CHALMERS

Peocounter

People Counting Software

Tryggvi Björgvinsson

Master's Thesis - 2006

Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Gothenburg, 2006

All rights reserved. This publication is protected by law in accordance with “Lagen om Upphovsrätt, 1960:729”. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the authors.

© 2006
Tryggvi Björgvinsson

Abstract

Tallying people gives retailers knowledge about their visitors. How they can be manipulated by advertisements, weather, time of day et cetera. This is extremely valuable and helpful information which can be used to, for example, plan staffing and optimum times for cleaning and maintenance, determine opening hours and preferred advertisements, and so forth.

Peocon is a company which is built around people counting software. The application itself is based on analysing video streams from cameras in real time. Peocon specialises in interpreting visitor counts into valuable information for the customer. Without accurate people counting software the visitor data would be rather pointless.

This project originated around an idea to create embedded people counting devices, i.e. perform the counting directly on the cameras. The current scenario suffers from failures related to human interaction and by having the people counting software on the cameras themselves; human interaction will decrease drastically and the system cost will be reduced.

Relatively early on, it became clear that the software was far from being capable of being placed on the cameras. It is poorly designed with no discrete separation between functionality and user interface, relied on large proprietary libraries and the algorithms were inefficient. The project's goal therefore became to redesign the whole software and its algorithms to prepare it for various different usages.

The result is a platform independent library, with no user interface and devoid of any dependency on proprietary software. The software is now ready to be used in an embedded device or any surroundings for that matter.

Acknowledgments

I would like to express my deepest gratitude to the employees of Peocon. Especially *Timothy Bishop*, CTO, which has been extremely helpful and a source of valuable information and experience which he has been more than willing to share with me. Timothy never always took my comments with a smile on his face and either supported or gave counter arguments to new proposals I made. Timothy has also given himself time from his tight schedule to proof read this report which has sped up the process a lot.

Sigurjón Örn Sigurjónson and *Guðmundur Þór Guðmundsson* also deserve thanks for the work they put into testing the algorithms I devised for this project. They allowed the algorithms to be tested in a real retail environment.

Last, but not least, I want to thank *Einar Sigvaldason*, Peocon's CEO for giving me the opportunity to do this project and the freedom to take it into new directions.

Contents

1 Problem Description **1**
1.1 Project Scope 1
1.2 Current System 1
1.2.1 Solution 1
1.2.2 Shortcomings 2
1.3 Project Description 3

2 Analysis **5**
2.1 Background Subtraction 5
2.1.1 General 5
2.1.2 Approach 6
2.1.3 Compression 6
2.2 Blob Analysis 8
2.2.1 General 8
2.2.2 Approach 10
2.2.3 Data Structure 10
2.3 Blob Tracking 13
2.3.1 General 13
2.3.2 Approach 14
2.3.3 Statistics 15
2.3.4 Regions and Measurement Points 15

3 Design **19**
3.1 Background Subtraction 19
3.1.1 Structure 19
3.1.2 Algorithm 20
3.2 Blob Analysis 21

3.2.1 Structure 21

3.2.2 Algorithm 22

3.3 Blob Tracking 23

3.3.1 Structure 23

3.3.2 Algorithm 23

4 Implementation **25**

4.1 Program Structure 25

4.2 Classes and Methods 26

4.2.1 `analyser` 26

4.2.2 `analyser_set` 27

4.2.3 `background` 27

4.2.4 `background_set` 28

4.2.5 `blob` 28

4.2.6 `compression` 29

4.2.7 `counter` 29

4.2.8 `mp` 30

4.2.9 `region` 31

4.2.10 `setup` 31

4.2.11 `statistics` 32

4.2.12 `tracker` 32

4.2.13 `tracker_set` 33

5 Conclusions **35**

5.1 Results 35

5.1.1 Algorithms 35

5.1.2 Final Product 35

5.2 Problems 36

5.3 Future Work 37

5.4 Conclusion 37

6 Bibliography **39**

Appendices 41

A Algorithms 43

A.1 Background Subtraction . . . 43

A.2 Compression . . . 43

A.3 Blob Analysis . . . 44

A.4 Adding Pixel Run to a Neighbouring Blob . . . 44

A.5 Blob Tracking . . . 45

A.6 Find Probable Blob . . . 45

A.7 Dividing a Large Blob . . . 46

1 Problem Description

1.1 Project Scope

Counting visitors gives a retailer, with more than one store, the opportunity to improve sales at the lower performing stores and raise them to the level of the better stores. It enables the retailer to plan staffing within each day and around high seasons, e.g. sales or Christmas. The retailer is also able to determine opening hours and negotiate rent if positioned in shopping centres. Visitor counts also indicate effectiveness of marketing efforts such as advertising and ability and number of employees in the store.

Shopping centres can additionally use visitor counting to find optimal times for cleaning and maintenance, analyse flow of people through individual entrances, which for example show the preferred parking lots. In case of an emergency the shopping centre managers can react through real time customer counts inside the mall.

People counting software from Peocon ehf. is widely used in various countries. The main target group is retailers but it is also used in e.g. airports and convention centres. Peocon ehf., based in Iceland, was founded in 2000 and uses breakthrough computer-vision technology for tallying people in retail settings with extremely high accuracy.

Peocon ehf. has built its business around visitor counting. The technology is at the core of the company, but it specialises in interpreting the outcome and converting the data into valuable information.

1.2 Current System

1.2.1 Solution

The current system can be divided into two independent systems. One is positioned in the store and the second is at a remote location, typically the store's headquarters. These two systems communicate together by XML documents via a web server.

Visitors are counted in the store itself, via analog CCTV cameras positioned directly above the entrances. The cameras are connected to a frame grabber which sit in an ordinary workstation. This computer executes the Peocounter software which counts and analyses visitors going through each entrance. The results are then sent to the webserver which is located in the store's headquarters. Figure 1.1 shows a diagram of the stores system.

At the headquarters the web server receives the XML file from the store. A web service which is a part of the Peoweb application is on the web server and reads the XML file. It saves the visitor counts in a database, as well as retail data provided by the customer. The retail data can be saved in the database in various ways, depending on the customer's needs. At predefined intervals email messages are sent to corporate managers as well as the store managers. The email messages are automatic and contain information about both visitor counts and various

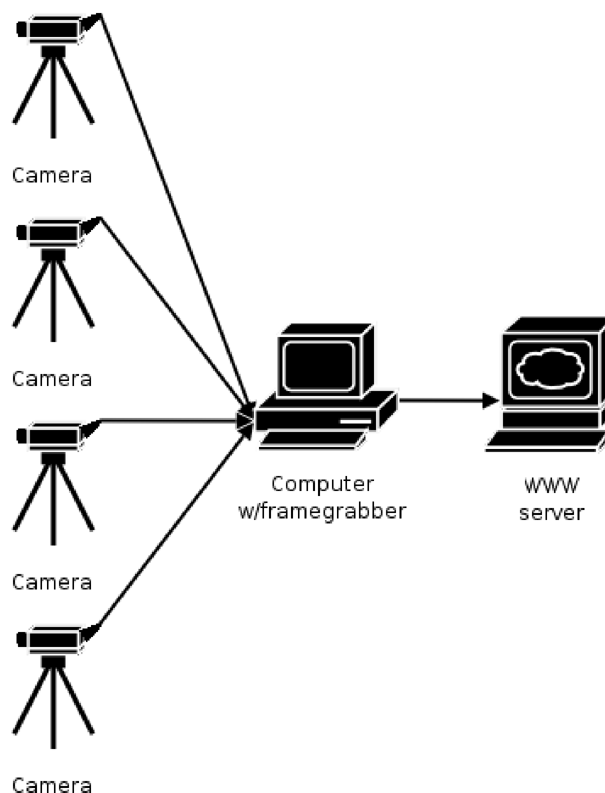

Figure 1.1: *Current system - Retail environment*

retail data. The Peoweb software is responsible for managing the email messages. This is the server side of the whole counting system and it is visualised in figure 1.2.

1.2.2 Shortcomings

The whole system has proved itself to be quite effective through the years. The server side is often tightly integrated into the customer's network and is designed to only be an extra program which runs on e.g. the same machine as the webserver. In this system there are not that many problems which occur as it only relies on network and servers provided by the customer and not controlled by Peocon ehf.

The computer system managed by Peocon ehf. is at the store itself. The physical design of this system is outdated. It contains a computer which both limits the resources as well as being a critical failure point. The computer can only contain one frame grabber, which is only able to serve four cameras at the same time. If something would happen to this computer no output from any camera would be analysed and nothing would be sent to the server.

Humans interfering with the computer and operating system faults are the most frequent causes of a system failure. Operating system crashes have occured which demand that the computer must be restarted. This means that the computer must be easily accessible so they can be restarted, which leads to another problem. Store employees which are not familiar with how the Peocounter system works tend to shut down the computer or unplug either network or power cable. Once in a while network connections are reconfigured without reconfiguring Peocounter. This results in failed network transmissions, which means the visitor counts will be delayed if ever sent.

Another limiting factor to this system is the frame grabber technology. The drivers for the

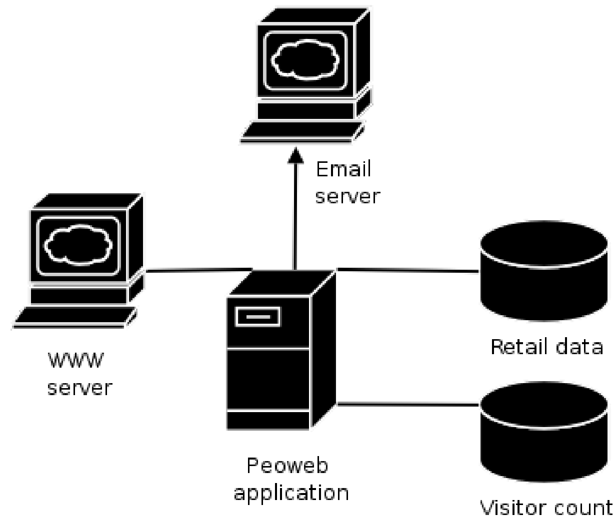

Figure 1.2: *Current system - Headquarters*

frame grabbers must be licensed from a company named Euresys s.a. The cost of a license for each frame grabber is very high which reduces the revenues for Peocon ehf. Another problem with the Euresys s.a. system is that it is proprietary and therefore binds some technological advances in the Peocounter system to the Euresys s.a. products.

Another disadvantage with the Euresys s.a. frame grabbers is the frame rate. The counting software needs minimum of 15 frames per second but higher is often needed, especially when the camera is not very high from the ground. The frame grabbers are only able to give a frame rate of maximum 15 frames per second, which extends the setup time to achieve minimum accuracy in some difficult circumstances.

1.3 Project Description

Peocon ehf. is looking at the option of swapping out the analog CCTV cameras for digital IP cameras. There are many factors vital to this decision. Most important factor is cost. The analog cameras are more expensive than the digital ones. The digital cameras also allow easier installation and more flexibility since the frame grabbers are replaced by a network connection, either wireless or ethernet. The network connection has two very important advantages. Firstly, it is cost effective and secondly, it removes the limitation set by the framegrabbers, such as cost of licenses and technological advances. Digital cameras also support higher frame rate. They can be configured and set to different frame rates with maximum of 30 frames per second. The digital cameras also give a better quality in the video since they use progressive scan while the analog cameras are interlaced.

Digital IP cameras allow Peocon ehf. to investigate if they are able to transform Peocounter into an embedded device and fit it on the camera itself. This will most probably improve the stability of the system as the computer which is the source of most of the failures, in the current system, can be left out. Figure 1.3 shows the schema of the embedded system in the store.

This system is a lot simpler to install and add cameras to if needed. Each camera is independent from one another and there is no bottleneck which limits the number of visitor counting areas, except for the network itself. Since only XML documents are sent at certain intervals this

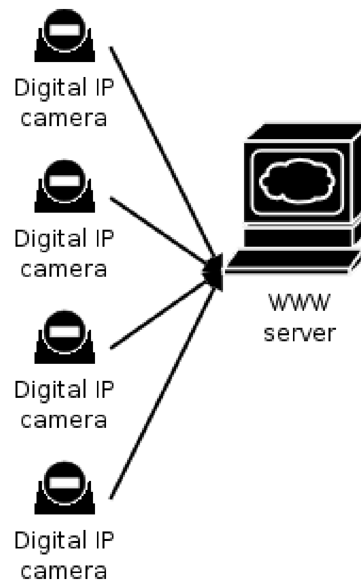

Figure 1.3: *Embedded system - Retail environment*

should not be noticable. This also reduces cost as the system does not include an expensive computer. The system may become completely free of operating system crashes and is not accessible to humans.

This project looks closer at how ready the existing Peocounter software is to become an embedded device. Algorithms for background subtraction, blob analysis and blob tracking will be redesigned if necessary in order to reduce the size and complexity of the software.

The code will be rewritten and ported to C++. This will make the code compatible with more platforms and computer architectures. The code will also be redesigned to become more object orientated. This will make it easier to manage and change as well as more adaptable to different systems.

It is not the goal of this project to make an embedded device but to prepare the software for any future decisions, be it an embedded system or something else.

2 Analysis

2.1 Background Subtraction

2.1.1 General

Using regular surveillance cameras to count visitors introduces some consequential problems. The video feed acquired from the cameras show an exact image of the environment and circumstances. For example, infrared cameras do not give rise to the same problem since anything but heat sources (such as human beings) is filtered out. Because of this all images must be processed before being analysed. Processing the image involves removing all excessive data from the image such as background and noise. After processing the image one will have an image which clearly shows only the foreground of the image.

Figure 2.1: *Person walking underneath a camera*

Take, for example, the image shown in figure 2.1. This image depicts a customer walking into a store. In order to extract the foreground, i.e. the person, in the image the background, i.e. floor and stationary objects must be known. The background image is shown in figure 2.2.

Figure 2.2: *Background image*

When the background is known one can use background subtraction algorithms to remove the background from the image. The algorithms can also be tweaked to exclude some sorts of foreground objects, such as shadows, from the image. If needed one can add noise filters

on top of the images. Figure 2.3 shows the output image when the background, shown in figure 2.2, has been subtracted from figure 2.1. The image shows the foreground objects, often referred to as blobs. In this image there is only one blob showing the person which was walking underneath the camera.

Figure 2.3: *Foreground image*

In visitor counting the foreground image, or blob image, shows any objects which move through the frame. In order to count one has to analyse the blobs and track each individual through the frame. It is therefore vital that the background subtraction algorithms return as accurate blob images a possible, so that all steps of the visitor counting are performed correctly.

2.1.2 Approach

Work is being done by other employees of Peocon ehf., on improving the background subtraction algorithm. The current algorithm works for most counting environments but can be improved by e.g. adding sunblock,which removes shadows and better noise removal. Decision was made not to try to redesign the algorithm, only adapt the output of the algorithm to the needs of this project. This approach prevents duplicated work and allows the design of the improved algorithm to be independent of this project.

The existing algorithm is written in Delphi and will be ported to a more appropriate programming language so that the background subtraction can be redesigned to become more object orientated. This will allow faster integration of the upcoming subtraction algorithm. It will also make it easier to manipulate the output of the background subtraction algorithm.

2.1.3 Compression

For the hybrid system, where images are preprocessed on the camera before being sent to a computer where the images are analysed and visitors counted, it was decided to perform all placement specific computations on the camera and analysis, tracking and counting on the computer. The output of the camera based computations are sent via a network up to 30 times per second and therefore fast and effective compression is needed.

Compressing the image also creates the possibility of designing faster analysis algorithms. In the current system the algorithms work pixel wise but by using compression one may be able to analyse many more pixels at the same time, even the whole picture.

The compression method needed must be very fast and it should as well be able to compress on the fly, i.e. when each pixel has been subtracted from the background it should be able to be compressed right away. The compression should be lossless. Another important factor is that the image is a black and white one, which in most cases is completely white.

With this in mind, one can exclude most compression techniques. Code redundancy is not a problem, so compression methods such as Huffman, arithmetic and variable-length coding are not needed. Gray-level encoding, such as LZW or Bit-plane coding is not needed either.

Constant area or run length coding are the most prominent methods of compression. Some quick tests were made to estimate how many frames are completely white, i.e. no foreground objects (humans passing under the camera). Short sequence taken from Leifsstöð, the Icelandic airport, shows that about 30% of the frames are totally white, and this includes people standing directly under the camera for a few minutes. Often there are only one or two persons passing under the camera and these persons only take about 2% of the frame each. The same test made on Smáralind, Iceland's biggest shopping centre, shows that the image shows only the background 98% of the time. The cameras in Smáralind are placed lower than in Leifsstöð which means the persons take up more space (about 4% of the image), but they pass the camera in a shorter amount of time. The rough results of these test show that the wisest compression method to use is most likely run length coding.

Run length coding is a very simple but effective compression which has been around since 1950. A black and white image is coded by counting the number of times the same pixel value appears in a row. To illustrate the the coding procedure figure 2.4 will be coded.

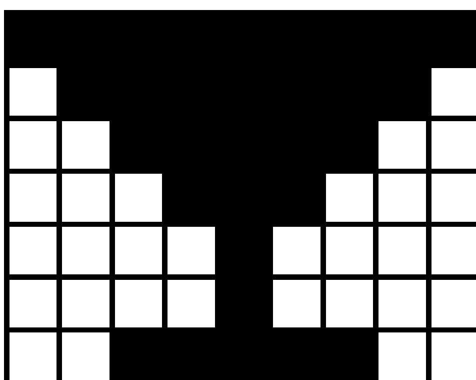

Figure 2.4: *Black/White image* - 9×7 *pixels*

The image is nine pixels wide and its height is seven pixels. Two different variants of run length coding are commonly used. One specifies the initial pixel value of each row while the other assumes the inital value (usually white). The difference is barely noticable. Images where each line, in most cases, starts with the same pixel value, i.e black handwriting on white paper, the second coding is preferred. Figure 2.4 encoded with the initial value specified at the beginning of each line becomes (black = 0, white = 1),

```
0,9,1,1,7,1,1,2,5,2,1,3,3,3,1,4,1,4,1,4,1,4,1,2,5,2.
```

Since most lines begin with a white pixel it would be better to use the other encoding method which assumes each line begins with a white pixel. The output of the run length coding would then be,

```
0,9,1,7,1,2,5,2,3,3,3,4,1,4,4,1,4,2,5,2.
```

One can see that the former technique's compression ratio, C_R, is 2.42. In the latter, which assumes the inital pixel value to be white, C_R is 3.15. As one can see, this is not a significant difference but in the case where one pixel value is more probable than another, the latter is preferred.

2.2 Blob Analysis

2.2.1 General

Blob analysis involves examining the blob image and find each individual foreground object. This may in some special cases become a rather difficult task, even for the human eye. Figure 2.5 shows two blobs which are easily distinguished from one another.

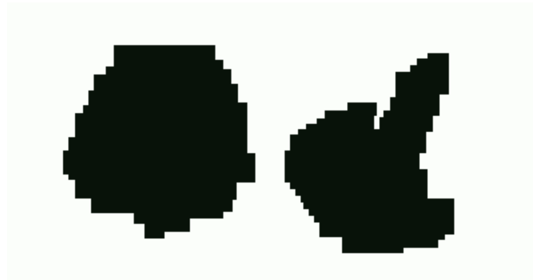

Figure 2.5: *Two classic cases of blobs*

When the moving persons have similar colours as the background the blobs become rather distorted. Holes in the middle of the blob can appear, similar to what is shown in figure 2.6.

Figure 2.6: *Hole appears within blob*

In some cases the edges of the blob become unclear and often certain parts of the blob appear as tiny scratches. These will often be regarded as noise. A distorted blob, with unclean edges, is shown in figure 2.7.

Figure 2.7: *Unclean edges of a blob make it harder to distinguish*

Blobs are found by finding all adjacent pixels and putting them together so that the pixels form the blob object. Analysing contiguous blobs is a fairly straightforward process. Fragmented

blobs, like the one shown in figure 2.8, are harder to analyse since it difficult to see if the blob is in fact many persons. It becomes impossible to do in crowded areas.

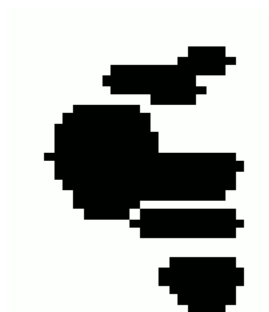

Figure 2.8: *One large fragmented blob can appear as smaller individual blobs*

People standing in groups are hard to deal with, even for the human eye. When they stand close together they form one big blob, like the one shown in figure 2.9. It is hard to determine how many blobs there actually are in the image. Figure 2.10 shows how the foreground objects can be divided into six blobs.

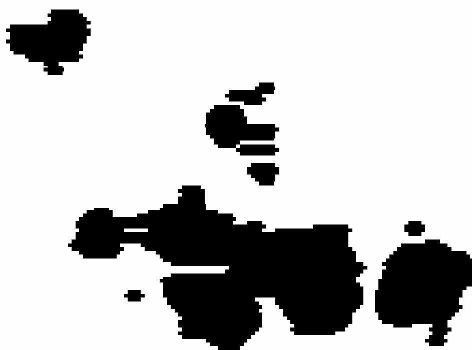

Figure 2.9: *Multiple blobs form a larger blob*

It may seem relatively straightforward to divide the larger blobs into smaller blobs by splitting the blobs with respect to height and width. This is not as easy as one may think, especially in retail environment. People coming from grocery stores or other similar shops often push trolleys ahead of them. These trolleys usually appear to be of the same size as a human being, like the one shown in figure 2.11. This makes splitting blobs by height or width a hard task which must take movement into account. Two blobs moving horizontally appear larger in height, while moving vertically have larger width. Movement does not come into the equation until in the blob tracking phase.

Blob analysis is all about identifying the blobs, grouping together adjacent pixels and representing them using a good data structure. The size of the blob is not the issue. Even though the blob consists of numerous people it will still be represented as one blob. This can give rise to other problems such as with the fragmented blobs which will make one blob appear as many. If the background subtraction is configured properly it should be relatively rare that a blob will be totally fragmented. Therefore all indvidual blobs, consisting of more persons or a part of a larger blob will be represented in an appropriate data structure. The name of this phase may be confusing, it does not involve analysing blobs, but analysing the pixels in the input image and finding all of the blobs.

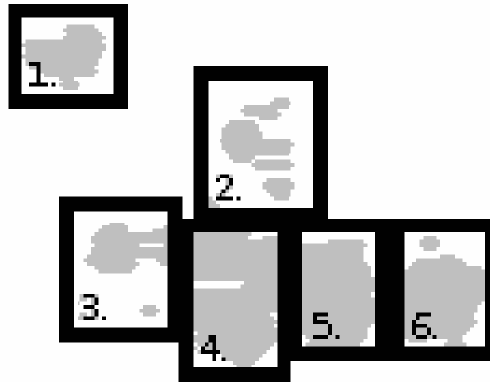

Figure 2.10: *Larger blob divided into smaller blobs*

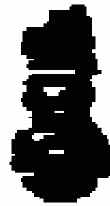

Figure 2.11: *Human with a trolley can make the blob look like two persons*

2.2.2 Approach

Up until this project Peocon ehf. has been using a component from Euresys s.a. which handles blob analysis. This solution means increased expenses for licenses and no control over any technological advances in blob analysis. The Euresys component will therefore be completely removed, making the computer software independent of any proprietary software.

The algorithm for the blob analysis must be written completely from scratch. This means that the programming language used for background subtraction can be used here as well. This makes the program more easy to follow or make additions to. It will give it a more solid base and the company will have a better overview of their software.

Like with the background subtraction, fast but effective algorithms are needed to separate the foreground objects from each other. Likewise is a suitable data structure needed to hold the objects so that they are easy to manipulate and can quickly be compared to foreground objects from the previous frame in order to track them.

2.2.3 Data Structure

Finding the appropriate data structure to hold the blobs was a relatively easy task. The first step is to assess all properties of the blobs and find the best way to represent the blobs.

A relatively simple but expandable way of gathering and holding information regarding the blobs is to use the new object orientated structure and create a blob class. This class will keep track of blob statistics as well as methods for adding pixels to the blob and merging it with other blobs. The best way to begin is to look at the various properties which are needed. One must bear in mind that the properties must make it easy to compare the blobs to one another.

The method of comparing must be quick and effective, so the properties used must be simple.

Obvious properties of a blob are its position, width and height. A good way to combine position with width and height is to keep information about the coordinates of the smallest X and Y value, as well as the largest X and Y values. In this way one can easily know the position of the blob, even with mirrored Y axis. One can find the width by subtracting the smallest X coordinate from the largest and in the same way the height by using the Y coordinates. Changes when one adds a new row to the blob and affect the coordinates are easy to manage, only by comparing the smallest or largest values. The limitation of this approach is that the blob's shape will not be known only a rectangular area surrounding it. This is shown in figure 2.12. The point of origin for the blob is the smallest X coordinate combined with the smallest Y coordinate.

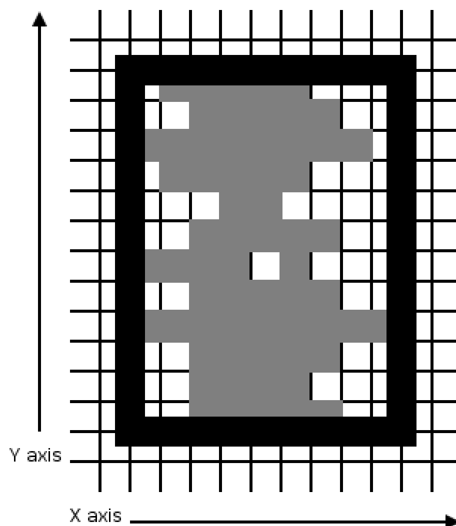

Figure 2.12: *A blob is recognised by a rectangular frame which surrounds it*

The algorithm scans through the blob image and adds rows of pixels to existing or new blobs. This is usually very simple, find a blob which connects to the row of pixels and then add these pixels to that blob. If no blob is found create a new one. To achieve this, it is sufficient to only keep information about the blob's last pixel row, i.e. the first pixel in that row and the length of that pixel run. There are two cases which are special and need additional processing. The former is when the pixel run is split in two, like shown in figure 2.13. The black pixel run is the most recently added one and the pixel run on its left is the one added before it. As one can see it is necessary to keep track of second last row so that that split runs like these can be assigned to the same blob.

The latter special case is the reverse of the former. It concerns the scenario when two blobs which have been distinct are joined together by a pixel run. The two blobs must be merged together to form one larger blob. This is noticable when one is trying to find a blob which is connected to a pixel run and two blobs are found. Therefore it is necessary to keep on iterating through the list of blobs until two blobs are found or the end of the list is reached. A pixel run connecting two distinct blobs is visualised in figure 2.14.

When blobs can easily be recognised one needs some sort of a data structure to hold all of the blob instances. Trying to organise them so that one will quickly match the blobs from older frames to newer ones is a good idea. To do this in a fast and effective way one could use some advanced structure like a skiplist to categorise them by, for example area or direction. The overhead of this method and the varying sizes of blobs on each location make this an ineffective way to compare blobs quickly. The fastest way to process the blobs is to simply

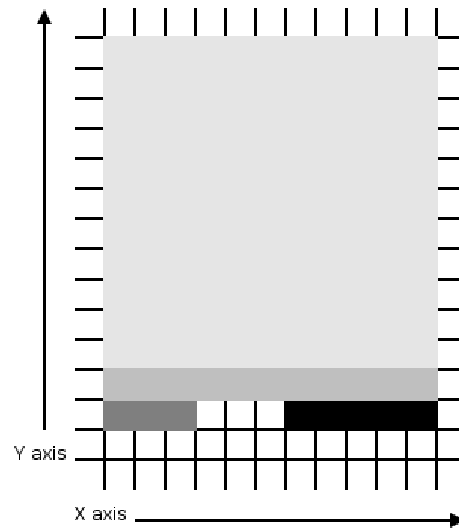

Figure 2.13: *Rows added to blobs. The darker the rows, the more recently added*

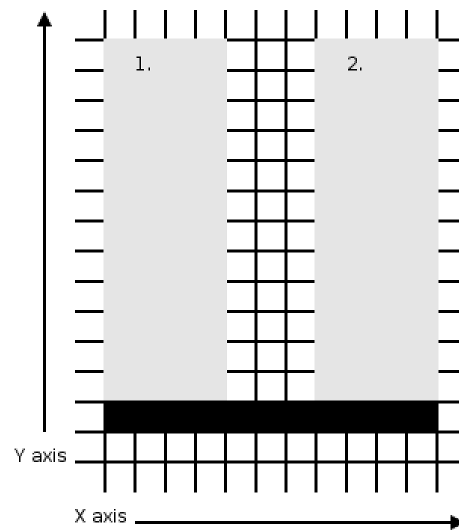

Figure 2.14: *Two blobs (1. and 2.) connected by the black pixel run are merged together to form a larger blob*

put them into a list as they appear. By doing this one is actually categorising them by their X and Y coordinates since the blob image is systematically processed starting at coordinates (0,0). What one needs then is a list which is easily manageable so that one can quickly merge blobs and add new ones. Since the amount of blobs varies with time it is smart to have an expandable list. With these reasons in mind a linked list was chosen to keep track of all the blobs.

2.3 Blob Tracking

2.3.1 General

The last step in order to count persons moving underneath the camera is to see whether a blob comes through an entrance and disappears in an exit area, if so the in count should be increased. If the blob travels through the image the other way around, the out count should be increased. Some blobs only move under the camera and never enter or exit the store. These are in most cases not counted.

The program is not required to hold track of the blob's exact path through the image, only the areas where it appeared and disappeared. This still requires the program to track the blobs from the time they appear until they disappear from the camera's view. Pairing the blobs in a frame with blobs in a previous frame will indicate the blobs movement. This is done using probability which can always result in incorrect tracking, but using correct techniuqes may minimise the chances of mixing up the blobs.

Figure 2.15: *Blob image of persons walking underneath the camera*

Figure 2.15 show three persons detected underneath the camera. These persons are moving in separate directions. The next frame, shown in figure 2.16 shows how the blobs have slightly changed their positions. The faded blobs are from the previous frame while the black blobs show the blob's new position. It can be seen that these blobs are all moving in different directions.

Knowing the blob's direction can give some valuable information. For instance, the middle blob in figure 2.16 appears to be pushing a small trolley. If the blob would be moving from left to right (or vice versa) it would most certainly consist of two people, indicated by its width.

The path of each blob is shown in figure 2.17. By tracking the blobs one will get an idea of their path. If we were to count people going through a large entrance at the top of the image,

Figure 2.16: *Movement of blobs between frames*

knowing the path of the blobs will show that the middle blob can be counted as entering, the rightmost blob as exiting and the lefmost blob should not be counted. Analysing the path of the blobs is the goal of blob tracking.

Figure 2.17: *Movement of individual blobs*

2.3.2 Approach

To be able to count a blob passing through an area of the image it is essential to know where a blob appeared and where it disappeared. This way one will know whether it is entering, exiting or only moving between two points and should not be counted.

The current version of Peocounter uses confusing settings to mark the counting area and its entrance and exit. Instead of trying to adapt the counting in the new version to this configuration, a more straightforward means to counting and analysing will be designed. This newer configuration must be built as similar to the older version as possible without limiting the expandability of the program. This will be done by introducing regions and measurement points. Hopefully will this new configuration enable Peocounter to perform as a flow analyser as well.

The algorithm must be designed to use the new blob list, i.e. the one that the blob analyser outputs. It should be of no surprise that it must be fast but still effective. That is, pairing of the blobs and checking them against both entrances and exits should be a quick operation.

2.3.3 Statistics

The best way to pair blobs quickly is to look whether a blob in the new list overlaps a blob in the older list. Two blobs which do not overlap are probably not related. This fact can be backed up by analysing the depth of persons and how it translates to pixels.

A person's depth must be at least 10 pixels so that it can be regarded as a blob. The depth of a person is affected by the limbs. A person standing straight with arms on the side is about 0.5 meters. If the person is walking or running it may go up to 2 meters.

An image which is 240x320 pixels can therefore cover a ground area of 12 by 16 meters (when 10 pixels equal 0.5 meters). This means that a running person takes roughly 40 pixels. The camera allows up to 30 frames per second which means that a person, with no outstretched limbs can pass through the image in about a second if the newer blob always overlaps the older one.

16 meters in a second results in the maximum speed of a person being about 57.6 kilometers per hour. The average speed of a person jogging is about 10 kilometers per hour so that these limits fall well under the maximum speed.

Based on the previous calculation blobs which overlap each other can be paired and others are discarded. If two blobs overlap one another, a statistic which can separate the blobs must be introduced. Distance travelled between blobs is a rather obvious choice. The blob which has moved the shortest Euclidean distance from the original one is more likely to be the same person.

One could use more statistics like for example change in area of the blob or its direction, but since the algorithm will have to be fast it becomes necessary to limit the comparison of the blobs to only a few techniques.

The idea of comparing the change in direction sprouted off another idea. The blobs of two persons holding hands, e.g. a couple, may appear as a wider single blob. This can also happen in crowded areas as is shown in figure 2.9. Given the maximum width of a blob one could divide a single blob representing a couple, into two separate blobs. This task is not an easy one since the persons can move in literally any direction and these persons can be pushing a trolley in front of them, as shown in figure 2.11. One wants to avoid counting the trolley as a person. By analysing the direction of the blob one can see whether or not the blob is wide, i.e. couple holding hands etc., or high, that is pushing a trolley.

It is unthinkable to correctly track all blobs correctly. There are always some special cases which cannot be accounted for. These do not occur very often as most people only walk in and out through entrances of shopping centres and stores. The statistics used should cover the most basic cases of blobs and track most blobs correctly through the image. The statistics and methods used have been carefully chosen with respect to real time analysis and limits of memory in embedded systems.

2.3.4 Regions and Measurement Points

Marking entrances and exits in an image require human interaction. In the current version of Peocounter the user interface leaves room for human errors which could prevent the system counting. To begin with one must define a flow area. This area is a rectangle which indicates the floor area where people move underneath the camera. Then one continues to mark the entrances and exits of the flow area. These are as well rectangles but three of each rectangle's edges must lie outside of the flow area. This configuration is shown in figure 2.18.

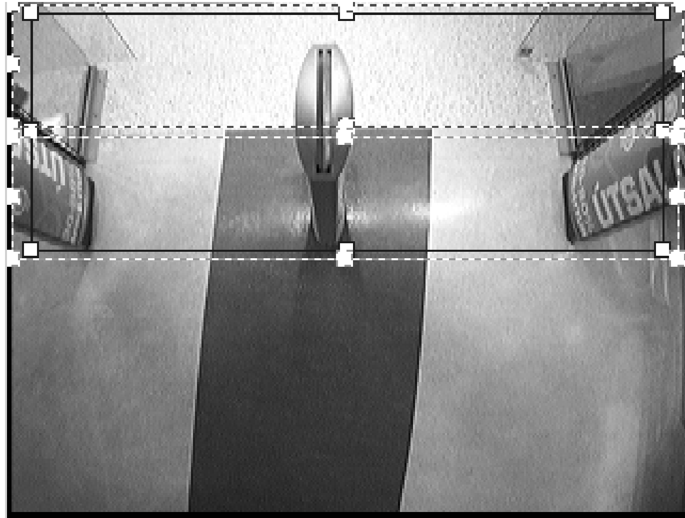

Figure 2.18: *Flow area defined in current version*

These confusing settings are not the only limiting factor of the current version of Peocounter. The current version does not allow multiple flow areas per image. Take, for example the background shown in figure 2.19 in which there is one entrance, i.e. the doorway. Let us, for the sake of explanation, say that a client would be interested in seeing how many of the store's visitors go left and how many go right, i.e. this is a shared entrance to two stores. In the current system it would not be possible. This limits what the users of Peocounter can do.

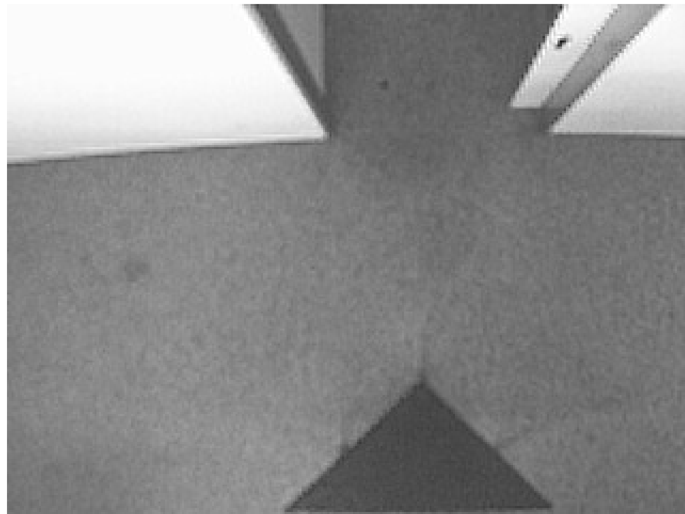

Figure 2.19: *Counting scenario with two flow areas*

To make the flow areas more expandable and not as limiting nor confusing a new schema has been designed. In this schema one marks possible entrances and exits as rectangles, named regions. It is not necessary to mark them as entrances or exits. Each region can be both an entrance as well as an exit.

In order to count persons going left and right in figure 2.19 one can define three regions. The first region would be the doorway (A), the second the area (B), will be the one to the people's right, and the third (C) to their left. These regions are shown in figure 2.20

Using these regions one continues to define measurement points (MP). These will indicate what regions should be set as entrances and which region will be marked as an exit. Measurement

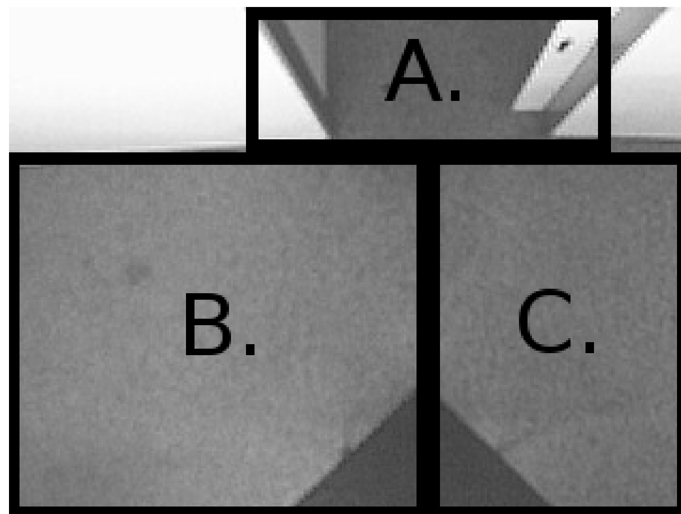

Figure 2.20: *Regions defined for figure 2.19*

points therefore define relations between two regions. In the example where one wants to see how many persons go to the left and how many to the right one would create two measurement points. One from region A to B (people going to the right), and the other from A to C (people going to the left). If one is interested to see the flow between the two stores one could define a third measurement point to count the flow between B and C. The measurement points are visualised in figure 2.21.

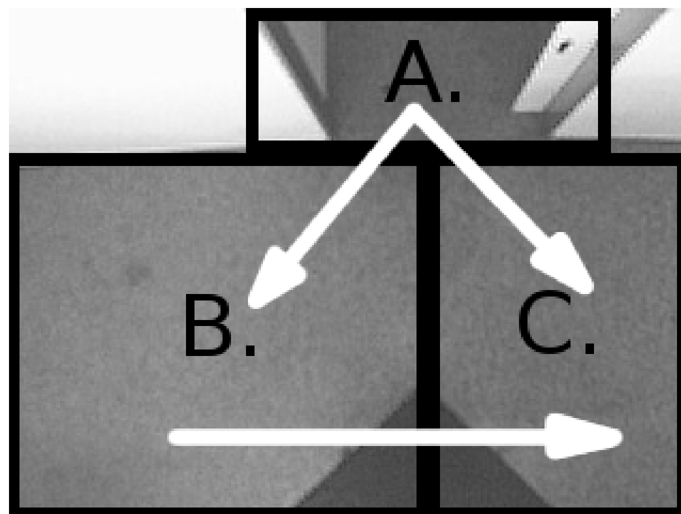

Figure 2.21: *Measure defined between the regions in figure 2.20*

This configuration is similar to the one used in the current version, but allows more expandability, multiple flow areas, reuse of regions in different flow areas. This configuration is also more manageable if one uses simple text files for configuration of Peocounter.

3 Design

3.1 Background Subtraction

3.1.1 Structure

One of the goals with this project is to make the program more object orientated and by doing so making it easier to understand, modify and develop further. The design of the background subtraction will form the design of all remaining components since by using the same design for all remaining components, system development is simplified. Figure 3.1 shows the sequence diagram for the background subtraction.

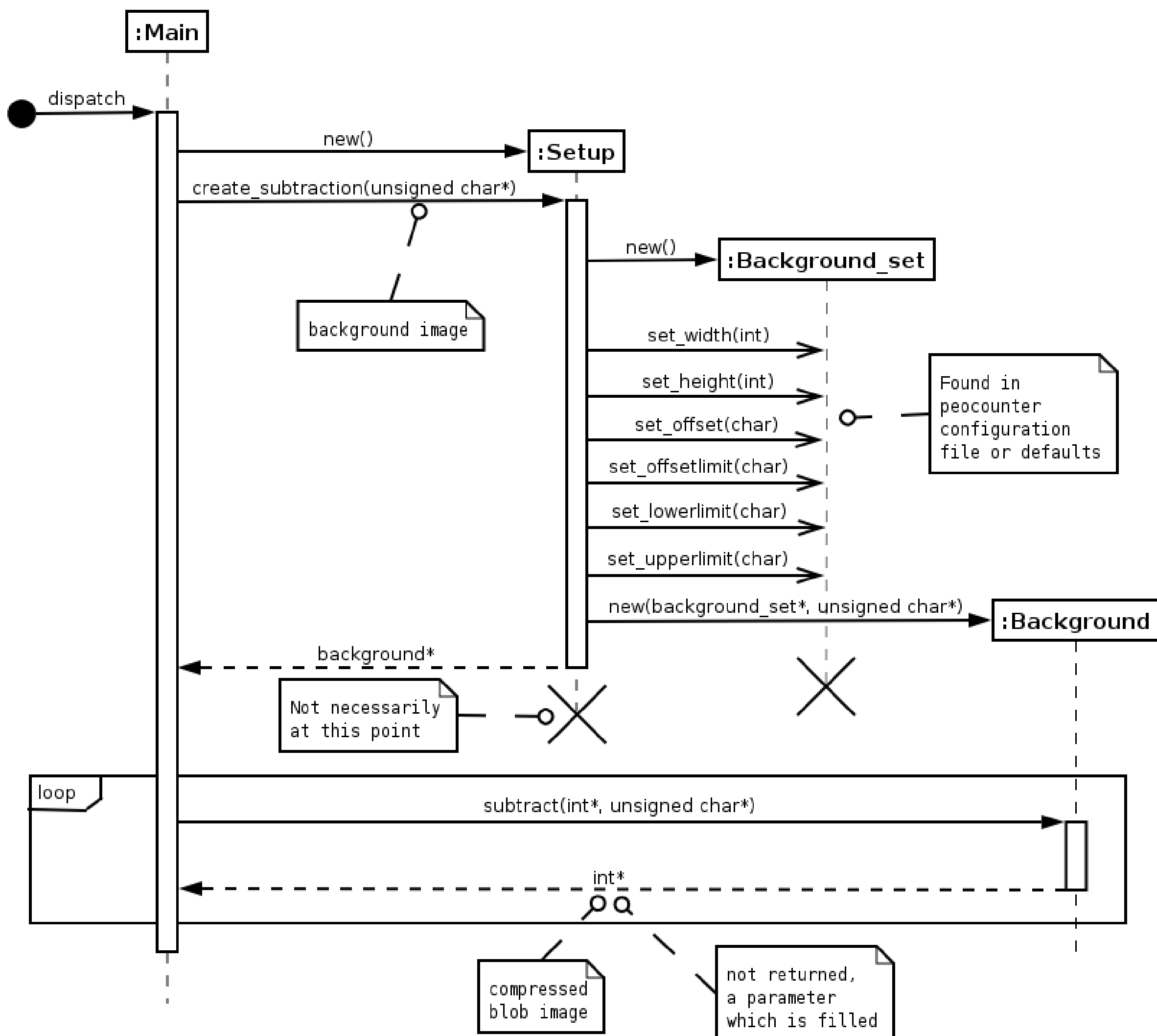

Figure 3.1: *Sequence diagram of the background subtraction*

A `Setup` class will read a configuration file and create a `Background` object via a settings

class, `Background_set`. Using the `Background` object one can quickly and easily start doing background subtraction by calling the function `subtract()`. This function has two parameters, an array to hold the compressed image containing the foreground objects (blob image) and the image frame from which the background should be subtracted.

In order to use the background subtraction component one needs to call two functions. Firstly, `create_subtraction()`, which takes a snapshot of the background as a parameter and returns the `background` object. Then one can call the `subtract` method of the `background` object and acquire the compressed blob image. The user does not need to interact with the background subtraction in any other way.

3.1.2 Algorithm

The background subtraction algorithm is a very simple but fast method of computing whether a pixel in the input image is a foreground object or if it is a part of the background. The algorithm firstly checks each pixel value in the background image and from that level an upper and lower limit for the pixel value in the input image is found using two parameters. A percentage either added to or subtracted from the background pixel value for the upper or lower limits, respectively. That is, the upper limit is computed using an upper percentage value, p_u, using the following equation where b is the pixel value of the background.

$$U(x) = (1 + p_u) \times b \tag{3.1}$$

In the same way one uses a lower percentage value, p_l to compute the lower value using a similar equation.

$$L(x) = (1 - p_l) \times b \tag{3.2}$$

Figure 3.2 shows how the upper and lower limit grow larger as the background pixel value gets higher. Any value not within these allowed intervals will be marked as a part of a foreground object.

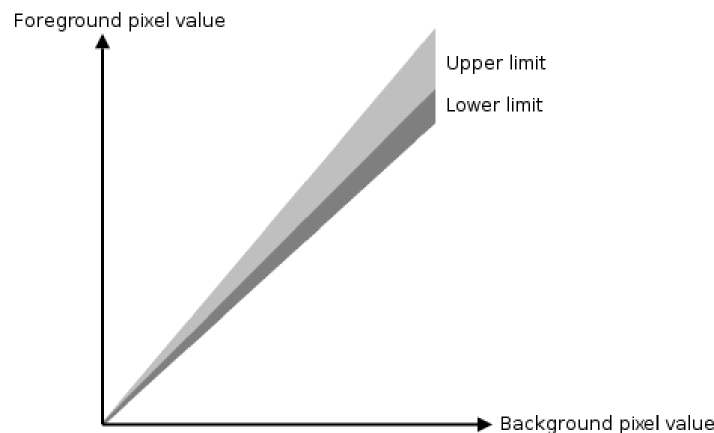

Figure 3.2: *Lower and upper limit of foreground pixel value*

An addition to this algorithm is a rather simplistic pattern matching. This pattern matching is effective when comparing different textures. It may well be that a person clothed in similar colours as the background passes under the camera, to still notice this person it is good to have some sort of texture analysis. This could be done using some advanced techniques like,

for example, by analysing spectrums but since visitors are counted in real time such analysis is too time consuming.

What is done instead is that difference between two close pixels in the background image is compared to the difference of the same pixels in the input image. High difference means texture change, i.e. a foreground object with similar colours as the background but another texture.

Pseudo code of the background subtraction algorithm can be found in appendix A.1 and the compression algorithm is shown in appendix A.2.

3.2 Blob Analysis

3.2.1 Structure

The structure of this part of the software is designed to be similar to the background subtraction bit. Figure 3.3 shows the sequence diagram of the blob analysis.

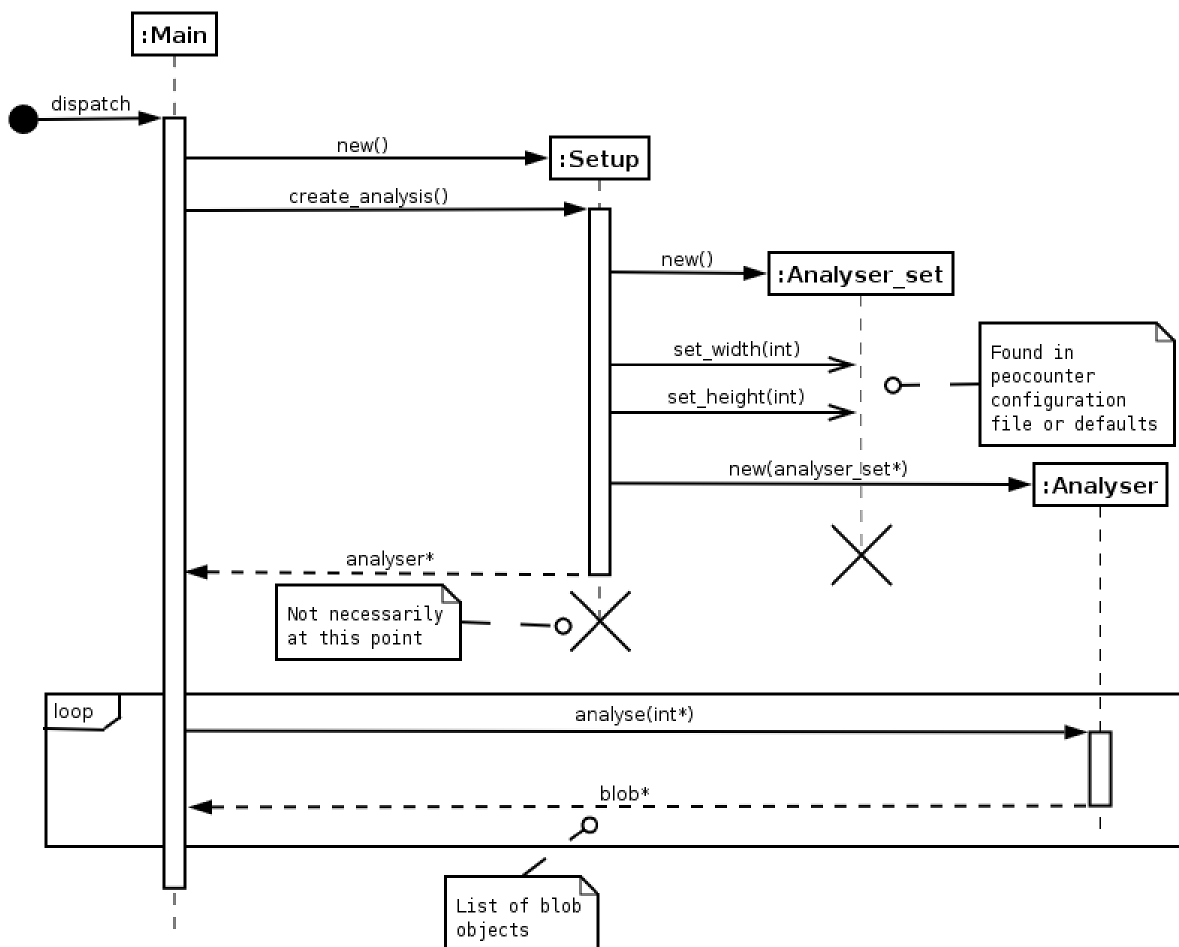

Figure 3.3: *Sequence diagram of the blob analysis*

As one can see the sequence diagram is essentially the same as in the background subtraction (figure 3.1). The same setup class can be used to create the analyser class. This is done by calling the method `create_analysis()` which resembles the `create_subtraction()` method in background subtraction. However, in this case an instance of the `Analyser` class is returned.

For the regular user of this software it should normally be as easy to use as the background subtraction. The only thing required is to call the function `analyse()` which will return a data structure containing the foreground objects, even called blobs. Of course all classes, like for example `Analyser_set`, are available to use for some special cases.

3.2.2 Algorithm

The algorithms works on the run length coded image. It jumps over all white pixels and processes only the black ones. When it finds a black pixel run it starts by computing the X and Y coordinates. When the coordinates are known it looks at the length of the run to find out if it stretches over the image boundaries, i.e. there are two blobs on the same row, one on the right edge of the image and the other on the left one. Figure 3.4 shows how two separate blobs on each side. The compressed image for this frame is

```
7,4,5,4,5,4,5,4,25.
```

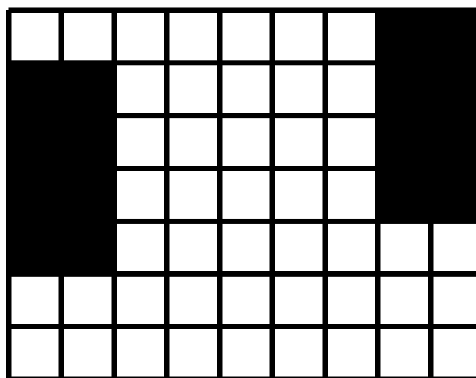

Figure 3.4: *Separate blobs on each side of the image are joined in the compressed image*

In this case the blobs must be split up. Which is done using modulus calculations to divide the blobs and then add them separately to the list of blobs. After processing the black pixel run the algorithm jumps over the white pixels and repeats the procedure for the next black pixel run. The algorithm which processes the black pixel runs is shown in appendix A.3.

Adding pixels to an adjacent blob or a new one is done using two methods. One of them is a very simple one which tries to add the pixels to a neigbouring blob by calling the other method and if it does not succeed it creates a new blob and adds it to the back of the list.

Adding a pixel run to neighbouring blobs is slightly more complex. The algorithm iterates through the list of blobs and checks if the pixel run is adjacent to any blob. This is done by checking both the last or second last row of the blob within a method in the blob instance. The method looks at the row number of the pixel run, if it is the same as the blob's last row number it compares with the second last row. If, however, the row number is next to the blob's last row number it compares to the last row. If the row number is neither of these the blob cannot be connected to the pixel run.

If the pixel run is connected to the blob the algorithm proceeds to see if another connecting blob was found previously. If not it adds the pixels to the blob but if found it merges the the blob to the one previously found. This algorithm is shown in appendix A.4

If an adjacent blob is found it is returned and if not a zero is returned.

3.3 Blob Tracking

3.3.1 Structure

The sequence diagram for the blob tracking (figure 3.5) is similar to the previous ones. The `setup` class reads the configurations and creates an instance of a tracker settings class, `tracker_set`. This class holds the maximum width of a person, the regions of interest and a `counter` object. This `counter` object per se, holds all of the measurement points (mp).

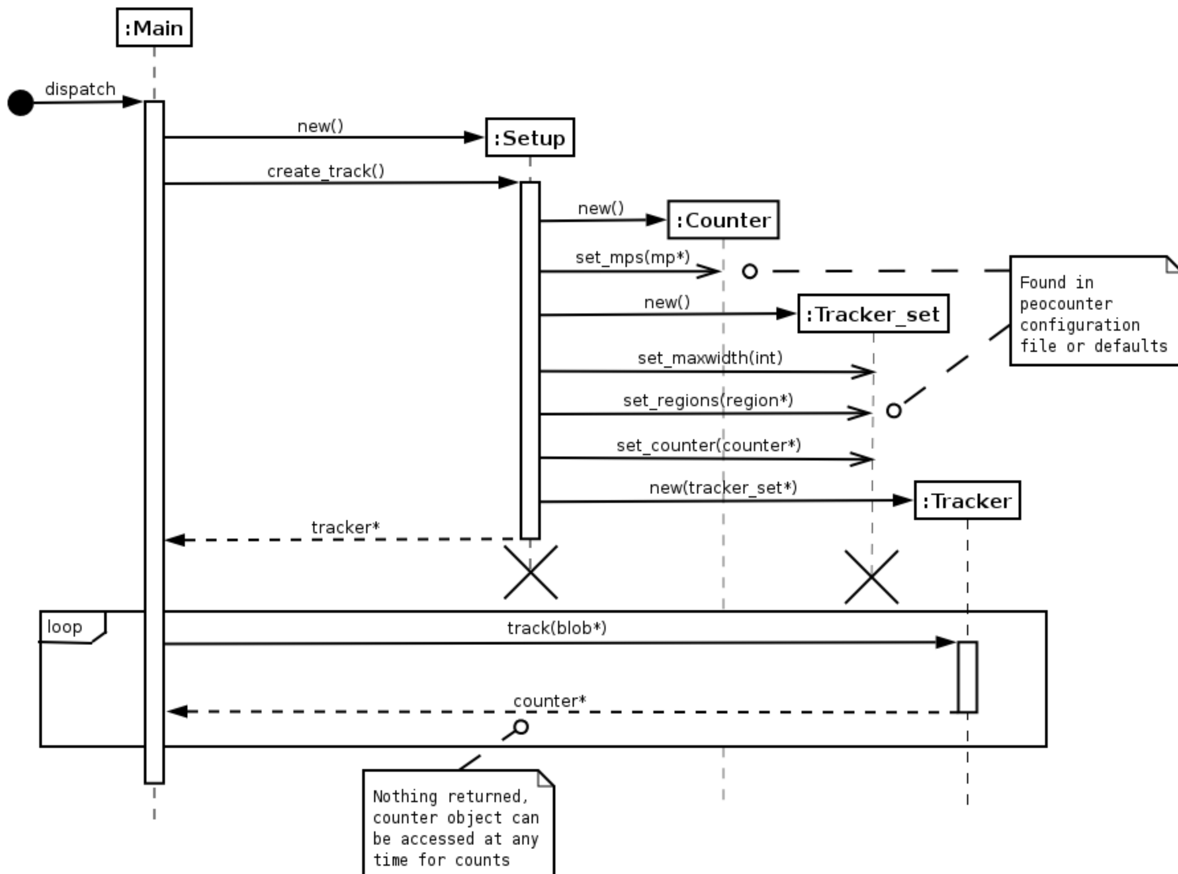

Figure 3.5: *Sequence diagram for blob tracking*

The instance of the tracker settings are then used to create the `tracker`. After its creation one can destroy both the instance of the `setup` class and the tracker settings since both of them are of no use anymore. The `counter` instance cannot be destroyed since nothing is returned when one calls the method `track()` of the `tracker` class. To get the desired in and out counts of the measurement points one can retrieve the list from the `counter` object.

3.3.2 Algorithm

The simplest and best way to count blobs is to track the movements of them through the image. Marking where it appeared and then disappeared and then see if it is to be counted as a person entering, exiting or just passing through. When a blob appears its region is saved in variable, `origin`, which is contained in the `blob` class.

In order to do so one must pair a blob from an analysed image with a blob from the previously analysed image, i.e. track the movements of the blob. Pairing blob lists from two consecutive frames provides all the necessary information needed for the tracking and counting. If an old

Angle range	Output
$[0°, 45°)$	0
$(-45°, 0°]$	0
$[45°, 90°)$	1
$(-90°, -45°]$	1
$[90°, 135°)$	2
$(-135°, -90°]$	2
$[135°, 180°)$	3
$(-180°, -135°]$	3

Table 3.1: *Angular range converted to integer*

blob is paired with a new one we know its new position, remaining blobs in the new list have just appeared on the screen and those remaining in the old list have disappeared from the image. When a blob disappears, one checks where it appeared and then if the person should be counted as entering, exiting or not at all, by looking at the measurement points.

The algorithm is designed with the former facts in mind. The algorithm traverses through a list of the new blobs. For each blob in the list it is compared with blobs in the older list and see if it can be paired with anyone of them. The pairing is based on seeing whether the blob overlaps the older one, if it does the two blobs are paired together. If more than one blobs are found to overlap, the one which correlates most closely to the new coordinates is chosen to be paired with the blob.

If a blob from the old list has been found and returned the new blob's originating region is set as the old blob's origin. Afterwards, the old blob is removed from the list. When the algorithm has gone through the whole list of new blobs the blobs remaining in the old list are the ones that have disappeared. By always updating the originating region one can then quickly look at the region of the disappearing blob and check whether the two regions are a part of any measurement point and then either the in or out count is updated accordingly.

If no blob was paired with the new one the only thing set is the originating region which is easily found by comparing the coordinates of the blob with the coordinates of the regions. The regions are kept in a global list which is set using the tracker settings. Finally the new list is set as the old list to prepare for the next time the algorithm will be called.

An addition to the tracking algorithm is a procedure which breaks blobs apart if they are wider than the maximum width of a person. This is done by computing the direction and checking whether a blob should divided by height or width. The direction is divided by $\frac{\pi}{4}$ and the decimals are removed (by forcing a conversion from double to integer) and using the absolute value of the outcome. Table 3.1 shows the output of each angular range.

When the algorithm outputs 0 or 3, the height of the blob is actually the width of it, while in cases of 1 or 2 the width of the blob is the correct one. Finding the correct case is then just a matter of using some light modulus calculations. When these are known one can compare them with the maximum width of a person and see whether they should be divided or not. This procedure is useful since roughly 20% of blobs in retail settings merge together at some point.

The algorithms for the blob tracking and dividing into more blobs are shown in appendix A.5 and A.7, respectively. The blob tracking algorithm uses an algorithm, described in A.6, to pair blobs together.

4 Implementation

4.1 Program Structure

Algorithms for background subtraction, blob analysis and tracking are implemented in an application, Peocounter, which does not separate them from a graphical user interface. This means that while the algorithms are executed the operating system and the application are constantly redrawn which limits the number of cameras per computer, if the counting should be done in real time. Since each camera can only have one counting area, only four entrances can be counted per computer. Based on this limitation it was reasonably early decided to build the application as a library. This makes the program easily manageable as well as expandable. Applications would not require to use all the function calls which means that the current version can be slowly migrated to the new version.

Peocon provides other solutions for the retail environment and by creating a library one can gather common methods and classes together. It does not only cover various solutions but also different versions of Peocounter. Using a library allows using two programs, e.g. one configuration program which can use a graphical user interface to setup the necessary configurations, ranging from offset in background subtraction to regions and measurment points. These can then be written to configuration files and then used by the main Peocounter application. Peocounter would use the library to access many of the same methods as the configuration program and run as a background deamon.

The current version of Peocounter is only compatible with a 32 bit Microsoft Windows operating system. The program is written in Delphi, a derivative of Pascal, which is mainly targeted at Microsoft Windows. There do exist some compilers for other platforms but for compatibility reasons and more known and accepted standards the C++ was chosen as the implementation language. C++ made creation of the library extremely simple. It also allows more compatibility with both different platforms and other programming languages as many of them, most notably the Delphi programming language, are able to use C/C++ libraries. This helps Peocon to slowly migrate from the current version so that the code because the code which has been written in Delphi can be slowly replaced by the library code.

Using only well known and accepted standards allows the program to be compiled on various platforms. This makes the software more flexible and does not force Peocon to choose between embedded devices or computer-based solutions. Possible target cameras for the embedded version were analysed thoroughly to give some idea of possible uses for the code. They are as different as they are many. The most promising ones were from Axis, a company based in Sweden. The cameras from Axis included their own operating system, a variant of the Red Hat GNU/Linux system. Axis provides their own C/C++ compilers and the cameras have a very simple interface for uploading software. Another company, Visual Components, based in Germany also have very powerful cameras which can be used. Their cameras are built around digital signal processors so the program can be compiled directly as a single application and flashed to the memory. These cameras demanded more direct hardware communications but were more powerful than the Axis cameras. No decision was made which camera should be used or whether Peocounter should be purely embedded, partially embedded, i.e. background

subtraction on the camera with blob analysis and tracking performed on a computer, or completely computer based. This means that the code is written with flexibility and platform independence in mind.

Since the cameras are so different using the GNU Compiler Collection, GCC, is a good choice. GCC is a free software, maintained by specially appointed committee, and well known for its extremely efficient optimisation and vast amount of target platforms. GCC is also the standard compiler collection for the GNU/Linux operating system so all code is developed and written using both Fedora and Ubuntu GNU/Linux distributions.

4.2 Classes and Methods

Figure 4.1 shows the class diagram of the library. Note that neither operations nor variables are visual. The operations are listed below in alphabetical order and categorised by the classes.

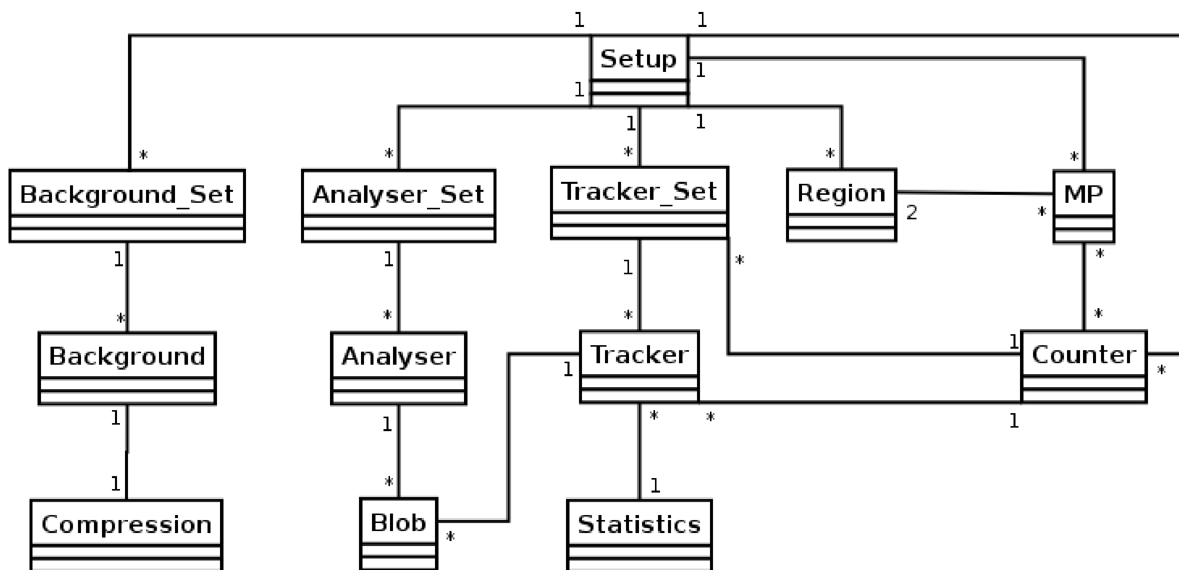

Figure 4.1: *Class diagram of the library, with multiplicities*

Since the product of this project is a C++ library both private and public methods will be accessible, therefore access control is not mentioned in the list. Each class and method is also described shortly to give an idea of their purpose. Simple *set* and *get* methods are excluded from the list, but instead mentioned in the description of the class. Destructors which do nothing, that is no memory must be freed, are not mentioned.

4.2.1 `analyser`

Takes care of analysing a run length coded blob image. Used after the background subtraction has performed its operations on the frame. The class uses *set* methods to manipulate global variables `height` and `width`.

```
void addBlob(blob*,int,int,int)
```

A black pixel run in the compressed image, represented by three integers (x and y coordinates of first black pixel and the length of the run), are added to an existing blob in a list (input to

the method). This is done by calling the `add2NB` method. If the pixel run is not adjacent to any blob in the list a new blob is created.

`blob* add2NB(int,int,int,blob*)`

This method goes through a blob list and adds a black pixel run to an adjacent neighbour an returns a pointer to that blob, if no adjacent blob is found, a `NULL` pointer is returned. The algorithm is described in appendix A.4.

`blob* analyse(int*)`

Main operation of the `analyser` class. Traverses through a compressed blob image which is given as an input and returns a list of blobs in the image. The algorithm is described in appendix A.3.

`analyser()`

Constructor with no parameters uses default settings. The settings used are height, set to 240 pixels, and width, 320 pixels, of the image. They are needed in order to work correctly with the compressed blob image.

`analyser(analyser_set*)`

Constructor with an `analyser_set` as an input and uses the settings from that object.

4.2.2 `analyser_set`

Settings for the `analyser` class. Set by the `setup` class and read from a configuration file. There are two settings, height and width of a regular image. The class uses *set/get* operation to manipulate the two variables which contain the settings, they can also be set directly using a certain constructor.

4.2.3 `background`

Manages background subtraction by working with a background image which can be set with a *set* method. In order to subtract the background correctly the instance of the class uses four variables for tweaking, offset, offset limit, lower limit and upper limit. They to are managed by *set* methods. Other variables needed, which can be *set* are height and width of the image. The input image must always be the same size as the background image.

`background()`

Constructor of the background class. Uses default values to set the variables needed for tweaking as well as the background image. The default value is zero (or `NULL` pointer for background image) for everything except height and width of the image which is set to 240 and 320, respectively.

`background(background_set*)`

Constructor which can set configurations found in the `background_set` parameter. The background image is set as a `NULL` pointer.

`background(unsigned char*, background_set*)`

Constructor which set configurations found in the `background_set` parameter and the background image as the `unsigned char` pointer.

`void subtract(int*,unsigned char*)`

Method which subtracts the background image from an input image, calls a compression method and places the output in the array of integers which is given as a parameter. The algorithm of this method is shown in appendix A.1.

4.2.4 `background_set`

An object which contains the `background` object's settings. Configurable variables are offset, offset limit, lower limit and upperlimit along with height and width of the image. The variables are manipulated using *set/get* operations but can be also be set directly when created by calling a certain constructor.

4.2.5 `blob`

Class which represents a countable person. Through functions one can manipulate the blob and interact with it. It has built-in variables and methods so that it can be a part of a double linked list. For this one uses *set/get* methods and a function, described below, `disconnet()`. The blob has a few other variables which can be accessed using *set/get* methods. These include the originating area, both starting and ending X and Y coordinates (represent a rectangle), and width as well as height. Two variables can be accessed only using *get* operations. These variables are the X coordinates of the first possible neighbouring pixel run and the length of that particular row. The blob object is created for row-wise additions of pixel runs.

`void add(int,int,int)`

This method adds a black pixel run to the blob. The pixel run is as usual represented by three integers (X and Y coordinates and length of the run).

`blob()`

Constructor, creates an empty blob object.

`blob(int,int,int)`

Constructor, creates a blob object and adds an initial black pixel run to the object.

`void disconnect()`

Disconnect the blob from a double linked list by connecting the previous blob and the next blob. Does not destroy this blob.

`int isNeighbour(int,int,int)`

Checks whether the blob is a neighbour to a black pixel run and returns 1 if so, 0 if not.

`void merge(blob*)`

Merge the two blobs together and delete the blob which is given as a parameter to the method.

`int overlap(blob*)`

Checks if a blob overlaps a blob which is given as a parameter. This is used for tracking to see whether the blobs possibly represent the same person or not.

4.2.6 `compression`

Class which handles compression of a black and white image using run length coding. It is designed to compress an image on the fly, i.e. future pixel values are not known. Background subtraction creates an instance of this class and when a decision is made whether a pixel is black or white, the value is sent to the `compress` function. When the whole image has been processed it is necessary to call the `finalise` method to finish writing to the integer array.

`void compress(int)`

On the fly compression of black or white pixel value represented by the parameter to the argument. The algorithm is described in appendix A.2.

`compression(int*, int)`

Constructor of the compression class. Parameters are an integer array which will contain the compressed image and the initial pixel value (black or white). Background subtraction uses white for an initial value.

`void finalise()`

Finish up and make the compressed image, i.e. the integer array, ready to use by the blob analyser.

4.2.7 `counter`

The main purpose of the counter object is to manage a list of measurement points. This list can be accessed by *set/get* functions. When the `add` function of the counter class is called, it

goes through the list and tries to add to the measurement points until it succeeds. A certain time intervals, when the user wants to access the in and out counts of the measurement points, an instance of the counter is accessed via the tracker class and from that instance the list of measurement points. Resetting the counts of all measurement points is done by calling the `clear` method.

`void add(region*,region*)`

Try to add a blob which appeared in region given as the first parameter and exited in the second parameter. This is done by calling the `add` method of the measurement points in the list until an addition succeeds or the end of the list is reached (blob should not be counted).

`void clear()`

Reset all counters of the measurement points in the list.

`counter()`

Constructor which sets no list of measurement points.

`counter(mp*)`

Constructor which sets a list of measurement points given in as the only parameter.

4.2.8 `mp`

Represents a measurement point, i.e. entrance and exit regions. Has built in functions so that it can be a part of a linked list which is managed by *set/get* functions. Other variables which can be manipulated in that way are the identity (an integer) of the measurement point and the entrance and exit regions. Two variables can only be retrieved from the measurement point and not directly set. These are the in and out count of blobs through the measurement point. They are incremented, indirectly, using the `add` method.

`int add(region*,region*)`

Try to add a blob passing from first parameter to the second parameter. If the regions comply with the entrance and exit the in or out count is incremented accordingly and 1 is returned. If not 0 is returned to mark failed addition.

`void reset()`

Reset the in and out counters of the measurement point, i.e. set them to be equal to zero.

`mp()`

Constructor of a measurement point, identity, entrance and exit are not set correctly when this constructor is called. Identity is set as -1 and both entrance and exit are set as `NULL`

pointer.

`mp(int,region*,region*)`

Constructor which sets identity, entrance and exit according to the parameters.

4.2.9 region

This class represents the rectangular area which can be an entrance and an exit. It does not have any additional methods to the constructors, destructor and a few *set/get* functions since that is not necessary. The variables which the class manages are the upper and lower X coordinates as well as the Y coordinates of the rectangle and a variable which allows the region to be a part of a list. The region can have an integer as an identity.

`region()`

Constructor which sets identity to -1, and all coordinates as 0.

`region(int,int,int,int,int)`

Constructor which uses the parameters to create a rectangle using coordinates and give it an identity.

4.2.10 setup

This class reads and manages the configuration files and creates `background`, `analyser` and `tracker` objects using static functions. All configurations are read from config files. This is not an official member of the end product and can easily be replaced by an application with a graphical user interface. The extremely simple interface is however recommended to be used instead of a graphical user interface which is only used for the configuration phase and nothing else. A graphical user interface would be better suited to create the configuration files which this class uses.

`static background* create_subtraction(unsigned char*)`

Creates a background subtraction object off the heap. Parameter to this method is the background image which should be used.

`static analyser* create_analysis()`

Creates a blob analyser object off the heap.

`static tracker* create_tracker()`

Creates a blob tracker and its counter object off the heap.

4.2.11 `statistics`

This class contains only static methods for various statistical computations. These computations could as well be a part of the `blob` class but were chosen not to, in order to enable more expandability for flow and blob analysis later on. At the moment the class only manages a few methods which can be used to compute blob statistics but later versions could include weather statistics, most-used path and various statistical functions which can be used for auto configuration of the software.

`static int difference(int,int)`

Finds difference between two parameters by subtracting the previous from the second.

`static double center(int,int)`

Returns the a double which lies directly between two integers given as parameters.

`static double distance(double,double)`

Computes the Euclidean distance between two differences, the former parameter shows the movement along the X axis and the latter along the Y axis.

`static double direction(double,double)`

Direction of movement computed from distance moved along the X and Y axis. Returns arc tangent of the two distances.

4.2.12 `tracker`

Manages the last phase of the counting process, the blob tracking. Used after the blob analyser has returned a new list of blobs. The only variable which is used is the maximum width of a person and this is manipulated using *set/get* functions. The list of regions which the blobs can appear or disappear in can also be *set* as well as the `counter` object. The most important *get* function of all returns the `counter` object instance so that the in and out counts of the measurement points can be retrieved and logged.

`void break_apart(blob*, double)`

This method is used to examine the direction of a blob and if it turns out to exceed the maximum width of a person it is broken into the appropriate amount of blobs. Detailed description of the algorithm can be found in appendix A.7.

`blob* find_probable(blob*,blob*)`

This algorithm goes through a list of blobs and finds the member of the list which is most likely to represent the same person. The algorithm is shown in more detail in appendix A.6.

`region* find_region(blob*)`

Goes through the list of regions and finds which region the blob is in. This region is returned and can be either the region where the blob appeared or disappeared, depending on when the function is called.

`void track(blob*)`

The main operation of the `tracker` class. Goes through a previous list of blobs and compares it to a new list given as a parameter. The blobs of the lists are compared, new blobs noticed and disappeared blobs are counted. The algorithm is described in detail in appendix A.5.

`tracker()`

Constructor of the tracker class. When this constructor is called it creates a `tracker` object with default settings which are all zero (`NULL` pointer for region list and counter objects).

`tracker(tracker_set*)`

Constructor which sets the maximum width, list of regions and counter object according to those contained in the `tracker_set`.

4.2.13 `tracker_set`

The object which contains configurations for the `tracker` class. These configurations are maximum width of a person, list of regions and the counter object. All of these are managed by using *set/get* functions or by using a certain constructor.

5 Conclusions

5.1 Results

5.1.1 Algorithms

Algorithm design and improvements have been the main research and focus of this project. Comparing the algorithms designed in this project with the ones in the older is an impossible task. The algorithm for background subtraction is essentially the same with only compression added.

Blob analysis was managed by proprietary code from Euresys s.a. so there is no method of comparing the two algortihms. One can however easily see that because the input to their algorithm is an uncompressed image their algorithm must at least travel through the image which forces their algorithm to work with all pixels of the image (typically $240 \times 320 = 76800$ pixels) which makes it more impractical.

The only comparable algorithm is the tracker algorithm. In a worst case scenario the newly devised blob has to go through all blobs via a nested loop, comparing the list of old blobs against each blob in the new list. In this scenario no blobs are connected and the algorithm must again go through the whole list of new blobs and then afterwards against the whole list of old blobs. In the older version, the one used in the Peocounter software, this was different. The algorithm in that version has to go through the new list five times, the old list ten times. Additionally it has to go through the old list for every blob in the new list three times and once through the new list for every item in the new list. The code for this section remains unoptimised and prohibitively impenetrable.

The algorithms have introduced some new features to the software. These include, for example, the compression which enables fast transmission if one would set the software up to perform background subtraction on camera and send the blob image for processing on a computer. Another feature is an improved multiple blob splitting. In the new version this is based on direction while in the old version this was based on X coordinates. So if a person would be walking across the image pushing a trolley it would be regarded as two or more blobs, this was meant that users would have to place the camera according to the software as the software is based on the camera angle.

5.1.2 Final Product

After the design phase of this project, when all algorithms had been carefully constructed and tested using a quick implementation in C++ some employees at Peocon started working on writing the algorithms in Delphi and incorporate them into the next release of the Peocounter software. By doing this they were able to test the algorithms extensively over about a seven month period by using it at real locations. The algorithms performed well in this implementation and they still are.

Meanwhile, the algorithms were also implemented as a part of this project in C++ and built

as a library. The final product is hard to compare with the Peocounter software itself as it is constructed to be a library used by the software. To be able to do some comparison relative parts from the Peocounter software were located and the source compared. The most obvious comparison is lines of code. The new version resulted in code which is only 60% the size of the old one. These results must be taken with precaution since they apply to different programming languages and other coding techniques. One must also notice that the code taken from the Peocounter software does not include blob analysis which the newer code does. The newer code also contains more comments since all code is documented using a program called *Doxygen* so that the employees of Peocon can easily understand the different variables and methods.

Difference in program sizes must also be taken with a precaution but may be worth mentioning. The Peocounter software is around 600 kilobytes while the new library is only about 30 kilobytes. The Peocounter software does, in contrast to the library, include a graphical user interface which explains the difference in size to some extent. However, in addition, the software also depends on a few external dynamic libraries which in total take about 7.5 megabytes of disk space.

The final product of this project is a single library which is totally independent of any external libraries and in contrast to the Peocounter application also independent of platform as it uses only static standard C libraries.

5.2 Problems

Like any other big project this one was not devoid of problems. However the largest problems with the project were not bound specifically to its task. Most of the problems derived from the legacy factors within Peocon and their lack of spare resources to support research work. Due to these facts the code is non-expandable and hard to modify.

To begin with the project was about trying to modify the Peocounter software and create an embedded device. Shortly after work started on the project it became clear that this would mean total redesign of the software and the goals changed from creating an embedded counting device to making the code modular and expandable so that it could easily be made an embedded device later on. The project has therefore resulted in a full redesign of the Peocounter software, and not in an embedded counting device as the original goal was.

Another key restriction was the difficulty with which knowledge of both the industry of people counting and detailed knowledge of Peocon's systems was gained. Owing to the nature of the business, the most efficient way to do this was verbally from Peocon developers and managers. The information and helpfulness of these employees was extremely valuable but made one very skeptical. Even though simply asking other employees is a fast way to get answers to ones question, this process disturbs the employees and one cannot help to think that some information gets forgotten and that it slows down the analysing phase of the project.

Start of work was slightly delayed while a GNU/Linux workstation was built and a project server setup. Peocon does not normally have a requirement to run GNU/Linux workstations and their source control system was out of date and not in use.

The problems were of course not only bound to the environment of the project. There were also problems with the project itself. The biggest one was a decision made when implementing the software. Using a C++ library which provides linked list created problems later on. This became problematic for the employees that implemented the algorithms in Delphi and used a third-party linked list, since the structure caused memory leaks. Later they created their own version which fixed the problems. Because of their problems the C++ implementation

was examined again. The decision of using a C++ library made the code itself not platform independent which it was claimed to be. It transpired that this was the only external library the code was dependent on and the need for an external linked list structure was removed by integrating the structure with the classes themselves, i.e. simply implementing the linked list data structure.

5.3 Future Work

After redesigning the core of the system some work is needed to create or adapt the Peocounter software to use the new library. The outcome can be similar to the old version of Peocounter or a software background deamon which runs on a workstation or even an embedded counting device similar to the one which was the original goal of the project. The embedded device will however not be created at Peocon since they are a purely software company and have no plans to expand into hardware development. Some knowledge of hardware and low level programming will be needed to make the embedded device. Whilst this option is open, it differs fundamentally from Peocon business strategy.

Improvements on the library can also be made. The library should be easy to expand and modify. Propositions for new features are, for example, background adaptation or noise removal (which can be a byproduct of introducing filters), et cetera. It is also possible to expand the `statistics` class to gather analytic information such as weather reports, common paths through the counting area and so forth.

Peocon is unlikely to invest any more resources in the devlopment of the embedded counting device. The possibility of buying embedded counting devices has surfaced. A company called Brickstream, based in the United States of America, started to offer embedded people counting devices which employees are examining. If these devices are accurate enough they will be used with other software from Peocon to analyse visitor counts and retail data. The devices are still under observation and no decision has been taken.

5.4 Conclusion

People counting is a challenging task. It is extremely difficult to automatically and accurately count every visitor that passes beneath the camera, in real time. Most software designs have some special cases which result in a wrong count. The focus of people counting software is therefore to provide as accurate numbers as possible.

Even though the Peocounter software, at time of writing, was an immature design featuring sub-optimal algorithms, it did its work. This project has not resulted in greater accuracy, but more efficient algorithms which then allow implementation of changes to the current systems. These changes will allow each computer to manage a greater amount of cameras in real time. The software is more modular and new features can be easily added so that it can be easily adapted to new surroundings and requirements which enables Peocon to expand their customer group. With a little bit of effort Peocounter can become a more powerful and accurate counting application with many exciting features and be applied to various markets, not only visitor counting.

6 Bibliography

Literature

[Gon2001] **Gonzales, Raphael C., Woods, Richard E.**
Digital Image Processing
Prentice Hall, 2001

[Jai1989] **Jain, Anil K.**
Fundamentals of Digital Image Processing
Prentice Hall, 1989

[Rus1999] **Russ, John C.**
The Image Processing Handbook
CRC Press, 1999

[Str2000] **Stroustrup, Bjarne**
The C++ Programming Language
Addison-Wesley, 2000

Articles

[Axi2005] **Unknown**
Axis 211A Network Camera
Architect & Engineer Specification, 2005

[Sig2002] **Sigvaldason, Einar**
People Flow Solutions
Fact sheet, 2002

[Vis2005] **Unknown**
Vision Components VCSBC4018 Manual
Hardware and Programming Manual, 2005

Webpages

[WIKI2006] **Wikipedia**
`http://en.wikipedia.org/wiki/Speed`
2006-07-01

Appendices

Appendix A

Algorithms

A.1 Background Subtraction

Algorithm used for background subtraction, which identifies foreground objects in an input image by comparing to a background image.

```
Input: image in, image bg
Constants: double pUp, double pLow, int offsetThreshold
           int offset, int width, int height
Output: pixelvalue o

Algorithm:

For every pixel p where p.y < (height-offset)

  bgdiff := |bg.p - bg.(p+offset*width)|
  fgdiff := |in.p - in.(p+offset*width)|
  offdiff := |bgdiff - fgdiff|

  upper := (1+pUp)*bg.p
  lower := (1-pLow)*bg.p

  if(in.p > upper
      or
     in.p < lower
      or offdiff > offsetThreshold)
    o = black
  else
    o = white
```

A.2 Compression

Run length compression used to compress a black and white blob image for faster image processing and network transmissions.

```
Input: pixelvalue v
```

```
Global variables: array a, pixelvalue l, int counter
Output: array a holds output

Algorithm:

if(v = l)
  counter++
else
  write counter to array
  counter := 1
  l = v
```

A.3 Blob Analysis

Blob analyser used to identify blobs in a run length coded blob image and add them to a blob list.

```
Input: compressed image cmprs
Constants: int width, int height
Output: list blobs

pixelcount := cmprs.value
cmprs.value := cmprs.nextvalue

While pixelcount < (width*height)
  ycoord := pixelcount/width
  xcoord := pixelcount-(ycoord*width)

  pixelrun := cmprs.value

  if (pixelcount%width)+pixelrun > width
    pixelsRight = width - (pixelcount%width)
    addtoblob(blobs, pixelsRight, xcoord, ycoord)
    ycoord++
    xcoord := 0
    pixelrun := pixelrun-pixelsRight

  addtoblob(blobs, pixelsRight, xcoord, ycoord)

  pixelcount := pixelcount + pixelrun + cmprs.value
  cmprs.value = cmprs.nextvalue
```

A.4 Adding Pixel Run to a Neighbouring Blob

Used to add a pixel run to a neighbouring blob found in a list. If a neighbouring pixel does not exists this algorithm returns 0 which indicates that a new blob should be created.

```
Input: int xcoord, int ycoord, int amount, list blobs
Output: blob r

r := 0

While blobs has another element x
  if x.isNeighbour(xcoord, ycoord, amount)
    if(r != 0)
      r.merge(x)
    else
      r := x
      r.add(xcoord, ycoord, amount)
```

A.5 Blob Tracking

This algorith is used to pair blobs from a new list of blobs with an older one. It traverses through the newer list if a blob is paired with an older one its originating region is updated. The remaining blobs of the new list are the blobs which have recently appeared while the remaining in the old list are those that have disappeared.

```
Input:  list newblobs
Global variable: list oldblobs

While newblobs has another element x
  p := findprobable(x,oldblobs)
  if(p is a blob)
    x.origin := p.origin
    remove p from oldblobs
  else
    x.origin := findareaof(x)

While oldblobs has another element y
  if(y.origin and findareaof(y) apartof mp)
    add count to mp

oldblobs := newblobs
```

A.6 Find Probable Blob

This algorithm finds an overlapping blob which appears to have moved the shortest distance from an input blob. If no blob is found nothing is returned. The input blob is broken into multiple blobs if it is wider than a maximum width. This is done by calling the algorithm in appendix A.7.

```
Input: blob b, list blobs
Output: blob r

r := 0

While blobs has another element x
  if b.overlaps(x)
    if(r = 0)
      if(euclideandist(b,x) < euclideandist(r,x))
        r := b
    else
      divideblob(x)
      r := x
```

A.7 Dividing a Large Blob

This algorithm takes a look at the direction of a blob and sees whether a large blob is wider than it should be. If so it divides the blob into smaller blobs, either by height or width according to the direction.

```
Input: blob b
Constant: int maxwidth

dir = (int) b.direction / (PI/4)

if(dir%3 = 0)
  amount = round up(height/maxwidth)
  if(amount > 1)
    newheight = height / amount
    While i = 0 < amount
      x := new blob
      x.width := b.width
      x.height := newheight
else
  amount = round up(width/maxwidth)
  if(amount > 1)
    newwidth = width / amount
    While i = 0 < amount
      x := new blob
      x.height := b.height
      x.width := newwidth
```