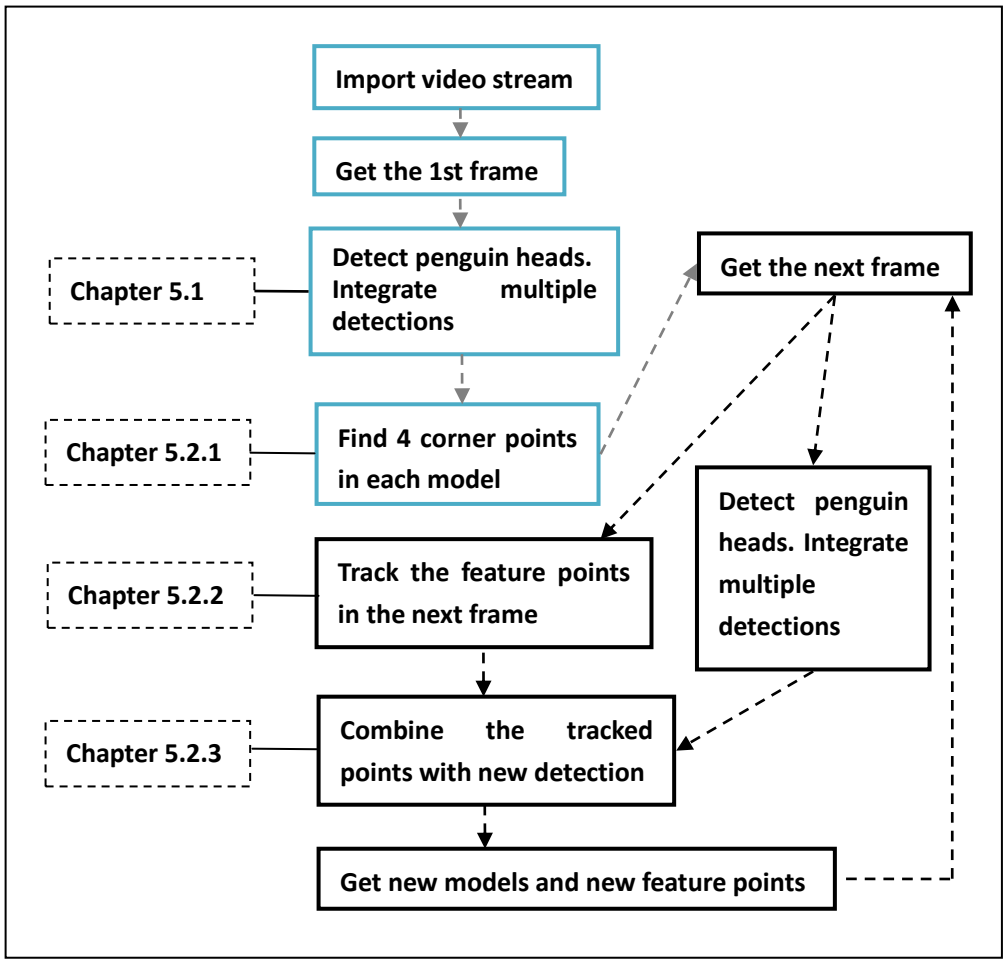


Figure 21: Flow chart for the main process

4.2 Algorithm design

The following pseudo-code describes the main algorithms of this automatic penguin head counting system. The algorithms for specific implementation are omitted here and they will be described in detail in Chapter 5.

```
main()
{
    Load Classifier ()                          /* cascade 1,2,3,4,5*/
    cvCaptureFromAVI()

    frame1 = cvQueryFrame()                     /*capture the 1st frame*/
    detect ( frame1 )
    draw(frame1,recNewArray)
    find_corner(frame1,recNewArray)
    cvCopy( frame1,frame_current,0)
```

```
    for(every  frame)                                   /*capture the next frame*/
    {
        frame = cvQueryFrame(capture)
        track(frame_current, frame, cornerArray)
        moved_model(frame)
        detect(frame)
        new_model(frame)
        draw(frame,recNewArray)
        cvCopy(frame,frame_current, 0 )
    }
}
detect( IplImage* img )
{
    for(every cascade)                                  /* cascade 1,2,3,4,5*/
    {   detect_penguin_head()   }                       /* cvHaarDetectObjects() */
    Integrate_multi-poses()                             /* save into recNewArray[]*/
    Update_model_num ()
}
draw(IplImage* img,CvRect recarray[])
{
    for(every interest model)
    {   if (the model has been counted)
        Draw_red_rectangle()                            /* cvRectangle */
        else
        Draw_white _rectangle()                         /* cvRectangle */
    }
}
find_corner(IplImage* img,CvRect recarray[])
{
    for(every interest model)
    {   Set_ROI()                                       /* cvSetImageROI */
        Find_Features()          /* cvGoodFeaturesToTrack  save into cornerArray[]  */
        Draw_green_circle ()                            /* cvCircle */
    }
}
find_corner_again(IplImage* img,CvRect recarray,int corner_num)
{
    Set_ recarray_as_ROI()                              /*cvSetImageROI */
    Find_Features()                                     /* cvGoodFeaturesToTrack */
                          /*Find corner_num good features, save into corner_new[] */
    Draw_green_circle()                                 /* cvCircle */
}
```

```
track( IplImage* img,IplImage* img2,CvPoint2D32f corarray[] )
{
    Save_features()          /*Save the features before tracking into cur_cornerArray[]*/
    Track _features()        /* cvCalcOpticalFlowPyrLK save into cornerArray[] */
    Draw_red_circle()                                              /* cvCircle */
}
moved_model(IplImage* img2)
{
    Calculate_center()       /*Calculate center of corner points for each model
                                before tracking and after tracking*/
    Calculate_delta()        /*Calculate the displacement between them */
    Calculate_moved_model()  /*moved_model=prev_model+delta*/
}
new_model(IplImage* img2)
{
    For(different case)
    {          /*case1: match detection. case2: new detection. case3: lost detection*/
        Update_new_model()
        Update_feature_points()                          /* find_corner_again()*/
        Update_status()
        Implement counting()
    }
```

5. The visual system implementation

5.1 Penguin head detection

5.1.1 Collect positive samples

In order to have enough samples for haartraining process, I took more than six thousands of positive images which contain penguin heads in different poses and 3366 of negative images which do not contain any penguin head in Bristol Zoo Gardens.

For collecting positive samples, I marked penguin heads in positive images manually and saved them into five categories, namely frontal, front side, side, side back and back (See Figure 22). The output file, which was saved as a text document contains the coordinates of penguin heads in the original image. The format of the file is as follows:

[File name/Image name] [Number of objects] [x y (left-upper corner of the object)] [Width height of the object]
For example: frontpos/DSC01864.jpg 1 505 111 50 50

Figure 22: Five categories for five poses

5.1.2 Collect negative samples

For training an effective classifier, a good set of negative samples is also very important. If the positive sample represents the object that we want to detect, then the negative sample represents the thing that we do not want to detect. The objective of this project is to detect the penguin head in their natural environment. Therefore, for improving the detection performance and achieving low false positive rate, the negative samples should contain both background and penguin's body information.

For testing the detection performances by using different sets of negative samples, two frontal penguin head classifiers (both 20 layers) were trained using two different sets of negative samples (See Figure 23). The first set consists of 3987 50 by 50 pixels negative images which do not contain any penguin while the second set consists of 2943 up to 1000 by 1000 pixels negative images which do not contain any penguin head but contains penguin's body. The comparision results are shown in Figure 24. Figure 25 shows the ROC curves for the two classifers.

Figure 23: Two different sets of negative samples.

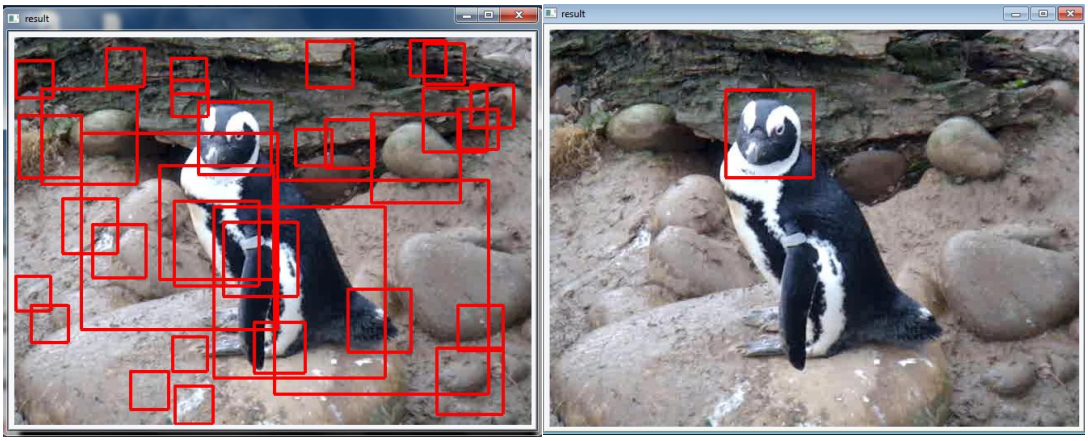


Figure 24: The comparision results

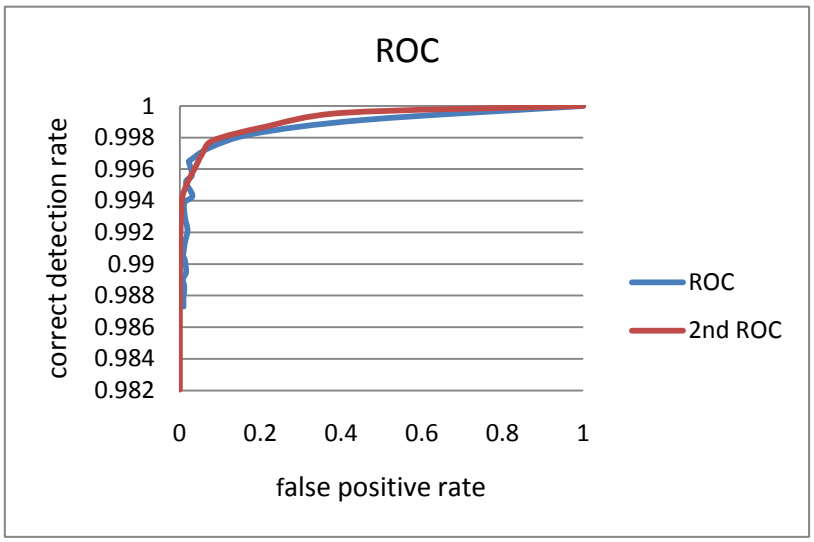


Figure 25: ROC curves comparing the first classifiers to the second one

5.1.3 Model construction/training

The following table shows the number of collected samples used for training.

Table 1: The number of positive and negative samples

	Positive samples	Negative samples
Front	3828	3366
Front Side	4253	3366
Side	5875	3366
Side Back	2812	3366
Back	1801	3366

We created training samples with the createsamples utility (See Figure 26 (top)) and trained our classifiers using the haartraining utility (See Figure 26 (bottom).

```
E:\PENGUIN\training>createsamples -info front_pos.txt -vec front_pos.vec -w 24 -
h 24 -num 3828 -show true_
```

```
E:\PENGUIN\training>haartraining -data haarcascade1 -vec front_pos.vec -bg front
_neg.txt -nstages 17 -nsplits 2 -minhitrate 0.999 -maxfalsealarm 0.5 -npos 3828
-nneg 3366 -w 24 -h 24 -nonsym -mem 512 -mode ALL_
```

Figure 26: (top) createsamples utility (bottom) haartraining utility

The following graphs (See Figure 27 - Figure 31) show the haartraining data for the five classifiers (five poses).

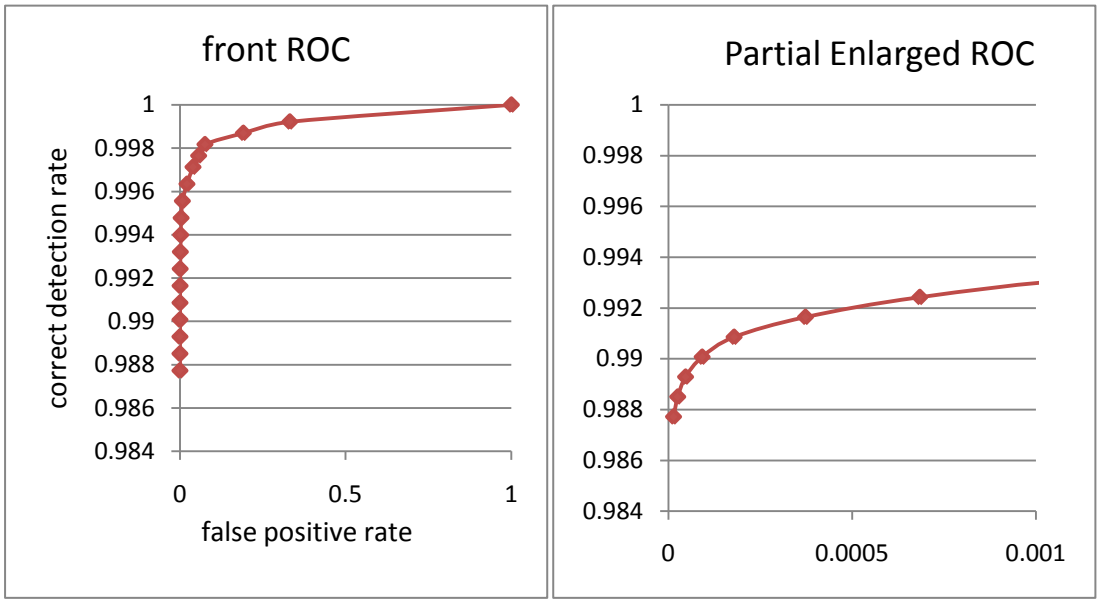


Figure 27: ROC curves for training the front classifier

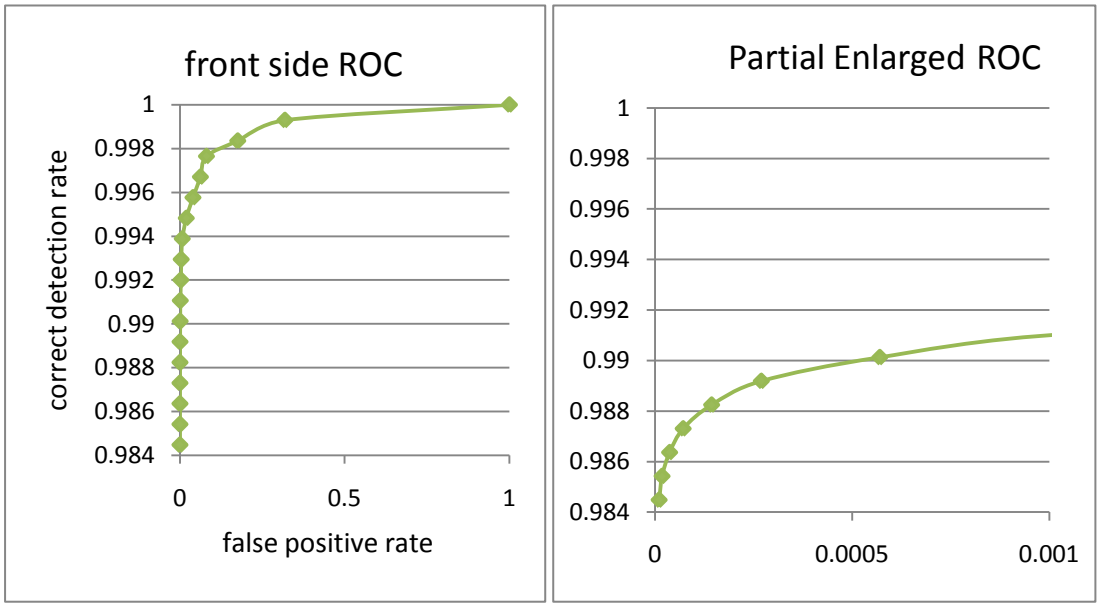


Figure 28: ROC curves for training the front side classifier

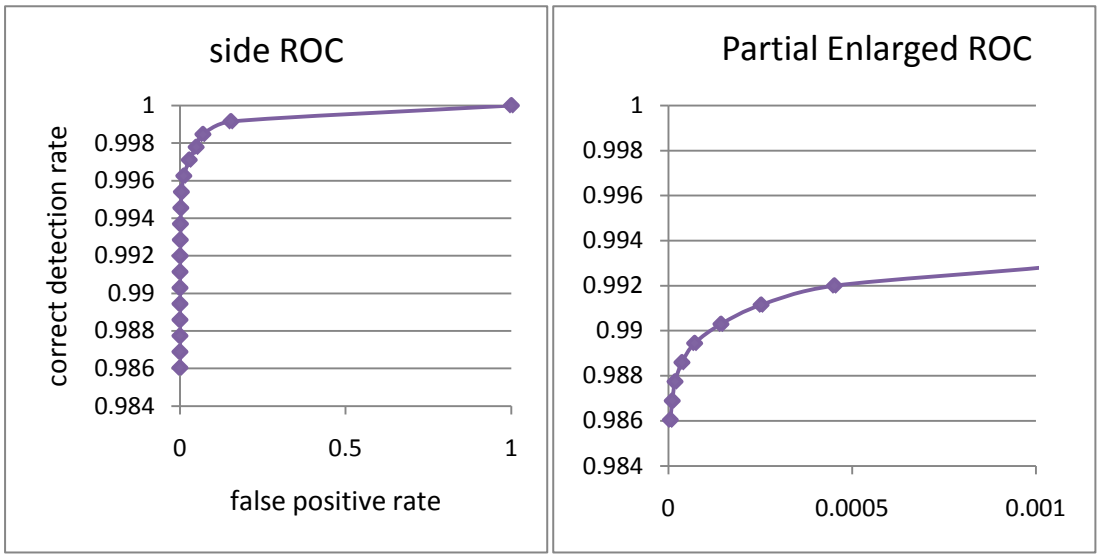


Figure 29: ROC curves for training the side classifier

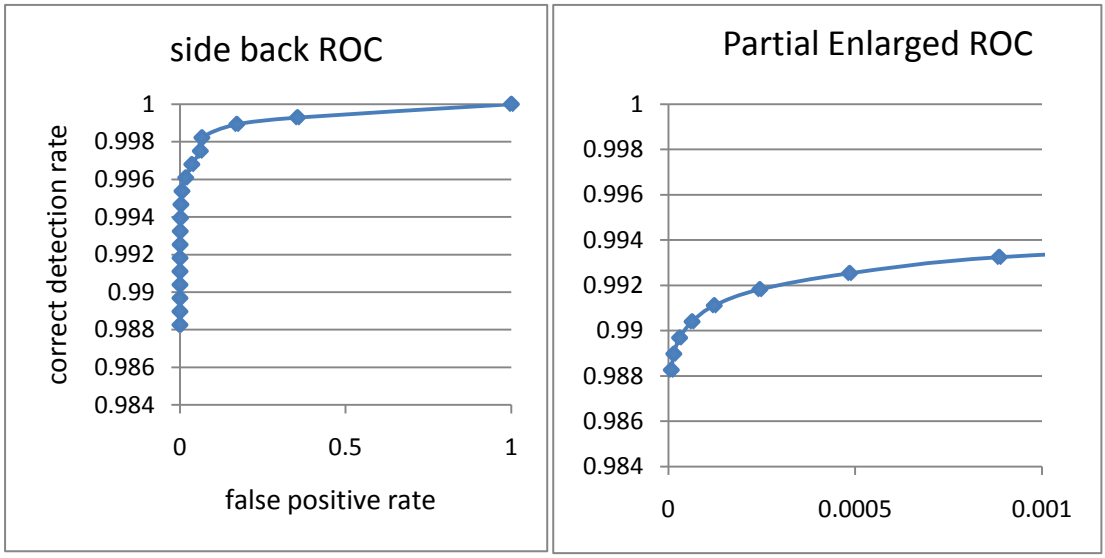


Figure 30: ROC curves for training the side back classifier

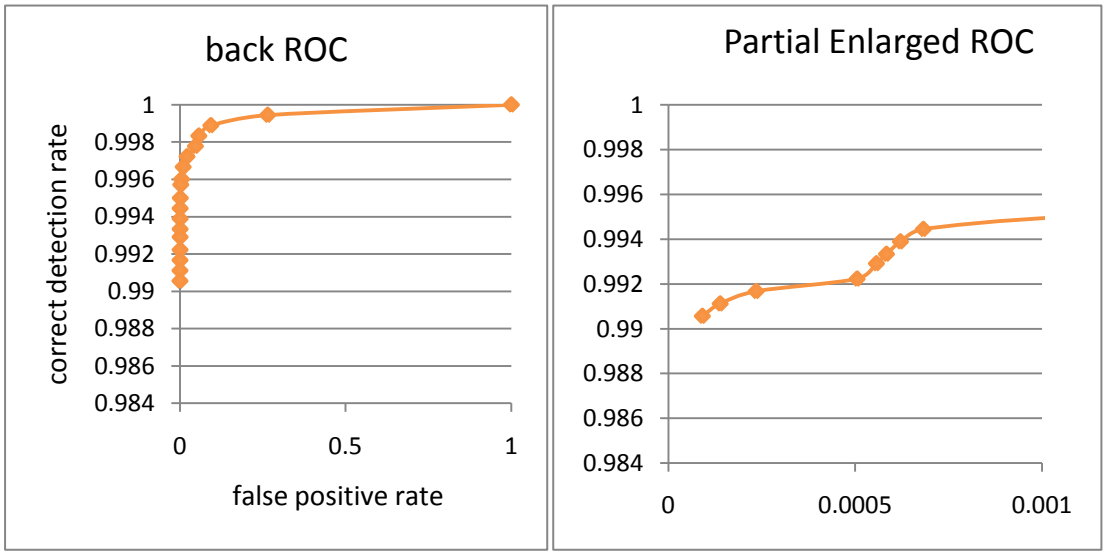


Figure 31: ROC curves for training the back classifier

As we can see from the partial enlarged line graphs, the training ROC curve for back classifier is not as smooth as the other four classifiers. The training data for back classifier is shown in Table 2. From the 9th stage to 13th stage, the false positive rate always stays around 0.0005. That means, from the 9th stage, the AdaBoost learning algorithm cannot find enough "key features" from the back of penguin head to separate the back (positive samples) from the background (negative samples). Therefore, the incorrectly classified negatives are not reduced until the algorithm overlearned the training samples from the 14th stage.

Table 2: Part of training data for back classifier

No. Stage	POS (GAR)	NEG(FAR)
8	0.995003	0.0010869
9	**0.994448**	**0.000682634**
10	**0.993892**	**0.00062134**
11	**0.993337**	**0.0005839**
12	**0.992914**	**0.000556952**
13	**0.992227**	**0.000505247**
14	0.991671	0.000234755

5.1.4 Classifier performance evaluation

For evaluating the classifier performance, I used 497 positive test images and 200 negative test images, which have never been used to train classifiers before. For each pose, I created a ground truth text including information for locations of penguin head in original image. I created 662 positive test samples and 200 negative test samples. I evaluated the performance of the five detectors by using the performance utility (See Figure 32).

```
E:\PENGUIN\test>performance -data haarcascade1.xml -info test1.txt -ni_
```

Figure 32: Performance utility

Table 3 shows the detection rates (GAR) of the five detectors on test set. 'Hits' means the detection matches the hand labeled ground-truth. 'Missed' means the detector did not detect the hand labeled ground-truth. We can get Genuine Acceptance Rate (GAR) by:

$$GAR = \frac{Number\ of\ Hits}{Number\ of hand\ labeled\ ground-truth}$$

Table 3: Detection Rates (GAR) of the five detectors on test set

Pose	Classifier	Hand labeled Ground-Truth	Hits	Missed	Detection Rate (GAR)
Front	haarcascade1	123	112	11	91.06%
Front Side	haarcascade2	138	128	10	92.75%
Side	haarcascade3	206	200	6	97.09%
Side Back	haarcascade4	130	110	20	84.62%
AVG (for the first four classifiers)					**91.38%**
Back	haarcascade5	65	39	26	60%

The performance utility also records the 'false' (false positives), which means the detector detects something which is not the hand labeled ground-truth. However, since the differences between adjacent poses are not significant enough, the functions of detectors for different poses might have lots of intersections. For example, the detector trained for frontal penguin head might also detect a penguin head in front side pose. However, for each detector, the hand labeled ground-truth only contains the penguin head in one pose. Therefore, the 'False positives' value does not represent the real situation.

For testing the False Acceptance Rate (FAR), I collected 200 300 by 300 pixels negative images.

$$Number\ of\ windows = \left(\frac{300 - 23 \times 1.1^{0}}{2}\right)^{2} + \left(\frac{300 - 23 \times 1.1^{1}}{2}\right)^{2} + \cdots \left(\frac{300 - 23 \times 1.1^{8}}{1.1^{8}}\right)^{2} + \left(\frac{300 - 23 \times 1.1^{9}}{1.1^{9}}\right)^{2} + \cdots \left(\frac{300 - 23 \times 1.1^{26}}{1.1^{26}}\right)^{2}$$

By using the above equation, in each image, the number of negative windows that will be checked by the Viola-Jones method (considering scale and shift invariance) is about 203,685. The number of negative windows for 200 negative images is 40,737,000. Table 4 shows the FAR (per window**)** of the five detectors

$$FAR = \frac{Number\ of\ False}{Number\ of\ negatives}$$

Table 4: FAR (per window) of the five detectors

Pose	Classifier Name	Negative windows	False	FAR (per window)
Front	haarcascade1	40,737,000	12	2.9457E-07
Front Side	haarcascade2	40,737,000	7	1.7183E-07
Side	haarcascade3	40,737,000	2	4.9095E-08
Side Back	haarcascade4	40,737,000	6	1.4729E-07
AVG (for the first four classifiers)				**1.6570E-07**
Back	haarcascade5	40,737,000	68	1.6692E-06

As we can see from Table 3 and Table 4, the average GAR of the first four classifiers reaches 91.38% while the average FAR (per window) of the first four classifiers reaches 1.6570E-07. However, the performance of the detector trained for back pose is not good enough. The GAR of this detector only reaches 60% while the FAR (per window) is 1.6692E-06. Therefore, we only use the first four detectors (front, front side, side, side back) and abandon the detector trained for back pose.

Because the number of test negative images (200 300 by 300 pixels) is far smaller than the number of negative images used for training (3366 up to 1000 by 1000 pixels). We cannot compare the test FAR (per window) with the training FAR (per window).

5.1.5 Extend the pose space

By now, four detectors (front, front side, side, side back) have been trained and tested. However, since the front side, side and side back detectors were trained using samples taken from the right side of a penguin (See Figure 22), these three detectors can only detect a penguin head from the right side.

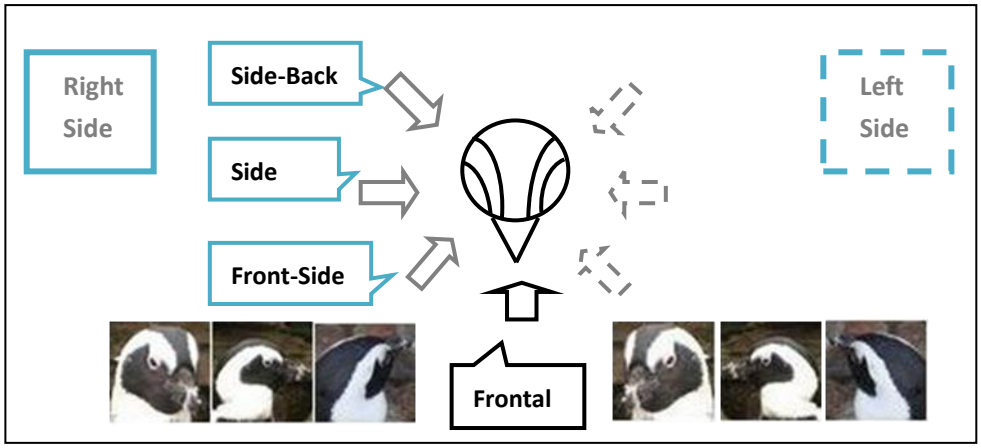


Figure 33: The penguin head pose space (at given camera elevation)

We can extend the pose space to a full 360 degrees coverage (at given camera elevation) exploiting the physiological symmetry of African penguins (See Figure 33). Firstly, we need to flip the original image around y-axis to get a horizontal mirror image. Then, we can use the detectors for right side to detect penguin heads in this horizontal mirror image. We get the penguin head's coordinates (x, y) in the horizontal mirror image (See Figure 34) and we can calculate the new coordinates $(x2, y2)$ in original image by the following function:

$$x2 = img\_w - x - rec\_w$$

$$y2 = y$$

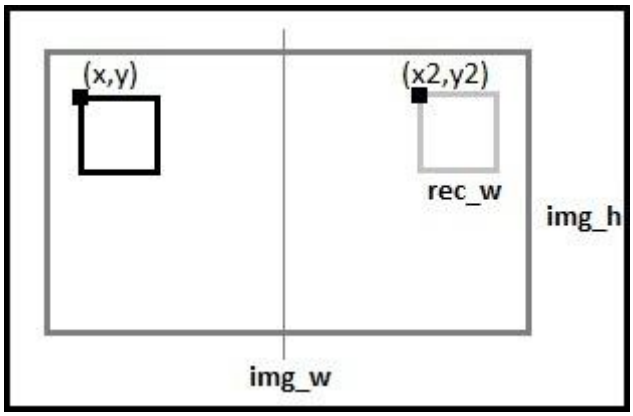


Figure 34: Flip the image

5.1.6 Integration of Multiple Detections

Currently, the detectors covering 7 poses work independently. Since the functions of detectors for different poses might have lots of intersections, multiple penguin heads are often detected at nearby location. Figure 35 shows the multiple detection results.

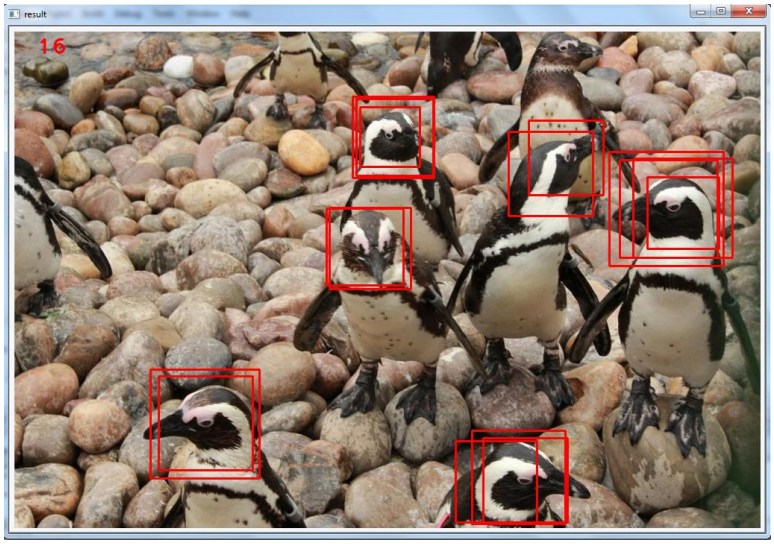


Figure 35: Multiple detections

Multiple detections will cause counting errors. Therefore, multiple nearby detection results should be merged and only one final detection for one penguin head should be reserved. There are many methods can be used for solving this problem. One method is based on the distance between the two models (See Figure 36 (a)). If the distance is smaller than the threshold, the two models should be merged.

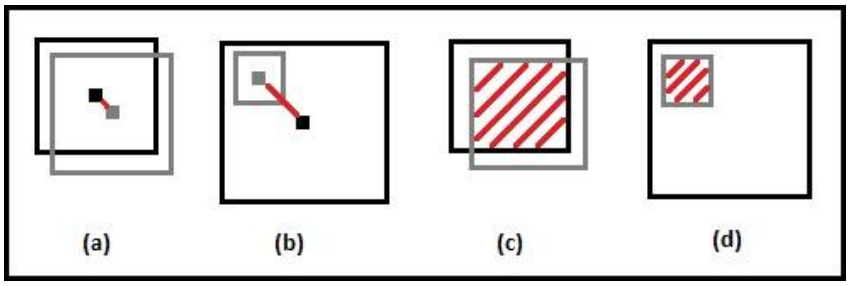


Figure 36: Two merging methods: (a) (b) distance and (c) (d) overlapping

However, this method has a defect when one model is inside another one. As we can see from Figure 36 (b), although the two models are multiple detections, the distance between them might not be small enough to reach the threshold. Therefore, I used another method which is based on the overlapping area between two nearby models. If the overlapping area is larger than the area of any one of the two models multiplied by threshold, the two models should be merged (See Figure 36 (c)). As we can see from Figure 36 (d), this method still works when one model is inside another one. The pseudo-code for this method is as follows. Figure 37 shows the correct detection result after merging multiple ones.

```
for(int a=0; a< number of models; a++)
{    for(int b=a+1; b< number of models; b++)
     {    if (two models overlap each other)
          {    Calculate_overlap_area()
               Calculate_areaA()
               Calculate_areaB()
               if (overlap_area >areaA * threshold ||  overlap_area >areaB * threshold)
               {    merge the model[a] with model [b]    }
     }    }    }
```

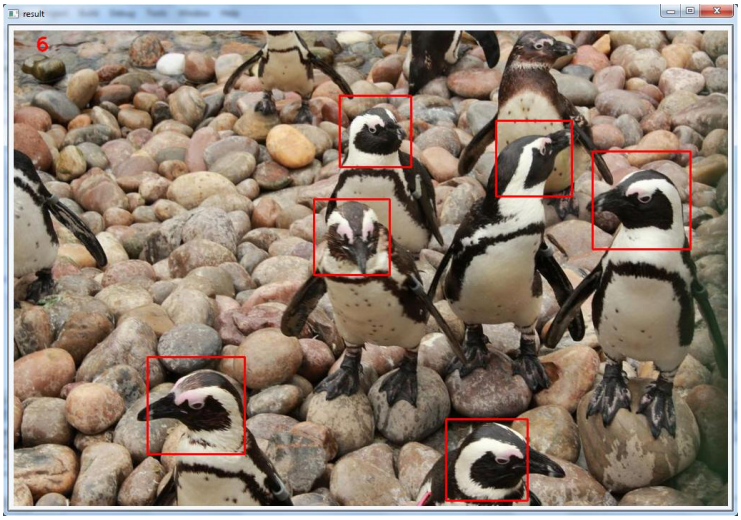


Figure 37: Detection results after combining multiple detections into a single detection

5.1.7 Test detection performance

The final detector was tested on a set of test images. Table 5 shows the results of the testing. As we can see from the table, the average genuine acceptance rate (GAR) of the final detector is 92.24%. The test image is 1000 by 750 pixels image. In each image, the number of negative windows that will be checked by the Viola-Jones method (considering scale and shift invariance) is about 2,146,249. The average false acceptance rate (FAR) (per window) of the final detector is 6.0571E-07.

As mentioned before, because the number of test negative images is far smaller than the number of negative images used for training. We cannot compare the test FAR (per window) with the training FAR (per window).

Table 5: GAR and FAR (per window) of the final detector on test set

No.	Image Name	Hand Labeled	Detection	Hits	Missed	False	GAR (%)	FAR (per window)
1	test_004.jpg	13	13	12	1	1	92.31	4.6593E-07
2	test_005.jpg	10	9	8	2	1	80	4.6593E-07
3	test_007.jpg	12	11	10	2	1	83.33	4.6593E-07
4	test_008.jpg	11	10	10	1	0	90.91	0

5	test_009.jpg	12	13	11	1	2	91.67	9.3186E-07
6	test_011.jpg	14	18	14	0	4	100	1.8637E-06
7	test_012.jpg	12	15	12	0	3	100	1.3978E-06
8	test_015.jpg	13	11	11	2	0	84.62	0
9	test_022.jpg	4	3	3	1	0	75	0
10	test_024.jpg	4	5	4	0	1	100	4.6593E-07
11	test_048.jpg	13	14	9	4	5	69.23	2.3297E-06
12	test_050.jpg	12	13	13	0	1	100	4.6593E-07
13	test_051.jpg	12	14	12	0	2	100	9.3186E-07
14	test_052.jpg	11	11	11	0	0	100	0
15	test_054.jpg	11	11	11	0	0	100	0
16	test_055.jpg	9	9	7	2	2	77.78	9.3186E-07
17	test_083.jpg	3	4	3	0	1	100	4.6593E-07
18	test_572.jpg	4	4	4	0	0	100	0
19	test_596.jpg	3	4	3	0	1	100	4.6593E-07
20	test_650.jpg	6	7	6	0	1	100	4.6593E-07
			AVG				**92.24**	6.0571E-07

The example in Figure 38 shows good detection performance while the examples in Figure 39 show the inadequacy. Viewed from the side, no matter the color or pattern, penguin's flipper looks very much like the penguin head. It is very difficult to distinguish between them for a haar-detector. A better solution is to train another classifier especially for the penguin's flipper. By finding the relationship between the flipper and the penguin head, we can finally distinguish them. Figure 40 shows the detection performance after added the back detector.

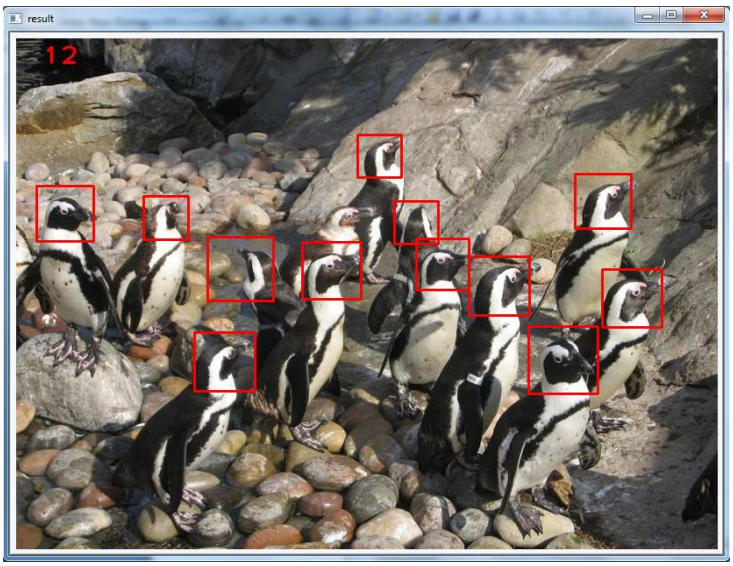


Figure 38: Example of good detection performance

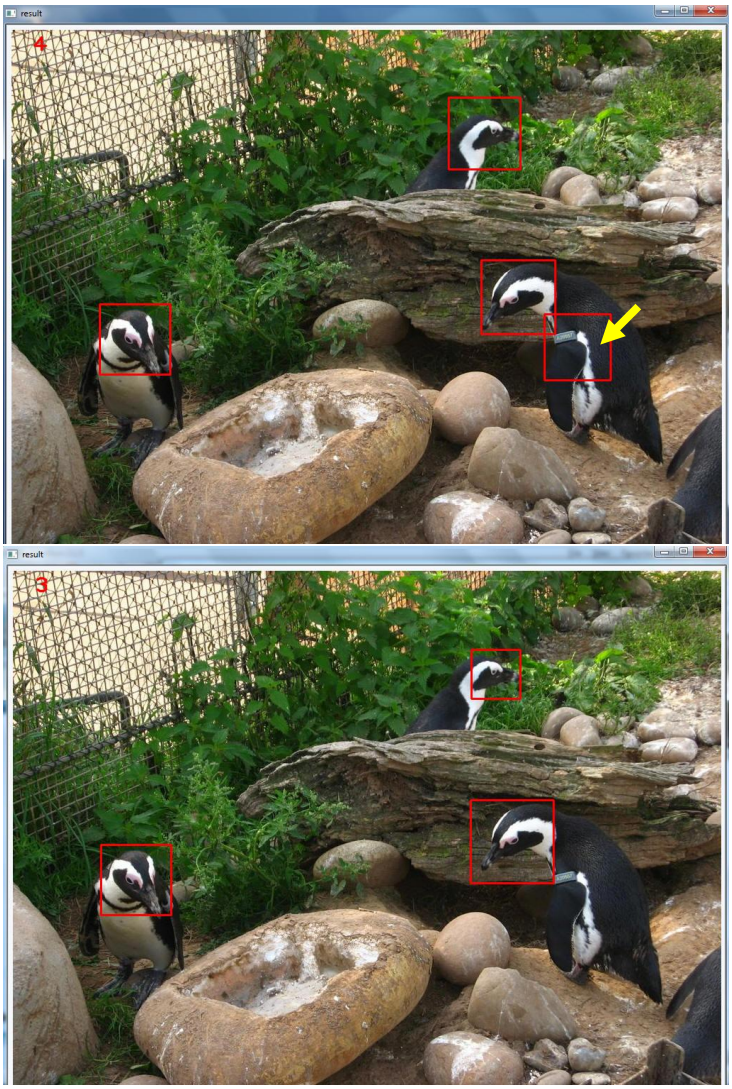
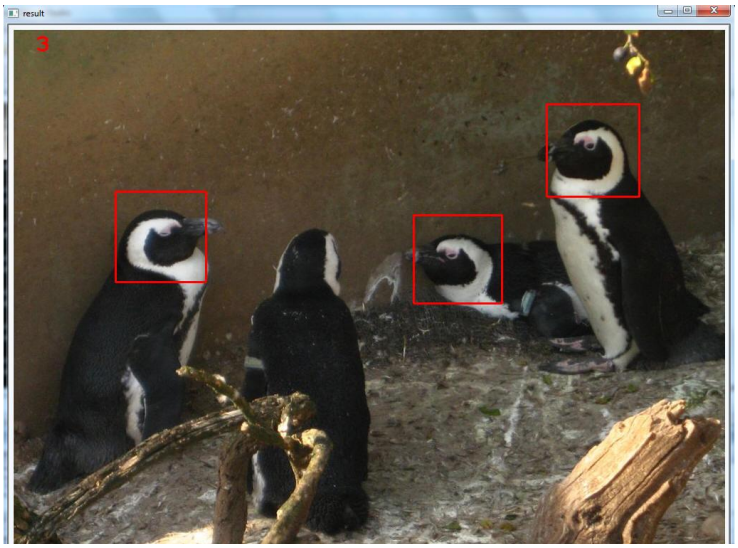


Figure 39: Examples show the inadequacy

Figure 40 shows the false positives made by the back pose detector. Although the back detector can detect the back of penguin head, the false positive rate is relatively high.



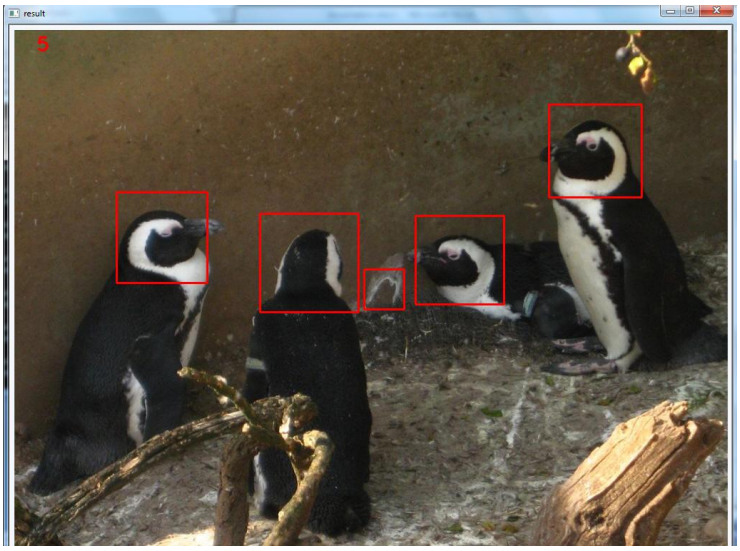


Figure 40: (a) The detection result without using back detector
(b) The false positives made by back detector

5.2 Penguin head tracking

5.2.1 Find good features on penguin head

As we mentioned in chapter 3.2.1, if we want to track an object in the video stream, we need to find some low-level image features to support the tracking. In this project, we used Shi and Tomasi's corner detector [17] to find the corner points on the penguin head.

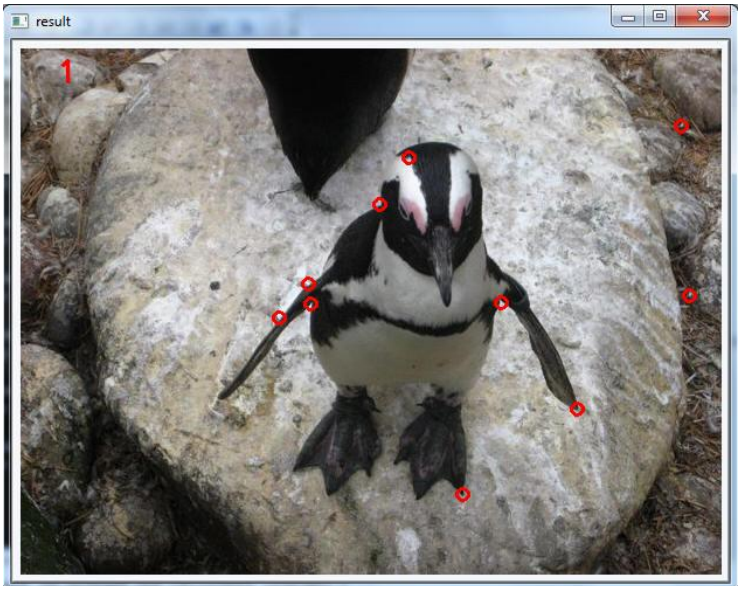


Figure 41: The corner points in the whole image

Obviously, selecting the corner points from the whole image will not help us to track the penguin head. Figure 41 shows the corner points in the whole image. Therefore, we need to set a region of interest before selecting the feature. The region of interest is the penguin head that we have detected.

In this project, I used the *cvSetImageROI* function to set the region of interest. All the labeled

penguin head have been saved in an array called r*ecarray[ ]*. Hence, the process of selecting corner points should be in a for-loop through every element of *recarray[ ]*. The coordinates of the selected corner points are the relative position in the region of interest. If we want to get their absolute position in the original image, we need to add the coordinates of the region of interest to the relative position. Figure 42 shows the corner points in the region of interest - the penguin head.

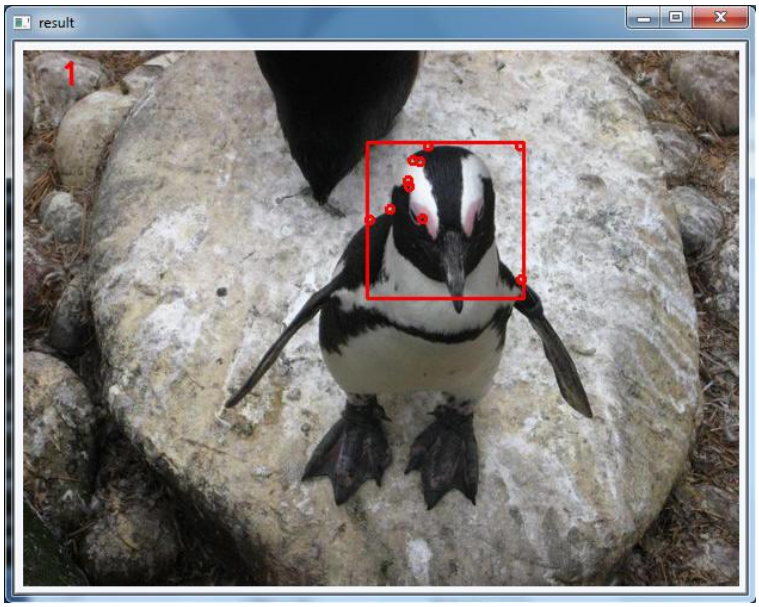


Figure 42: The corner points in the region of interest

The pseudo-code for this method is as follows.

```
void  find_corner(IplImage* img,CvRect recarray[])
{
    for(every element in recarray[])                    /*for every interest model*/
    {
        int x,y,w,h;
        x=recarray[m].x;
        y=recarray[m].y;
        w=recarray[m].width;
        h=recarray[m].height;
        cvSetImageROI(img, cvRect(x, y, w, h));
        imgroi = cvCreateImage(cvGetSize(img),img->depth,img->nChannels);
        cvCopy(img, imgroi, NULL);
        cvResetImageROI(img);
        cvGoodFeaturesToTrack();
        draw_circle();
    }
}
```

After determining the region of interest, we can find good features in that region. The function *cvGoodFeaturesToTrack* in OpenCV library implements Shi and Tomasi's corner detector [24]. Given the required parameters, this function can find the corners with big eigenvalues (strong corners) in an image.

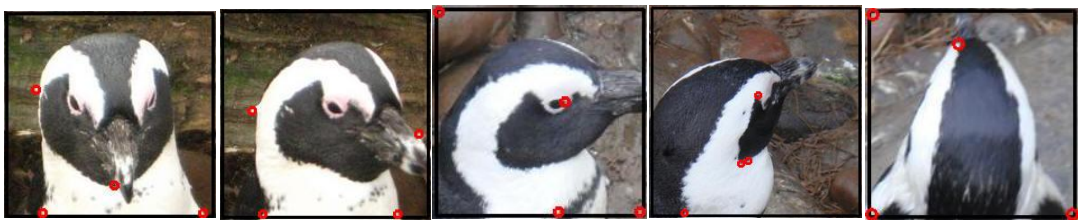


Figure 43: The selected corner features in different poses before reducing the size of region

Figure 43 shows the result of corner feature selection in different poses before reducing the size of region. However, as we noticed, some selected corners are on the edges of the region of interest. The penguin head usually appears at the center of the region while the background always appears near the edges of the region. Therefore, the selected corners which are on the edges of the region of interest are more likely to be the feature of the background rather than a penguin head. Using those corners for tracking the penguin is unsuited. Thus, I made a small change to the program. I slightly reduced the size of the interested region used for selecting corners. The changed part of the code is as follows.

```
x=recarray[m].x+0.1*recarray[m].width;
y=recarray[m].y+0.1*recarray[m].height;
w=recarray[m].width-0.2*recarray[m]. width;
h=recarray[m].height-0.2*recarray[m].height;
cvSetImageROI(img, cvRect(x, y, w, h));
```

Figure 44 shows the result of corner feature selection in different poses after reducing the size of region.

Figure 44: The selected corner features in different poses after reducing the size of region

5.2.2 Track the features

After finding the good features to track, we can the track those features from frame to frame. In this project, we use Kanade-Lucas-Tomasi feature tracker which have been described in detail in theoretical basis part. The function *cvCalcOpticalFlowPyrLK* in OpenCV library implements the sparse iterative version of the Lucas-Kanade optical flow in pyramids [24]. Given the previous frame, current frame, buffer for the pyramid for previous frame and current frame, the coordinates of features (the corner points) on the previous frame, and some

other parameters, this function can calculate the new coordinates of features on the current frame. A series of screenshots of frames (See Figure 45) shows the result of tracking the corner points on the penguin head. The red points represent the tracked points by using KLT tracker.

Figure 45: The screenshot of frames shows the result of tracking the feature points.

As we can see from the frames at earlier stages, the tracked points always move with the penguin head. However, when the penguin head moves very fast, the tracked points drift strongly and finally leave the penguin head. This is unavoidable. Therefore, if we want to use these tracked points to help us for tracking the penguin head, we need to figure out a way to update them dynamically. I will describe this method in detail in the next section.

5.2.3 Track the penguin head

As we described in the last section, now, we can track the corner points from frame to frame. However, we cannot track the penguin head by using the tracked points alone. We need to combine the tracked feature points with new detections.

As we can see from Figure 46, in the 1st frame, we get two interest models by using Viola and Jones object detector [1]. Then, we use Shi and Tomasi corner detector [17] to select four

strong corners for each model. In the 2nd frame, the corner points found in the 1st frame are tracked utilizing a pyramidal implementation of Kanade-Lucas-Tomasi feature tracker [19]. As a result, we can get a new set of feature points in the 2nd frame. At the same time, we use the object detector again in the current frame to get a new set of interest models. Then, we can get the estimated position of the penguin head by using a combination of tracked feature points and new detected interest models.

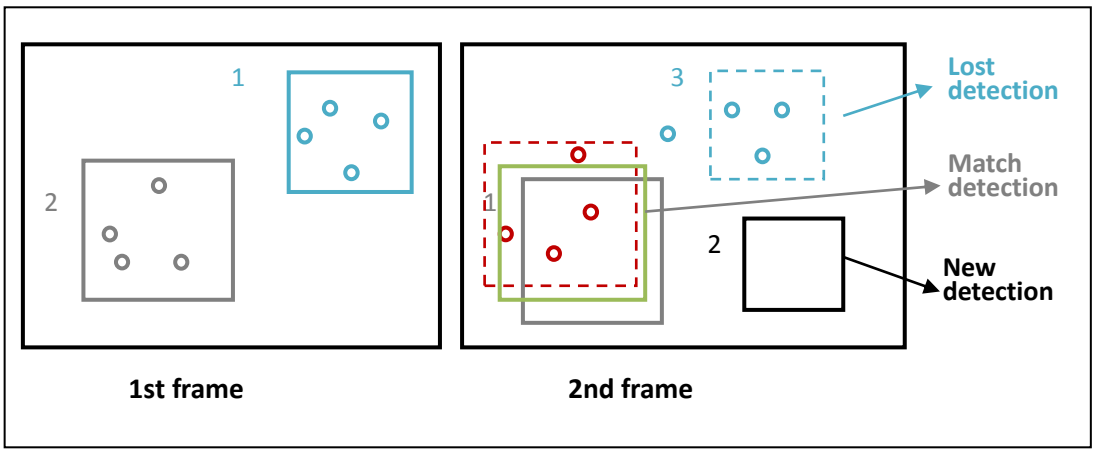


Figure 46: Three possible cases

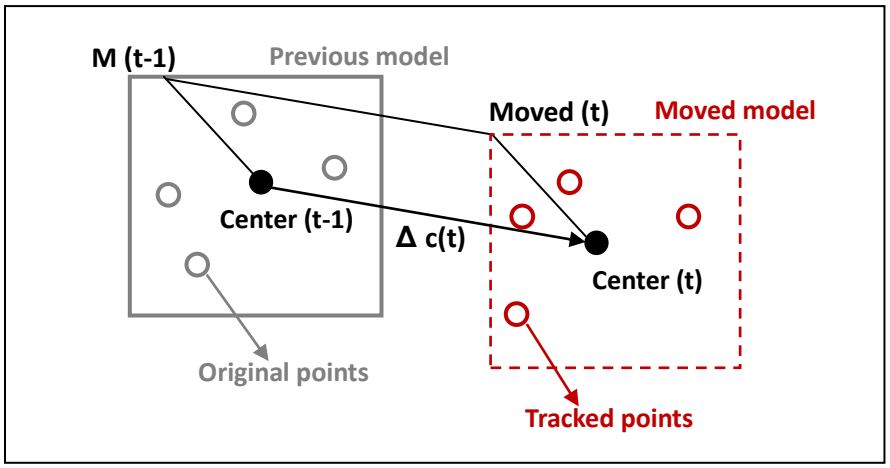


Figure 47: Visualization of the calculation of the moved model [25]

Firstly, for each interest model, we calculate the central point of the feature points in the previous frame and the central point of the tracked feature points in the current frame. And then, for each interest model, we can get its frame-to-frame position displacement $\Delta c(t)$ and use $\Delta c(t)$ to calculate the moved model in the 2nd frame.

$$\Delta c(t) = Center(t) - Center(t-1)$$

$$Moved(t) = \Delta c(t) + M(t-1)$$

Figure 47 shows the calculation of the moved model. This method is based on the algorithm described by Tilo Burghardt and Janko Calic in the study of Real-time Face Detection and Tracking of Animals [25].

By far, we get two sets of models in the 2nd frame – a set of new detected models and a set of

moved models. By calculating the overlapping area between the two sets of models, we can define three possible cases (See Figure 46). If the proportion of overlapping area is bigger than a threshold, it is case 1. Otherwise, it is case 2 or case 3. The pseudo-code for calculating the proportion of the overlapping area is as follows.

```
for(every new detection)                                    /* current*/
{    for(every moved model)                                 /* previous*/
     {
          if ( moved.x< (new.x+ new.width)  &&  (moved.x+ moved.width )> new.x
          &&  moved.y<( new.y+ new.height)  &&  (moved.y+ moved.height)> new.y  )
          {
          w=min(moved.x+ moved.width, new.x+ new.width)-max(moved.x, new.x);
          h=min(moved.y+ moved.height,rnew.y+ new.height)-max(moved.y, new.y);
          overlapping=w*h;
          moved_area=moved.width * moved_height;
          new_area= new.width* new.height;
          if(overlapping/ moved_area  >threshold || overlapping/ new_area > threshold)
          {    }                                              /* It is match detection*/
} } }
```

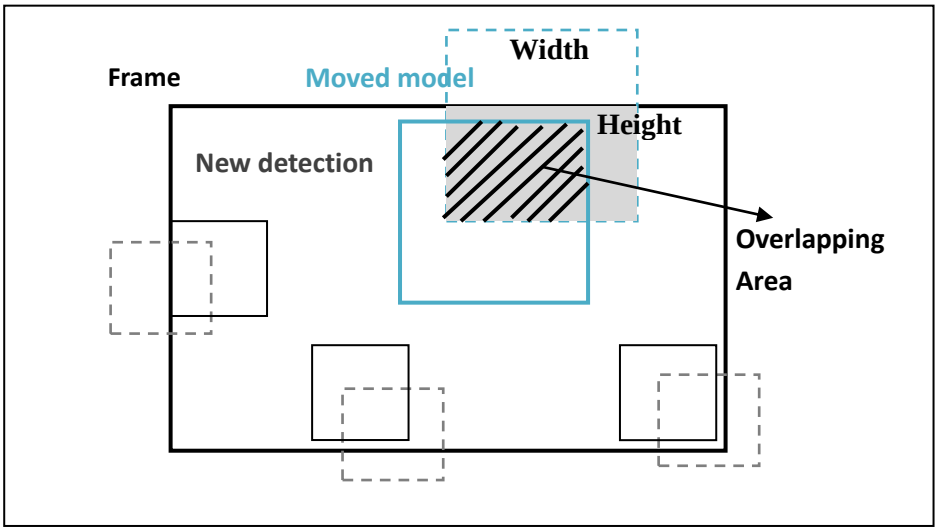


Figure 48: exceptional case- beyond the boundary.

However, as we can see from Figure 48, one or two edges of the calculated moved model may be beyond the boundary of the frame. In this case, if we still define:

$$moved\ model.width\ \times\ moved\ model.height = moved\ model.area.$$

$$proportionA = \frac{overlapping}{moved\ model.area}, propotionB = \frac{overlapping}{new\ detected\ model.area}$$

The proportion A might be very small. If the proportion B is not big enough to reach the threshold, we will miss this match detection.

Therefore, we need to redefine the area of the moved model. The real area of the moved model is where inside the boundary of the frame (the gray region in Figure 48). The following code should be added.

```
if (moved.x + moved.width <=frame->width  &&  moved.x >=0
        &&  moved.y + moved.height<= frame ->height  &&  moved.y >=0 )
    {
        w2= moved.width;
        h2= moved.height;
    }
if(moved.x+ moved.width> frame ->width)                    /* right*/
    {   w2= frame ->width-moved.x;   }
if(moved.x<0)                                              /* left*/
    {   w2= moved.width+ moved.x;   }
if(moved.y+ moved.height> frame ->height)                  /* down*/
    {   h2= frame ->height- moved.y;  }
if(moved.y >0)                                             /* up*/
    {   h2= moved.height+ moved.y;   }
    moved_area =w2*h2;
```

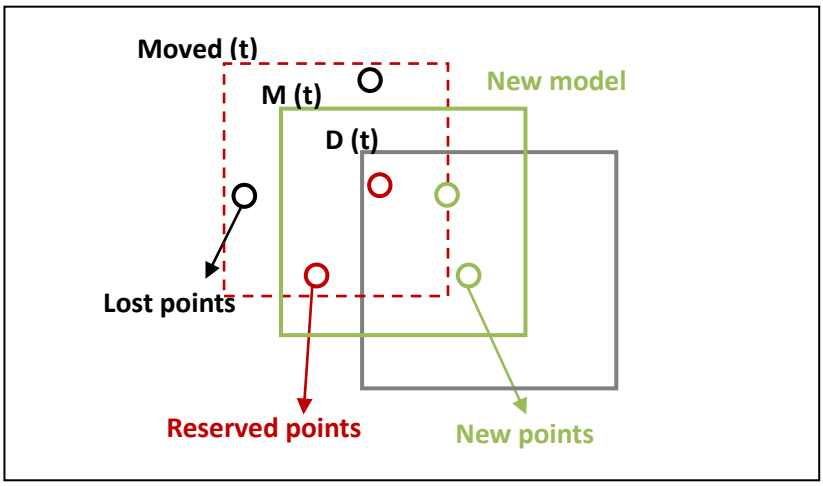


Figure 49: Visualization of calculation of new model and dynamic updating of feature points

Then, I will describe the three possible cases in detail.

CASE 1: MATCH DETECTION

If a new detected model and a moved model match, it is case 1- match detection. In this project, case 1 means the penguin head detected in this frame has been found in the last frame. In this case, we can calculate the estimated position of the penguin head by using the detected model and moved model together (See Figure 49).

$$M(t) = \left(\alpha \times Moved(t) + \beta \times D(t)\right) \times (\alpha + \beta)^{-1}$$

More specifically, α and β represents the weight of moved model and new detection model respectively in computing the new model. If β is much larger than α, the estimated new model will entirely rely on the new detection. Because of the spatial instability of the Haar-detector, the jumps of new model position are unavoidable. On the contrary, if α is much larger than β, the new model will entirely rely on the tracked features, the trajectory will be

more smooth but the model will drift strongly. There is a tradeoff between spatial jumps and model drift. Therefore, we need to find a compromise point between them. According to experimental results for maximizing the tracking accuracy, in this program, α is set to 3 while β is set to 7.

As we mentioned in section 5.2.2, when the target object moves very fast, the tracking points drift strongly and might leave the interest model. Therefore, after we got the new model position, we need to update the feature points dynamically as well. The points inside the new model are reserved. Then, we replace the points which are outside the new model with new corner points selected by using Shi and Tomasi corner detector again [17][25].

An array called *status* is used to indicate the confidence of the correct detection. In another word, high status value represents great confidence in correct detection, vice versa. The confidence accumulation strategy is based on the algorithm described by Tilo Burghardt and Janko Calic in the study of Real-time Face Detection and Tracking of Animals [25]. If it is match detection, the status value for this model will plus one.

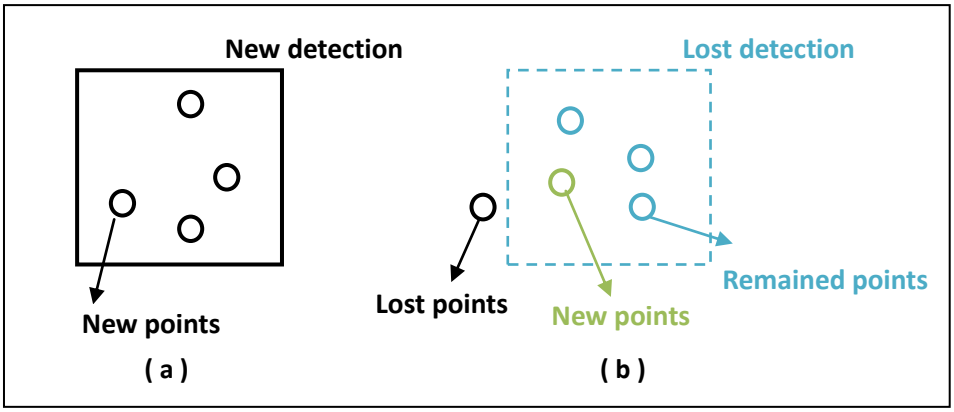


Figure 50: (a) Visualization of the new detection
(b) Visualization of the lost detection

CASE 2: NEW DETECTION

If a new detected model does not match any of moved models, it is case 2- new detection. In this project, case 2 means the penguin head detected in this frame has not been found in the last frame. In this case, the new detection is the new interest model. And new corner features should be selected in the new interest model utilizing Shi and Tomasi corner detector [17] (See Figure 50 (a)). The status value for this new model will be initialized to zero.

CASE 3: LOST DETECTION

If a moved model does not match any of new detected models, it is case 3- lost detection. In this project, case 3 means the penguin head which has been found in the last frame is not detected in this frame. In this case, the status value for this model will minus one.

Then, if the status value is smaller than a threshold, the lost model will be reserved for this time. And, we update the feature points for this model dynamically as well (See Figure 50(b)). In contrast, if the status value is larger than the threshold, the lost model will be discarded and the feature points will not be tracked any more.

5.2.4 Test tracking performance

The screenshots of frames (See Figure 51) shows the result of tracking the penguin head from frame to frame. Although the fast motion will cause the tracked points drift strongly and even leave the interest model, we can solve this problem by updating the points dynamically. As we can see from the frames at later stages, although the penguin head moves very fast, the tracked points still always move with the penguin head. The green points represent the new corner points selected by Shi and Tomasi corner detector while the red points represent the tracked points by using KLT tracker. The red rectangle represents the interest model.

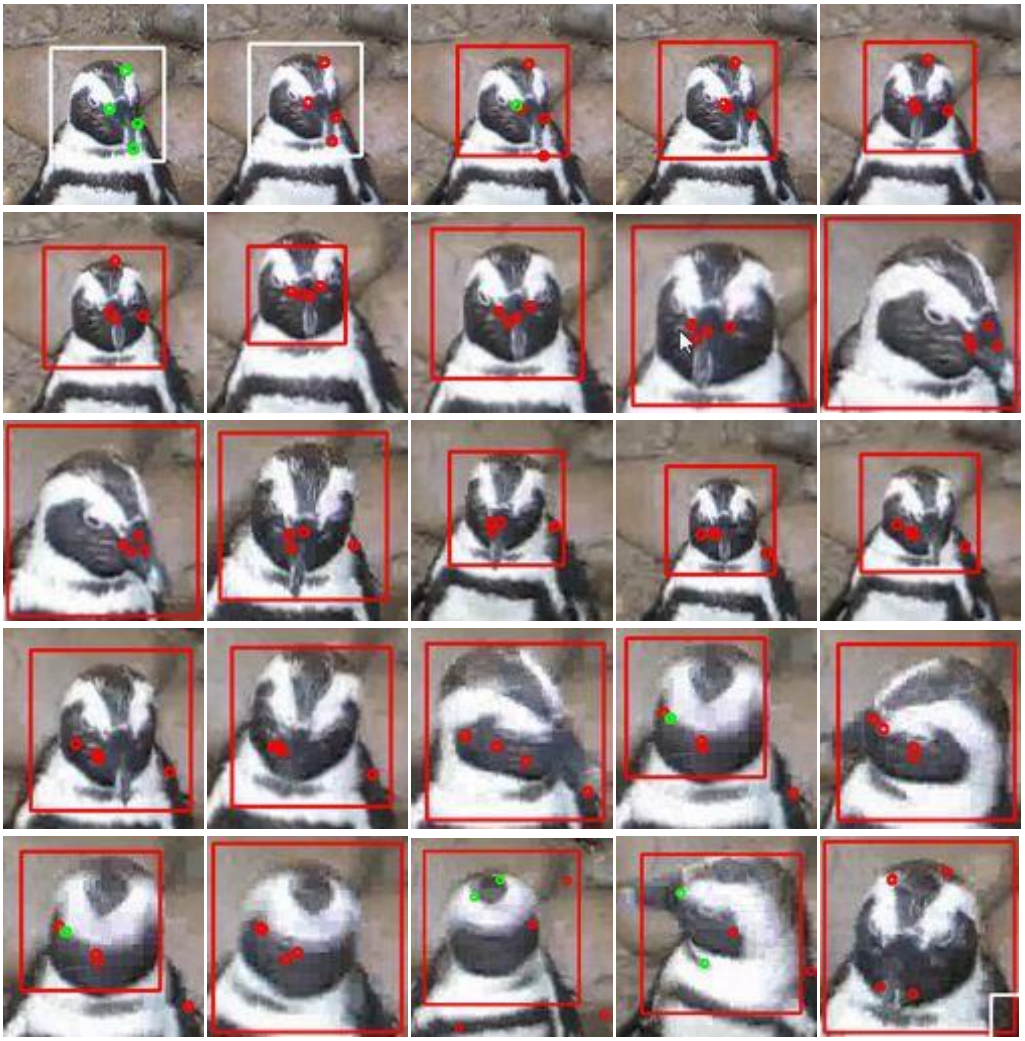


Figure 51: The screenshot of frames shows the result of tracking the penguin head

5.3 Penguin head counting

5.3.1 Counting implementation

After combining the features tracking with new detection, now, we can track the penguin head from frame to frame. On this basis, the penguin head counting can be implemented. The

following flow chart (See Figure 52) describes the process of dealing with the three cases discussed in the last section 5.2.3 and the counting strategy.

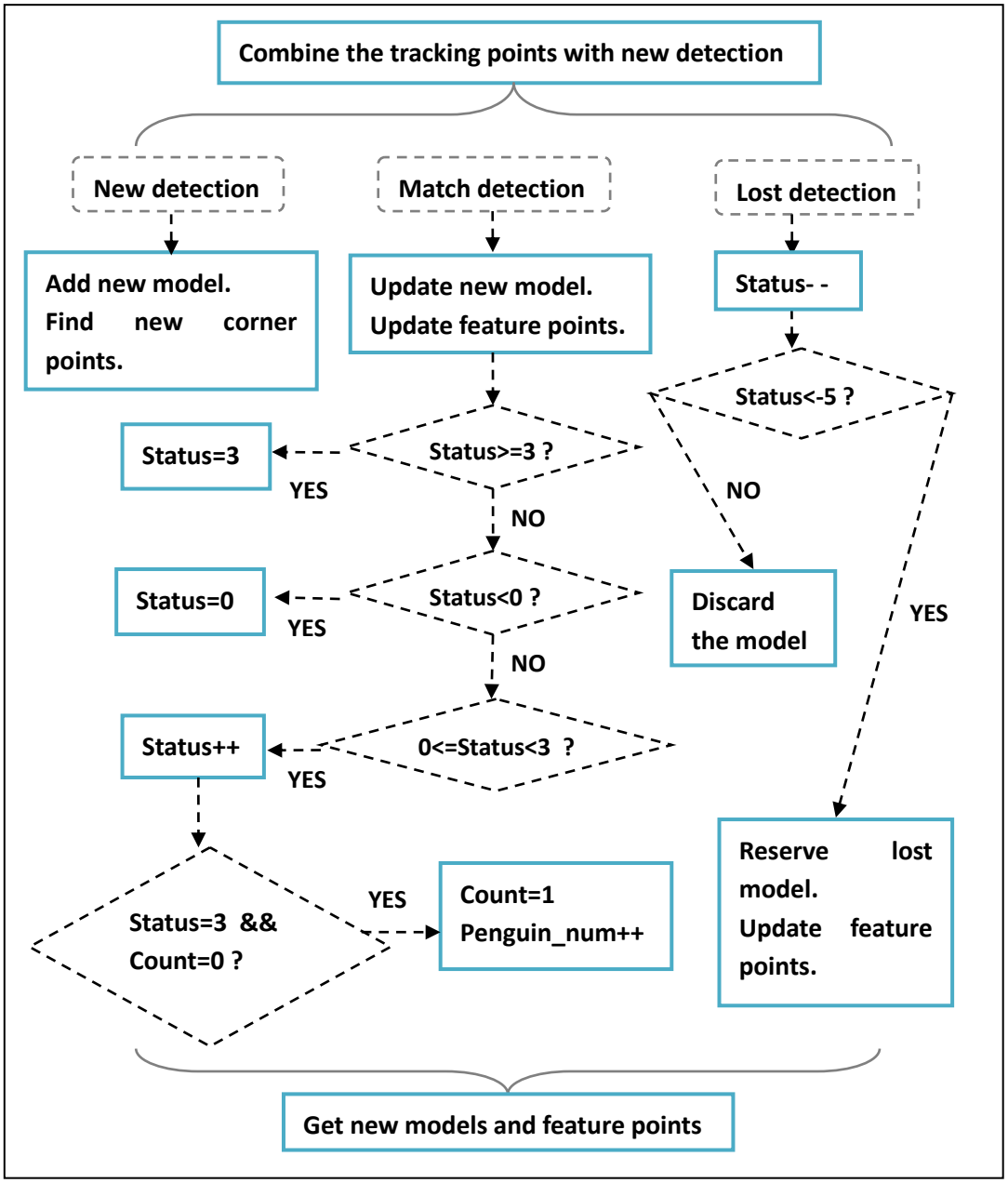


Figure 52: Visualization of the combination of tracking and detection and counting strategy

As we mentioned before, the *status* array is used to indicate the confidence of the correct detection. The *status* value has two thresholds. The negative one is used to determine whether a model should be discarded while the positive one is used to determine whether a model should be counted. According to experimental results for maximizing the counting accuracy, in this program, the negative threshold is set to -5 while the positive threshold is set to 3. Another array, namely *count* is used to record the counting status. More specifically, if the interest model has been counted, the count value will be set to 1, otherwise it will be set to 0. Besides, another two arrays, namely *pre_status* and *pre_count* are used to reserve the previous *status* and *count* value for each model.

As we can see from the flow chart, if it is match detection, the program will check the *pre_status* value for this model. Only when *pre_status* is in the range of 0 and 2, *status*=*pre_status*+1. Then if the *status* value reaches 3 and the *pre_count* equals 0, this model will be counted and the *count* value will be set to 1. Figure 53 shows two possible value curves of the *status*. The red line indicates the status value reaches 3 at the 5th frame while the blue line indicates the status value drops to -5 at the 7th frame.

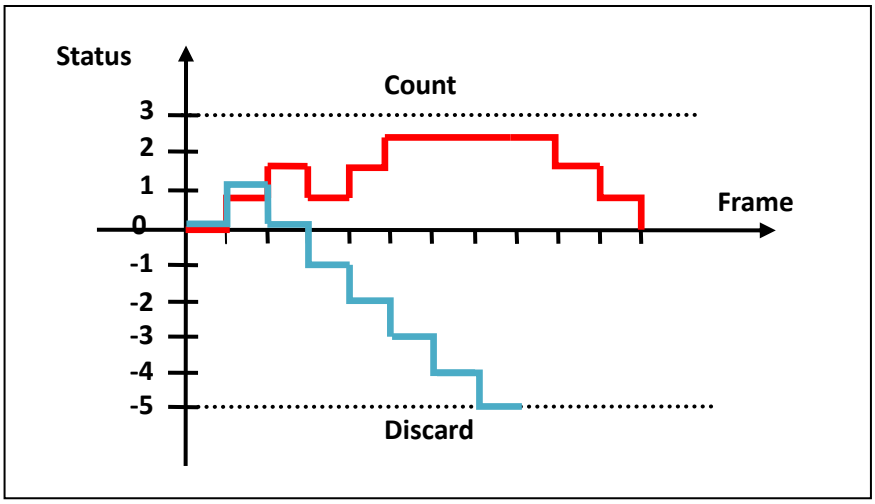


Figure 53: Two possible value curves of the 'status'

5.3.2 Algorithm design

The pseudo-code for the counting implementation is as follows.

```
for(every new detection)                                    /* current*/
{
    for(every moved model)                                  /* previous*/
    {
        Calculate_overlapping()
        if(overlapping area > threshold)                    /* It is match detection */
        {
            Mark_match()                                    /* match[moved model]=1*/
            Update_new_model()          /*new model=new detection+moved model*/
            Update_points()     /*if points outside new model, find_corner_again() */
            if (pre_status>=3)              /* Update status and implement counting*/
            {
                status=3
                count=pre_count
            }
            if(pre_status<0)
            {
                status=0
                count=pre_count
            }
            if (pre_status<3 && pre_status>=0)
```

```
                {
                    status=pre_status+1
                    count=pre_count
                    if(status==3 && pre_count==0)
                    {
                        penguin_num++
                        count=pre_count+1
                    }
                }
                a++;
            }
        }
        if(a==0)                                    /* It is new detection */
        {
            Update_new_model()                      /*new model=new detection*/
            find_corner_again()
            Update_status()                         /* status=0*/
        }
    }
    for(every moved model)                          /* previous*/
    {
        if (match[moved model]!=1)                  /* It is lost detection */
        {
            if(status>-5)
            {
                Update_new_model()                  /*new model= moved model*/
                Update_points()   /*if points outside new model, find_corner_again() */
                Update_status()                     /* status- -*/
            }
        }
    }
```

5.3.3 Test counting performance

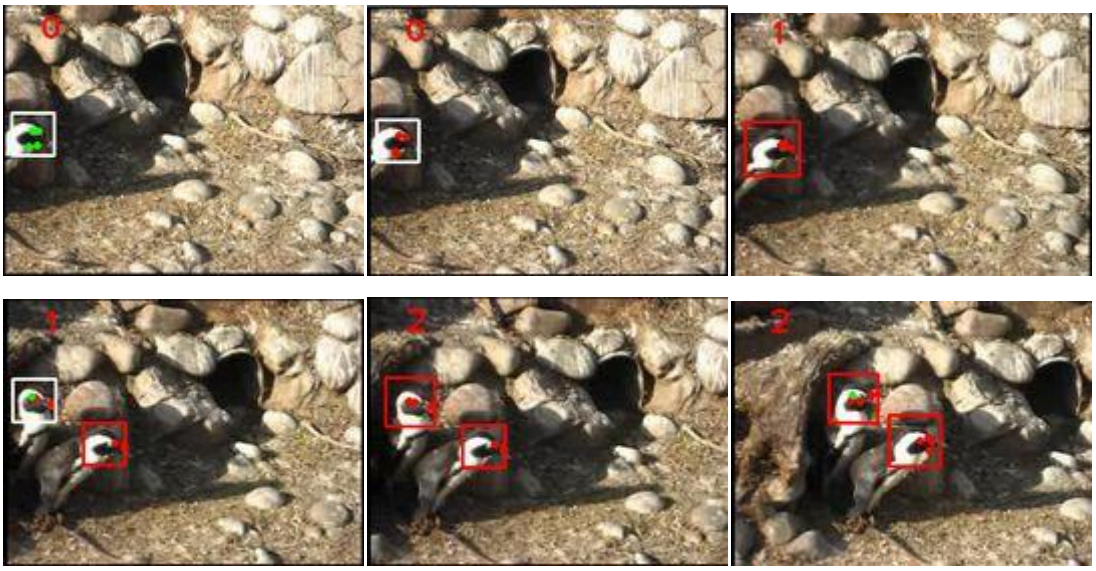
I tested the counting performance on a set of test videos. The following table (See Table 6) shows the results of the testing. As we can see from the table, the average evaluated counting accuracy on these 17 test videos is 91.96%.

Table 6: Testing results

No.	Video Name	Number of Hand Labeled	Number of Detection	Inaccuracy	Evaluated accuracy
1	MVI_8027.avi	1	1	0%	100%
2	MVI_8029.avi	3	1	66.67%	33.33%
3	MVI_8031.avi	3	3	0%	100%

4	MVI_8048.avi	1	1	0%	100%
5	MVI_8049.avi	2	2	0%	100%
6	MVI_8155.avi	3	3	0%	100%
7	MVI_8158.avi	4	4	0%	100%
8	MVI_8159.avi	3	3	0%	100%
9	MVI_8164.avi	2	2	0%	100%
10	MVI_8253.avi	2	2	0%	100%
11	MVI_8254.avi	2	2	0%	100%
12	MVI_8693.avi	3	3	0%	100%
13	MVI_8694.avi	5	6	20%	80%
14	MVI_8699.avi	3	3	0%	100%
15	MVI_8700.avi	2	2	0%	100%
16	MVI_8716.avi	2	2	0%	100%
17	MVI_8744.avi	2	3	50%	50%
		AVG		**8.04%**	**91.96%**

A series of screenshots in Figure 54 shows the counting performance of the penguin head counting system. The green points represent the new corner points selected by Shi and Tomasi corner detector [17] while the red points represent the tracked points by using KLT tracker [22]. The red rectangle represents an interest model which has been counted whilst white rectangle represents an interest model which has not been counted. The red number in the top left corner shows the number of penguins passing the field of view.



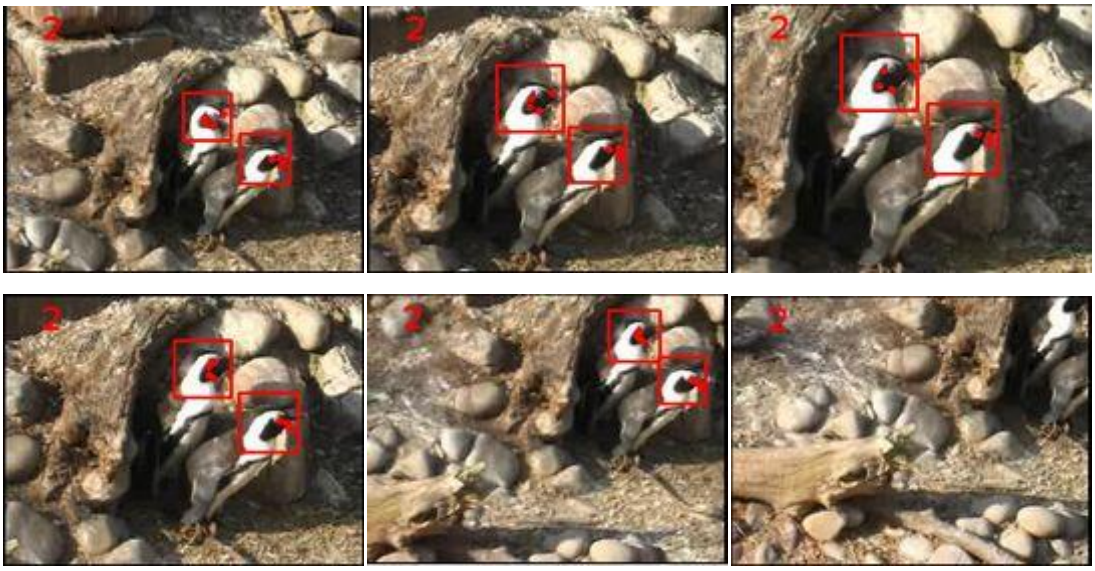


Figure 54: Example screenshots of the penguin head counting system

The following series of screenshots in Figure 55 shows the false positives on flippers. As we can see from the 23th screenshot, the penguin's flipper is mistaken for a penguin's head. As mentioned before, viewed from the side, no matter the color or pattern, penguin's flipper looks very much like the penguin's head. It is very difficult to distinguish between them for the Haar-detector.

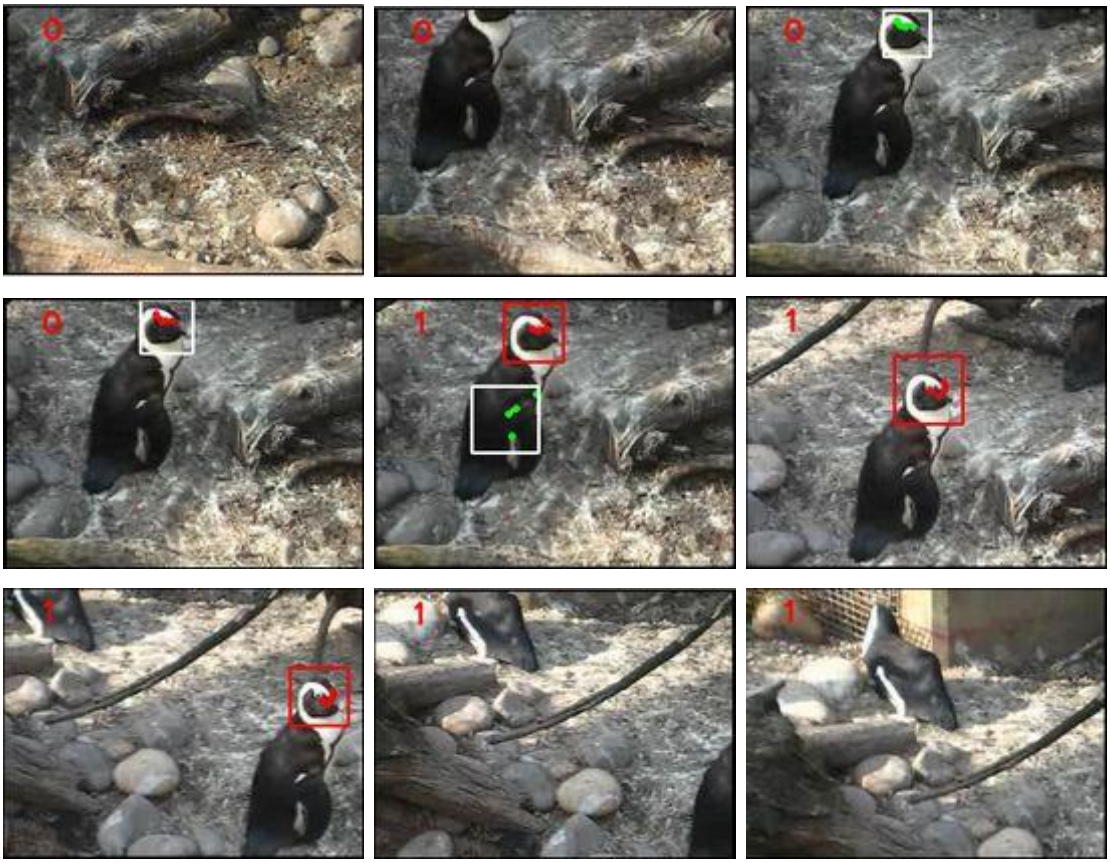


Figure 55: Example screenshots of the penguin head counting system shows the false positives on flippers

The following series of screenshots in Figure 56 show the inadequacy of this system. By far, this system cannot solve the occlusion problem. As we can see from the 7th and 8th screenshots, when a penguin walks through under the branch, the tracked features on the penguin head will be covered. As a result, our system cannot track these features anymore and the tracking will be interrupted.

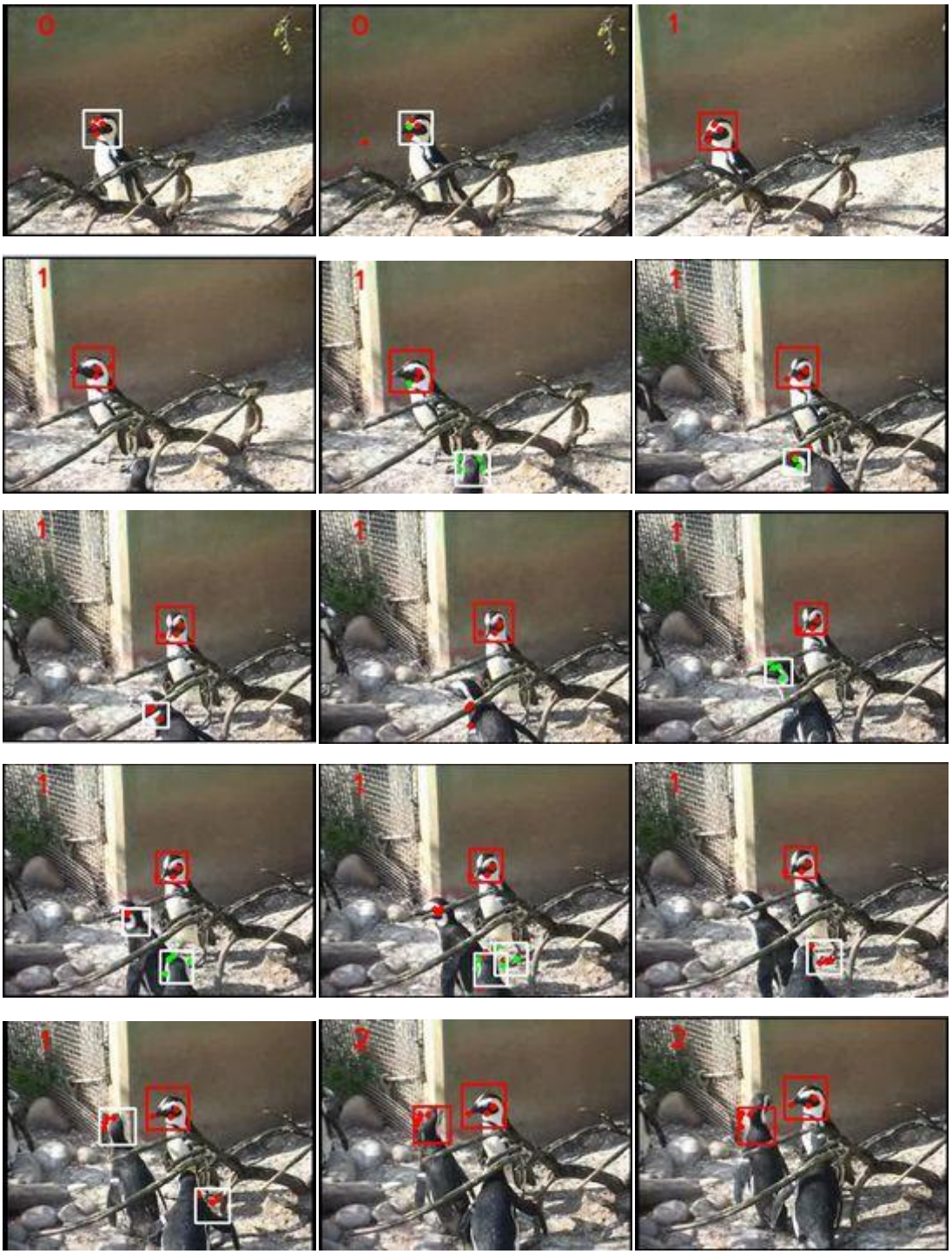


Figure 56: Example screenshots of the penguin head counting system shows the occlusion problem

6. Conclusion and Future Work

6.1 Conclusion

In this thesis, we have presented a vision system that implemented real-time head counting of African penguins in wildlife video. This system is capable of detecting and tracking the penguin heads in bird-view video stream and counting the number of penguins passing by.

The technique used for detecting penguin head is based upon Viola-Jones object detection algorithm [1]. We choose their object detection framework for many reasons. Firstly, its high detection speed by using integral image and cascade structure enables it applied to real-time applications. Secondly, a rich set of Haar-like features they used is good at modeling high-contrast penguin head. Thirdly, the effective AdaBoost learning algorithm [4] can construct strong classifiers by using the most representative features.

The technique used for tracking is based upon Shi- Tomasi [17] low-level feature selection and Kanade-Lucas-Tomasi Feature tracker [19] [22] [23]. By combining the object detection and features tracker [25], this system achieved smooth penguin head tracking and accurate penguin head counting in their natural environment.

The multi-poses detection performance was evaluated by using 20 test images. The average GAR is 92.24% while the average FAR (per window) is 6.0571E-07. The counting performance was evaluated by using 17 test videos. The average counting accuracy reaches 91.96%.

6.2 Future Work

The presented penguin head counting system can be improved and expanded in the following areas.

1) Feature enhancement

Although Haar-like features can model the penguin head very well, they still have limitation. They are very good at modeling those objects with regular rectangular shapes. However, for modeling a complex object, several overlapping Haar-like features are needed to achieve a better modeling performance. Obviously, the more features used, the more computation time needed.

Therefore, we might need to expand the feature set. There are many feature representations in image processing. A very useful one for this project is the color feature. The black and white penguin is colorless while the background is colored. Using color features can decrease the number of false positives (mark the background region as a penguin head).

2) False positives on flippers

Viewed from the side, no matter the color or pattern, penguin's flipper looks very much like the penguin head in profile. Therefore, the penguin's flipper is always marked as a penguin head incorrectly by the detector for side pose. It is very difficult to distinguish them for a

classifier trained using either Haar-like features or color features. A better solution is to train another classifier especially for the penguin's flipper. By finding the relationship between the flipper and penguin head, we can finally distinguish them.

3) Effective detector for back pose

Because of the lack of significant characteristics on the back of a penguin head, the AdaBoost learning algorithm cannot find enough features to train an effective classifier. Therefore, the performance of the detector for the back pose is unsatisfactory. The genuine acceptance rate of the detector is only 60% while the false acceptance rate (FAR) (per window) is much higher than the other four detectors.

For training a more effective detector for the back pose, more characteristics information is needed. For instance, instead of only using the back of the penguin head, we can use the back of the whole penguin body as positive sample to train the classifier. Therefore, the AdaBoost learning algorithm can find more features for training a classifier.

4) Solve the occlusion problem

By far, this system cannot solve the occlusion problem. As we know, occlusion is an unavoidable subject in most machine vision areas. There are several methods can help to solve this problem.

One method is based on the moving trajectory. More specifically, by tracking the moving penguin, we can estimate its moving trajectory. When the penguin is covered by some objects and the tracking is interrupted, we can predict the position where it may appear again by using the estimated moving trajectory. If the penguin appears at that position within a short time, we assume it is the one that just disappeared and the interrupted tracking can be continued.

Another method is based on the unique feature. Every penguin has its unique characteristics. By using its characteristics, we can determine whether the new detected penguin is the one that just disappeared.

7. Bibliography

[1] Paul Viola and Michael Jones. Robust Real-time Object Detection. International Journal of computer Vision , page 1-25, July 13, 2001

[2] C.Papageorgiou, M.Oren and T.Poggio. A general framework for object detection. International Conference on Computer Vision, 1998.

[3] Rainer Lienhart and Jochen Maydt. An Extended Set of Haar-like Features for Rapid Object Detection. Intel Labs, Intel Corporation, Santa Clara, CA 95052, USA, page1-2, 2002

[4] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences, no. 55. 1997

[5] Philip A. Whittington, Robert J.M. Crawford, Bruce M. Dyer, Anthony J. Williams et al. Return to Robben Island of African Penguins that were rehabilitated, relocated or reared in captivity following the Treasure oil spill of 2000. The Journal of African Ornithology. Page 1, November 2006.
[weblink:http://www.britannica.com/bps/additionalcontent/18/23728012/Return-to-Robben-Island-of-African-Penguins-that-were-rehabilitated-relocated-or-reared-in-captivity-following-the-Treasure-oil-spill-of-2000]. June 2, 2010

[6] Nic Huin. Falkland islands penguin census 2005/06. Published by Falklands Conservation. Page 3-5,April, 2007

[7] Tilo Burghardt, Barry Thomas, Peter J Barham. Automated Visual Recognition of Individual African Penguins. Page 1-10, September 2004
[weblink: http://www.spotthepenguin.com/]. June 6, 2010

[8] Konstantinos G. Derpanis. Integral image-based representations. July 14, 2007

[9] Peter J. Barham, Innes C. Cuthill, Neill Campbell, Tilo Burghardt, Richard Sherley Recognition System.
[weblink: http://www.spotthepenguin.com/]. June 6, 2010

[10] F. Estrada,A. Jepson,D. Fleet. Local Features Tutorial, page 4, Nov. 8, 2004

[11] Julien Meynet,Vlad Popovici,Jean-Philippe Thiran.Face detection with boosted Gaussian features. Page 2-3,2007

[12] Chengjun Liu,Harry Wechsler. Independent component analysis of Gabor features for face recognition. IEEE Trans. Neural Networks, vol. 14, no. 4, pp. 919-928, 2003

[13] Haar A. Zur Theorie der orthogonalen Funktionensysteme, Mathematische Annalen, 69, pp 331–371, 1910.

[14] Tilo Burghardt. Penguin head counts. Ideas for Student Projects.
[weblink:http://combine.cs.bris.ac.uk:8080/opencms/opencms/exabyte/projects/media/event_0002.html]. Aug 7, 2010

[15] A short introduction to boosting. Marc Bron. Page 21-22. September 29, 2006

[16] Lubomir Bourdev, Jonathan Brandt. Robust Object Detection Via Soft Cascade. page 2, 2005

[17] J. Shi and C. Tomasi. Good Features to Track . 9th IEEE Conference on Computer Vision and Pattern Recognition, Seattle, June 1994

[18] Chris Harris and Mike Stephens. “A Combined Corner and Edge Detector”. Proc. of The Fourth Alvey Vision Conference, Manchester, pp. 147-151. 1988

[19] C.Tomasi and T.Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.

[20] UTKARSH. The Shi-Tomasi Corner Detector [weblink: http://www.aishack.in/2010/05/the-shi-tomasi-corner-detector/] Aug 27, 2010

[21] Hans Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. PhD thesis, Stanford University, September 1980.

[22] Bruce D. Lucas and Takeo Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. International Joint Conference on Artificial Intelligence, pages 674-679, 1981.

[23] Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the algorithm.2002

[24] OpenCV 2.0 C Reference. cv. Image Processing and Computer Vision. [weblink:http://opencv.willowgarage.com/documentation/cv._image_processing_and_computer_vision.html] Aug 29, 2010

[25] Tilo Burghardt and Janko Calic. Real-time Face Detection and Tracking of Animals. 8th Seminar on Neural Network Applications in Electrical Engineering, NEUREL-2006

[26] Optical flow. From Wikipedia [weblink: http://en.wikipedia.org/wiki/Optical_flow] Sept 07, 2010

[27] Lucas–Kanade Optical Flow Method. From Wikipedia [weblink: http://en.wikipedia.org/wiki/Lucas%E2%80%93Kanade_Optical_Flow_Method] Sept 07, 2010

Appendix

List of Files:

Training source
This file contains all the original images used for training classifiers and the created samples for five penguin poses. All the images were taken in Bristol Zoo Gardens.
Download link: http://combine.cs.bris.ac.uk/naiwen/training.zip

Classifiers
This file contains the five trained classifiers for five penguin poses.
Download link: http://combine.cs.bris.ac.uk/naiwen/classifiers.zip

Test images
This file contains 20 images used for testing the detection performance. The testing result can be seen in page 38-39.
Download link: http://combine.cs.bris.ac.uk/naiwen/test_image.zip

Test videos
This file contains 17 videos used for testing the counting performance. The testing result can be seen in page 52-53.
Download link: http://combine.cs.bris.ac.uk/naiwen/test_video.zip

Classifier testing
This file contains 497 positive images and 200 negative images used for testing the classifiers performance. The testing result can be seen in page 34-35.
Download link: http://combine.cs.bris.ac.uk/naiwen/test_full.zip

Sample Code: (Notes: some codes are omitted)

```
int main( int argc, char** argv )
{
    CvCapture* capture = 0;
    IplImage *frame, *frame_copy = 0;
    IplImage *frame1, *frame_copy1 = 0;
    IplImage *frame_current = 0;
     CvFont font=cvFont(2,2);
    char *str =(char*) malloc (sizeof(char));
    const char* input_name;
     input_name = argc > 1 ? argv[1] : 0;
 storage = cvCreateMemStorage(0);
 if( !input_name || (isdigit(input_name[0]) && input_name[1] == '\0') )
    { capture = cvCaptureFromCAM( !input_name ? 0 : input_name[0] - '0' );}
else
    { capture = cvCaptureFromAVI( input_name ); }
 cascade_name1 ="haarcascade1.xml";
   cascade1 = (CvHaarClassifierCascade*)cvLoad( cascade_name1, 0, 0, 0 );
   /* cascade 2,3,4,5 omitted*/
 if( capture )
 {
     frame1 = cvQueryFrame(capture);
    double fps = cvGetCaptureProperty (capture,CV_CAP_PROP_FPS);
    CvSize size;
    size.width=frame1->width;
    size.height=frame1->height;
    CvVideoWriter *writer = cvCreateVideoWriter("record.avi",0,fps,size,1);
    if( !frame_copy1 )
    frame_copy1 = cvCreateImage( cvSize(frame1->width,frame1-
>height),IPL_DEPTH_8U, frame1->nChannels );
    frame_current = cvCreateImage( cvSize(frame1->width,frame1-
>height),IPL_DEPTH_8U, frame1->nChannels );
     if( !frame_copy )
    frame_copy = cvCreateImage( cvSize(frame1->width,frame1-
>height),IPL_DEPTH_8U, frame1->nChannels );

    if(frame1)
    {
        if( frame1->origin == IPL_ORIGIN_TL )
            cvCopy( frame1, frame_copy1, 0 );
        else
            cvFlip( frame1, frame_copy1, 0 );
        detect( frame_copy1 );
        draw(frame_copy1,recNewArray);
        find_corner(frame_copy1,recNewArray);
        sprintf(str,"%d",penguin_num);
        cvPutText(frame_copy1,str,cvPoint(30,30),&font,CV_RGB(255,0,0));
        cvShowImage( "result", frame_copy1 );
```

```
            cvWriteFrame( writer, frame_copy1 );
            cvCopy( frame_copy1,frame_current,0);
}
for(;;)
{
      frame = cvQueryFrame(capture);
       if( !frame )
            break;
       if( frame->origin == IPL_ORIGIN_TL )
            cvCopy( frame, frame_copy, 0 );
       else
            cvFlip( frame, frame_copy, 0 );
      track(frame_current, frame_copy, cornerArray);
      moved_model(frame_copy);
      detect(frame_copy);
      new_model(frame_copy);
      draw(frame_copy,recNewArray);
      cvPutText(frame_copy,str,cvPoint(30,30),&font,CV_RGB(255,0,0));
      cvShowImage( "result", frame_copy );
      cvWriteFrame( writer, frame_copy );
      cvCopy(frame_copy,frame_current, 0 );
      if( cvWaitKey( 10 ) >= 0 )
            break;
}
cvReleaseImage{ /*omitted*/}
} // if capture
else{ capture fail /*omitted*/}
 return 0;
} //end main

void detect( IplImage* img )
{
      cvClearMemStorage( storage );
      CvRect recArray[100] = {0};
      int i=0,j=0,jj=0,k=0,kk=0,l=0,ll=0,m=0;
      img_flip = cvCreateImage( cvSize(img->width,img->height),IPL_DEPTH_8U,
img->nChannels );
      cvFlip( img, img_flip, 1 );
   if( !cascade1 || !cascade2 || !cascade3 || !cascade4)
   {   fprintf( stderr, "ERROR: Could not load classifier cascade\n" );}
   if( cascade1 && cascade2 && cascade3 && cascade4)
   {
      CvSeq* faces1 = cvHaarDetectObjects( img, cascade1, storage,1.1, 2,
CV_HAAR_DO_CANNY_PRUNING,cvSize(24, 24) );
      for(i = 0; i < (faces1 ? faces1->total : 0); i++ )
      {      r1 = (CvRect*)cvGetSeqElem( faces1, i );
            recArray[i]=*r1;
       }
      CvSeq* faces2 = cvHaarDetectObjects( img, cascade2, storage,1.1, 2,
CV_HAAR_DO_CANNY_PRUNING,cvSize(24, 24) );
      for(j = 0; j < (faces2 ? faces2->total : 0); j++ )
      {      CvRect* r2 = (CvRect*)cvGetSeqElem( faces2, j );
            recArray[i+j]=*r2;
       }
       CvSeq* faces22 = cvHaarDetectObjects( img_flip, cascade2, storage,1.1, 2,
CV_HAAR_DO_CANNY_PRUNING,cvSize(24, 24) );
      for(jj = 0; jj < (faces22 ? faces22->total : 0); jj++ )
      {      CvRect* r22 = (CvRect*)cvGetSeqElem( faces22, jj );
            CvRect* r22flip= r22;
             r22flip->x=img->width-r22->x-r22->width;
             r22flip->y=r22->y;
              r22flip->height=r22->height;
              r22flip->width=r22->width;
             recArray[i+j+jj]=*r22flip;
       }
       CvSeq* faces3 { /*omitted*/}
       CvSeq* faces4  {/*omitted*/}
      /*  CvSeq* faces5 {/*omitted*/} */

rec_num=i+j+jj+k+kk+l+ll+m;
for(int a=0; a<rec_num;a++)
{
      for(int b=a+1;b<rec_num;b++)
      {
            if(recArray[a].x<(recArray[b].x+recArray[b].width) &&
(recArray[a].x+recArray[a].width)> recArray[b].x  && recArray[a].y <
(recArray[b].y+ recArray[b].height)   && (recArray[a].y+ recArray[a].height)  >
recArray[b].y  )
            {
            int w,h,overlap,areaA,areaB;
```

```
w=min(recArray[a].x+recArray[a].width,recArray[b].x+recArray[b].width)-
max(recArray[a].x,recArray[b].x) ;
h=min(recArray[a].y+recArray[a].height,recArray[b].y+recArray[b].height)-
max(recArray[a].y,recArray[b].y);
            overlap=w*h;
            areaA=recArray[a].width*recArray[a].height;
            areaB=recArray[b].width*recArray[b].height;
            if(overlap>=areaA*0.5 || overlap>=areaB*0.5)
            {     recArray[a].x=0.5*(recArray[a].x+recArray[b].x);
                  recArray[a].y=0.5*(recArray[a].y+recArray[b].y);
                  recArray[a].width=0.5*(recArray[a].width+recArray[b].width);
                  recArray[a].height=0.5*(recArray[a].height+recArray[b].height);
                  recArray[b].x=0;
                  recArray[b].y=0;
                  recArray[b].width=0;
                  recArray[b].height=0;
      }}}}
             for(int e=0; e<100;e++)
             {      recNewArray[e].x=0;
                    recNewArray[e].y=0;
                    recNewArray[e].width=0;
                    recNewArray[e].height=0;
             }
            int c=0;
            for(int d=0; d<rec_num;d++)
            {     if(recArray[d].width!=0 && recArray[d].height!=0 )
                  {recNewArray[c]=recArray[d];
                  c=c+1;}
            }
            rec_num=c;     } }

void find_corner(IplImage* img,CvRect recarray[])
{
      for(int m=0; m<rec_num;m++)
      {
            int x,y,w,h;
            x=recarray[m].x+0.1*recarray[m].width;
            y=recarray[m].y+0.1*recarray[m].height;
            w=recarray[m].width-0.2*recarray[m].width;
            h=recarray[m].height-0.2*recarray[m].height;
            cvSetImageROI(img, cvRect(x, y, w, h));
            imgroi = cvCreateImage(cvGetSize(img),img->depth,img->nChannels);
            cvCopy(img, imgroi, NULL);
            cvResetImageROI(img);
            CvPoint2D32f corners[ MAX_CORNERS ] = {0};
            double quality_level = 0.1;
            double min_distance = 5;
            int eig_block_size = 3;
            int use_harris = 0;
            gray_image = cvCreateImage(cvGetSize(imgroi), IPL_DEPTH_8U, 1);
            cvCvtColor(imgroi, gray_image, CV_BGR2GRAY);
            eig_image = cvCreateImage(cvGetSize(imgroi), IPL_DEPTH_32F, 1);
            temp_image = cvCreateImage(cvGetSize(imgroi), IPL_DEPTH_32F, 1);
      cvGoodFeaturesToTrack(gray_image,eig_image,temp_image,corners,&corner_n
um,quality_level,min_distance,NULL,eig_block_size,use_harris);
            for( int n = 0; n < corner_num; n++)
            {     int radius = 3;
                  corners[n].x=corners[n].x +x;
                  corners[n].y=corners[n].y +y;
                  cvCircle(img,cvPointFrom32f (corners[n]),radius,CV_RGB(0,255,0),
1, 8, 0);
                  cornerArray[n+corner_num*m].x=corners[n].x;
                  cornerArray[n+corner_num*m].y=corners[n].y;
            }
      }
}

void moved_model(IplImage* img2)
{
      CvRect rec_move_temp[100]={0};
      int o=0;
      CvPoint2D32f p1,p2,p3,p4,pp;
      for( int p = 0; p < 400; p=p+4)
      {     p1=pre_cornerArray[p];
            p2=pre_cornerArray[p+1];
            p3=pre_cornerArray[p+2];
            p4=pre_cornerArray[p+3];
            pp.x=0.25*(p1.x+p2.x+p3.x+p4.x);
            pp.y=0.25*(p1.y+p2.y+p3.y+p4.y);
            pre_centerArray[o]=pp;
            o=o+1;}
```

```
    int r=0;
    CvPoint2D32f p5,p6,p7,p8,pp2;
    for( int q = 0; q < 400; q=q+4)
    {   p5=cornerArray[q];
        p6=cornerArray[q+1];
        p7=cornerArray[q+2];
        p8=cornerArray[q+3];
        pp2.x=0.25*(p5.x+p6.x+p7.x+p8.x);
        pp2.y=0.25*(p5.y+p6.y+p7.y+p8.y);
        centerArray[r]=pp2;
        r=r+1;}
    for(int v=0;v<100;v++)
    {   delta[v].x=centerArray[v].x-pre_centerArray[v].x;
        delta[v].y=centerArray[v].y-pre_centerArray[v].y;
    }
    int temp=0,temp2=0;
    for(int m=0; m<rec_num;m++)
    {   temp=int(ceil(delta[m].x));
        temp2=int(ceil(delta[m].y));
        rec_move_temp[m].x=recNewArray[m].x+temp;
        rec_move_temp[m].y=recNewArray[m].y+temp2;
        rec_move_temp[m].width=recNewArray[m].width;
        rec_move_temp[m].height=recNewArray[m].height;
    }
    int cc=0;
    for(int dd=0; dd<rec_num;dd++)
    {   if(rec_move_temp[dd].x + 0.5*rec_move_temp[dd].width <img2->width
&& rec_move_temp[dd].x+ 0.5*rec_move_temp[dd].width  >0 &&
rec_move_temp[dd].y + 0.5*rec_move_temp[dd].height<img2->height &&
rec_move_temp[dd].y + 0.5*rec_move_temp[dd].height>0 )
        {   rec_move[cc]=rec_move_temp[dd];
            cc=cc+1;}
    }
    pre_rec_num=cc;
}

//omitted
void draw(IplImage* img,CvRect recarray[]);
void draw_image(IplImage* img,CvRect recarray[]);
void find_corner_again(IplImage* img,CvRect recarray,int corner_num);
void track( IplImage* image,IplImage* image2,CvPoint2D32f corarray[] );
void new_model(IplImage* img2)
{
    CvPoint2D32f cornerArray_temp[400] = {0};
    int match[100]={0};
    int status[100]={0};
    int count[100]={0};
    for(int h=0;h<pre_rec_num;h++)
    {match[h]=0;}
    for(int j=0;j<rec_num;j++ )// current
    {
        int a=0;
        for(int i=0;i<pre_rec_num;i++)// previous
        {
            int num=0;
            if(  rec_move[i].x< (recNewArray[j].x+recNewArray[j].width)   &&
(rec_move[i].x+ rec_move[i].width )> recNewArray[j].x  &&
rec_move[i].y<(recNewArray[j].y+recNewArray[j].height) &&
(rec_move[i].y+rec_move[i].height)>recNewArray[j].y  )
            {   int w,h,overlap,areaA,areaB,w2,h2;
    w=min(rec_move[i].x+rec_move[i].width,recNewArray[j].x+recNewArray[j].wi
dth)-max(rec_move[i].x,recNewArray[j].x);
    h=min(rec_move[i].y+rec_move[i].height,recNewArray[j].y+recNewArray[j].he
ight)-max(rec_move[i].y,recNewArray[j].y);
    overlap=w*h;
    if(rec_move[i].x + rec_move[i].width <=img2->width && rec_move[i].x >=0
&& rec_move[i].y + rec_move[i].height<=img2->height && rec_move[i].y >=0 )
    {   w2=rec_move[i].width;
        h2=rec_move[i].height; }
        if(rec_move[i].x+rec_move[i].width> img2->width)//right
        {   w2=img2->width-rec_move[i].x; }
        if(rec_move[i].x<0)                                    //left
        {   w2=rec_move[i].width+rec_move[i].x;}
        if(rec_move[i].y+ rec_move[i].height>img2->height) //down
        {   h2=img2->height-rec_move[i].y;}
        if(rec_move[i].y >0)                                   //up
        {   h2=rec_move[i].height+rec_move[i].y;}
        areaA=w2*h2;
        areaB=recNewArray[j].width*recNewArray[j].height;
        if(overlap>areaA*0.3 || overlap>areaB*0.3)
```

```
            {     match[i]=1;
                  if(pre_status[i]>=3)
                  {     status[j]=3;
                        count[j]=pre_count[i];}
                  if(pre_status[i]<3 && pre_status[i]>=0)
                  {     status[j]=pre_status[i]+1;   // confidence ++
                        count[j]=pre_count[i];
                        if(status[j]==3 && pre_count[i]==0)
                        {     penguin_num++;
                              count[j]=pre_count[i]+1;}
                  }
                  if(pre_status[i]<0)
                  {
                  status[j]=0;
                  count[j]=pre_count[i];}
      recNewArray[j].x=(3*rec_move[i].x+7*recNewArray[j].x)/10;
      recNewArray[j].y=(3*rec_move[i].y+7*recNewArray[j].y)/10;
      recNewArray[j].width=(rec_move[i].width+4*recNewArray[j].width)/5;
      recNewArray[j].height=(rec_move[i].height+4*recNewArray[j].height)/5;
      for(int s=0;s<4;s++)
      {     if(cornerArray[i*4+s].x<recNewArray[j].x+recNewArray[j].width &&
cornerArray[i*4+s].x>recNewArray[j].x &&
cornerArray[i*4+s].y<recNewArray[j].y+recNewArray[j].height &&
cornerArray[i*4+s].y>recNewArray[j].y)
            {cornerArray_temp[j*4+num]=cornerArray[i*4+s];
            num++;}
      }
      if(num<4)
      {     find_corner_again(img2,recNewArray[j],4-num);
            for(int t=0;t<4-num;t++)
            {     cornerArray_temp[j*4+num+t]=corner_new[t];}
      }
      a++;
}}} //end for i
      if(a==0)
      {
            status[j]=0;
            count[j]=0;
            find_corner_again(img2,recNewArray[j],4);
            cornerArray_temp[j*4]=corner_new[0];
            cornerArray_temp[j*4+1]=corner_new[1];
            cornerArray_temp[j*4+2]=corner_new[2];
            cornerArray_temp[j*4+3]=corner_new[3];
            }
      } //end for j
      int u=0;
      for(int c=0;c<pre_rec_num;c++)
      {     int num2=0;
            if(match[c]==0)
            {     if(pre_status[c]>-5)
                  {     recNewArray[rec_num+u]=rec_move[c];
                        status[rec_num+u]=pre_status[c]-1; //confidence --
                        count[rec_num+u]=pre_count[c];
                        for(int f=0;f<4;f++)
                        {
                        if(cornerArray[c*4+f].x<(rec_move[c].x+rec_move[c].width)
&& cornerArray[c*4+f].x>rec_move[c].x &&
cornerArray[c*4+f].y<(rec_move[c].y+rec_move[c].height) &&
cornerArray[c*4+f].y>rec_move[c].y)
      {cornerArray_temp[4*(rec_num+u)+num2]=cornerArray[c*4+f];
                              num2++;}
                        }
                        if(num2<4)
                        {     find_corner_again(img2,rec_move[c],4-num2);
                              for(int g=0;g<4-num2;g++)
                  {cornerArray_temp[4*(rec_num+u)+num2+g]=corner_new[g]; }
                        }
                        u++;
                  }}}
      rec_num=rec_num+u;
      for(int y=0;y<100;y++)
      {     pre_status[y]=status[y];}
      for(int y=0;y<100;y++)
      {     pre_count[y]=count[y];}
       for(int z=0; z<400;z++)
      {    cornerArray[z]=cornerArray_temp[z];}
}
```