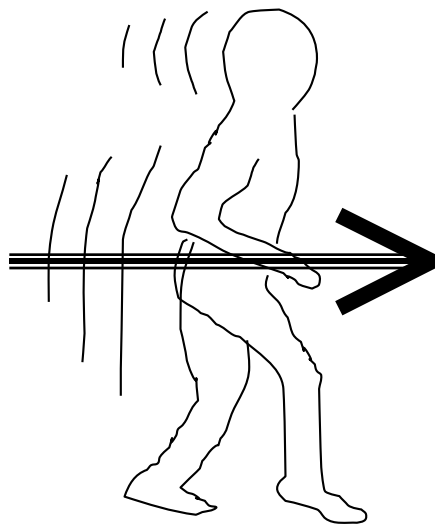


Technion – Israel institute of technology,
department of computer science.

Advanced Programming Lab B

People Counter Using Image Processing



Hanoch 'Nuke' Goldstein
024894602

July 1999

1.	INTRODUCTION	3
2.	PROJECT'S HARDWARE AND SOFTWARE	4
3.	SOFTWARE DESIGN.....	4
4.	SETTING THE 'AREA OF INTEREST'	5
4.1.	PROCEDURES AND FUNCTIONS USED	6
5.	MOTION FILTERING	7
5.1.	THE MOTION FILTER	7
5.2.	SECOND PASS (CLEANING THE RESULT).....	8
5.3.	PROCEDURES AND FUNCTIONS USED	8
6.	SUB-REGIONS IN THE AREA OF INTEREST.....	9
7.	IDENTIFYING A MOVING OBJECT WITHIN A SUB-REGION	10
7.1.	TRYING TO SEPARATE PEOPLE.....	10
7.2.	PROCEDURES AND FUNCTIONS USED	11
8.	THE STATE MACHINE.....	12
8.1.	EXAMPLE OF THE STATE MACHINE WORK.....	13
8.2.	COUNTING IN BOTH DIRECTIONS	13
8.3.	PROCEDURES AND FUNCTIONS USED	14
9.	THE MAIN LOOP	15
10.	TESTING &RESULTS.....	16
11.	CONCLUSION	17

1. Introduction

This project attempts to create a core for a marketable product with actual use in the market. The product at hand will make use of image processing techniques in order to keep track of human motion at a predefined area within a live video feed. The motion will be then analyzed to give the number of people that passed in front of the camera in each direction. The 'neat' trick is that the product will be able to count people moving in each direction in the same area on the screen, not only that but most of the times even people moving in opposite directions in the same times will be counted correctly.

As it is this project is the core for a potential product which can be used by medium and large establishments where large amount of people pass throughout the day. The product may help to handle statistic counting and security tasks day and night. For example, a commodity store may need to keep statistics on customer traffic throughout the workday, and at night leave the same system as a security motion detector.

The project at hand will not provide the implementation of security application only the core of statistic handling, i.e. defining the area of interest and counting people as they pass.

2. Project's hardware and software

The project's software was written in C on a Pentium 133MHZ PC (32M RAM) using MS-Visual C++ 6.0. A BarGold IVP-150 hardware card handles the image processing. There is some special attention not to use unique features of the hardware (look up tables and so on), but rather some universal filters that can be later transferred to a different platform. The only hardware need using this design is that the video feed will be interlaced. The interlaced image was used to filter out the motion (as will be explained later on).

I used the IVP hardware for its frame grabber and overlay display, and also to get a boost of performance when using area filters. Using these conditions I achieved a performance of up to 2-3 frames per second. There is a strong possibility that a better PC will increase the frame rate. As the performance betters the better are the results. A good PC might get close to real time performance.

In order to simulate a commodity store I shot a video from a security camera placed in one of these stores. I then used the videotape as if I was given live video feed. No slow motion or other tricks where used.

3. Software design

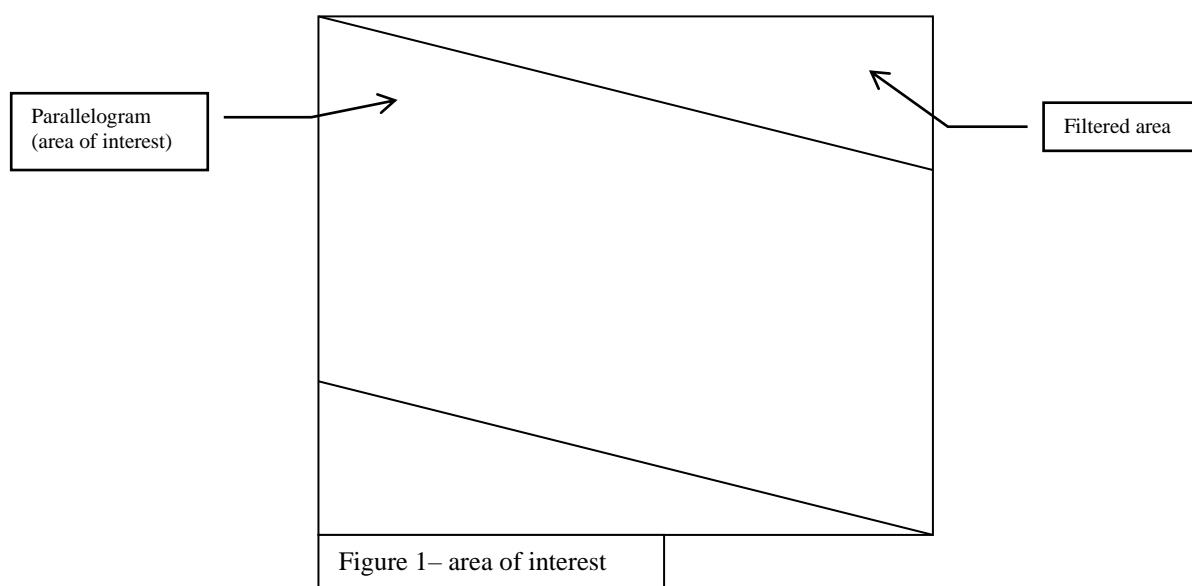
In order to achieve the people counter this project aims to provide, a simple idea was implemented. The idea was to use video motion detection, then evaluate its 'mass' of 'motion trail'. The bigger and faster the object moves the larger is its 'motion trail'. Since the camera is placed in areas like a supermarket entrance, most people move at walking speed. That and some other tricks (later explained) help when trying to separate people from grocery carts for example.

Using a set of serially placed, the software will create a set of binary (true if there is motion, false otherwise) values. These values are collected from all regions and an integer number is created. That number represents a state and processed by a state machine.

The purpose of the state machine is to follow the motion through and decide when a counter should be increased (according to the direction of the motion). Although a state machine is not the ideal solution in case we will want to expand or decrease the number of corresponding regions, it is the easiest replacement for a smart solution. Since AI decisions are not the topic of this project and since it does an adequate job this was the solution and implementation.

4. Setting the ‘area of interest’

There is no point of performing heavy duty image processing and number crunching on the whole image. Most of the time just a small area will be the one we have interest in, I will refer to it as the ‘area of interest’. Most security cameras are located overhead observing the people in an askew angle. The area of interest is an imaginary parallelogram in which people pass through (*figure 1*). To choose that area I used a man shaped image to set the direction of entering people and the average size of a man in that area. The man’s image can change its location (using the mouse), and its size and direction (using the keyboard). After setting these parameters the software automatically creates a rectangle. To set the angle of movement this rectangle is shaped by the user in order to get the requested parallelogram.



4.1. Procedures and functions used

- **void choose_man_region(parallelogram *r)**

This procedure will draw on the video monitor a vectorized man shape which can be manipulated by the keyboard and mouse. After setting the requested size and direction *angle_parallelogram()* is called. The values of the returned parallelogram are set into *r.

- **void angle_parallelogram(parallelogram *r)**

Using the keyboard the parallelogram's angle and position may be set. The values are set into *r.

- **void choose_region(parallelogram *r)**

This procedure enables a different method of choosing the area of interest. It is called from the main menu (pressing 'R'). It sets the parallelogram using two points and its height.

5. Motion filtering

The first stage of this project was to create an adequate motion detector. To achieve that a video-interlace affect was harnessed. When pausing an interlaced video while shooting motion, blinking fogged out areas appear where motion is. This blur is created because of a very short delay between the image captured in the odd horizontal lines and the even ones. The bigger and faster the object moves the bigger these blurred areas are. It was more then natural to try and use this feature in order to create a motion detector. In order to ‘paint’ these areas and clear any still area, a custom filter was created. This filter differs between the odd and even horizontal lines. Since the blurry areas are most common to pop up when applying this filter, a second cleanup would leave us with white areas that signify the motion areas.

5.1. The motion filter

Two filter passes are used in order to get the motion area highlighted. The first filter is a simple 3X3 (K_FILTER set in *demo.h*):

1	1	1
-2	-2	-2
1	1	1

This filter simply differs the odd and even lines. Since the motion blur is much ‘cruder’ than horizontal edges this filter will create brighter and larger areas for them. I experimented with numerous filters that use the same idea, for examples a 5X5 filter:

1	1	1	1	1
-2	-2	-2	-2	-2
2	2	2	2	2
-2	-2	-2	-2	-2
1	1	1	1	1

This filter got somewhat better results but took valuable performance time. The 3X3 filter proved adequate for the job and was the smallest I could use in order to get the best performance out of the system.

5.2. Second pass (cleaning the result)

After running the motion filter some edges in the image still appear on the resulted image. These can be remove using the IVP's *binary_area()* function. This function simply calculates the average pixel value of a square area (around the pixel) and highlights it (gray scale of 255) if it exceeds a threshold value or erases it otherwise. I used a 3X3 area with a threshold of 100 (out of 256 gray levels) to get good results with the motion filter I used. Other filters may need a different threshold to get optimal results.

5.3. Procedures and functions used

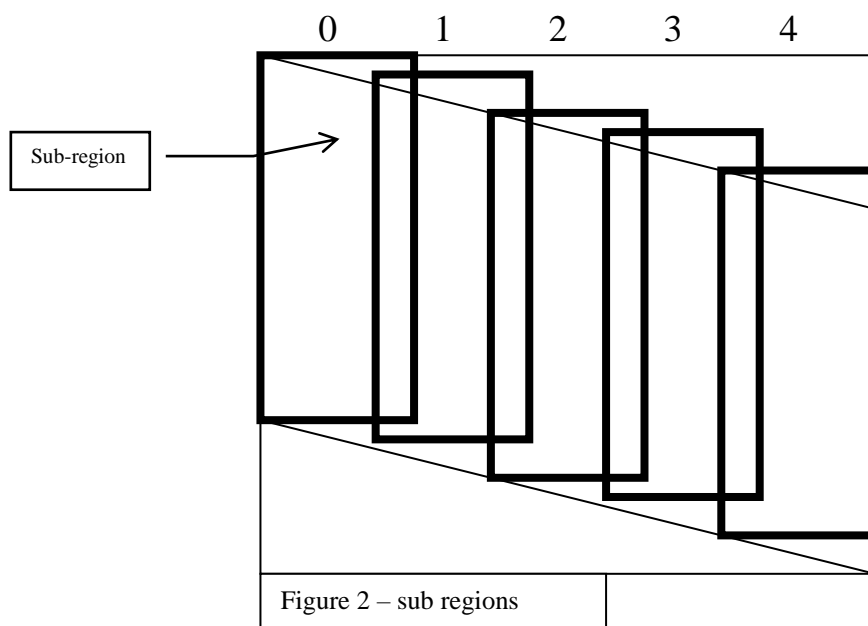
- **void movement_filter(region r, int *k, int mem)**

This procedure is a straight forward routine that activates the previously explained filters on the predefined area of interest. The filters are implemented on a square region that contains the chosen parallelogram. Writing the functions that perform the filtering was spared from me since the IVP software library handles that task.

6. Sub-regions in the area of interest

In order to decide the direction of the motion inside the area-of-interest, several sub-regions are created (*figure 2*) and checked before the data will be transferred to the state-machine (explained later on). For simplicity reasons this project uses a constant number of 5 sub-regions. Each sub-region has an overlapping area of 25% of its surface with an adjacent sub-region. This overlapping area insures tracking the flow of motion between the regions. If the sub-region do not have a common area between them a moving mass passing from region to region might be lost, because its relative ‘motion-blur’ will be divided between two or more sub-regions so the total threshold in each sub-region may not be achieved. Another good reason for the overlapping area will be better understood later on when trying to separate humans from other moving objects. In short the mass of the ‘motion-blur’ created by the heads is taken into account, so we’ll want to keep track of where the head of a man is and if it has passed between two regions.

The sub-regions are serially numbered from 0 to 5. The idea is that a motion that started from region 0 and ended up in region 5 is considered as an ‘entering’ person while the opposite direction is considered ‘exiting’.



7. Identifying a moving object within a sub-region

Up until now we filtered out the motion in the area-of-interest and divided it to several overlapping sub-regions. Now its time to decide whether a sub region contains enough movement to be considered as a person or not. To do that we need to sum up all the highlighted pixels inside every sub-region. Trial and error led me to the conclusion that if 2% of the sub-region has motion in it than it is considered as a potential person moving.

7.1. Trying to separate people

The aim of this project is not to create a smart human identifying and tracking machine, it is a completely different story, but in order to better the performance of this machine motion alone is not sufficient. Consider the case of a bunch of people moving in line, or a person shoving a cart, these examples may create a motion in each and every sub-region. In that case our state machine will be stuck on the same state while people that need to be counted pass by.

Since the area-of-interest does not suppose to contain more that two persons walking one after another, counting the number of heads provides a partial solution. More profound explanation will be provided in the state machine section. Heads are considered as peaks in the heights of the motion sub-regions.

7.2. Procedures and functions used

- **int sum_area_pixels(region r, int thres, int mem)**

This function counts pixels within a rectangular region *r* up to a number of *thres* pixels. If the threshold of pixels was not achieved a value of 512 is returned (the returned value is used for counting head later on). If a smaller threshold, that is considered noise, is passed then the height of this point is saved and returned when and if the actual threshold is passed.

- **byte check_movement_in_subregions(region subregions[], word *heads_vector)**

Using *sum_area_pixels()* every sub-region is checked for motion. A visual indication on the monitor is displayed as an 'X' above that region. The binary state of each sub-region is collected to an integer, to be used by the state machine. For example: if regions 0, 1 and 4 indicated motion than a 10011000 = 98 hex value is returned.

- **byte count_heads(word *heights_vector , int thres)**

This function uses the heights returned by the *sum_area_pixels()* function and the region states calculated by the *check_movement_in_subregions()* function in order to estimate if there is one or two heads inside the region. A head is considered a peak in the graph created by the parameters given.

The following example will return **2** heads since there are two peaks portrayed by the graph (to be compatible with video display 0 is considered top while 512 is bottom):

Region	0	1	2	3	4
Height	10	60	512	20	50
Motion State	1	1	0	1	1
Peak	X			X	

8. The state machine

The function of the state machine is to keep track of the history of the motion, and decide if an exiting or entering motion was completed. 16 states are supported. A state value is decided by the value retrieved by the *check_movement_in_subregions()* function, that was explained earlier. The value is masked (STATE_MASK[]) and compared to the states supported (STATE_VALUE[]). The program checks if the previous step may lead to the current step and if the change of state leads to a counter increment. This information is given in the states table array (STATES_PASS[][]).

If the states table row (previous step) and column (new step) is marked 0, than the state is not changed, otherwise the new step will be the next step. If the table returns a value that is 2 the counter incremented, if it is 3 or 4 the number of heads counted becomes a factor before incrementing the counter.

8.1. Example of the state machine work

Lets follow the following 10 results that the motion analyzer returned:

Step no.	Current state	Check state	Masked State checked	Table return value	Remark
0	0	10000000=80	80 = state 1	1	Entering
1	1	11001000=C8	C0 = state 2	1	Noise filtered
2	2	01100000=60	60 = state 6	1	
3	6	01101000=68	60 = state 6	1	Noise filtered
4	6	10111000=B8	98 = state D	1	Second item entering
5	D	01011000=58	50 = state 8	2	Counter++
6	8	00101000=28	08 = state C	0	Following second only
7	8	00110000=30	30 = state 7	1	
8	7	00011000=18	18 = state B	1	
9	B	00000000=00	00 = state 0	2	Counter++

The state machine is designed to follow motion from the most significant bit (10000000) to the least significant relevant bit (00001000). Since we're dealing with only five sub-regions the last 3 bits are not used.

In this demo two objects the machine knew to count both items that moved through the area, even though the second item entered while still tracking the first item. Note that in steps 1 and 3 the machine knew that the new item is noise since it didn't follow it from the start.

8.2. Counting in both directions

In order to check for movement in two directions the same state machine is used but with a mirror image of the state. Since it works with one side it will work with the other. The mirror is done on the 5 relevant bits.

8.3. Procedures and functions used

- **BOOL get_motion_state(byte in, byte prev_heads, byte curr_heads, state current, state *next)**

This function handles the change of states and decides if a counter increment is due. The function will return TRUE in case the counter should be increased and FALSE otherwise.

- **int no_of_heads(byte heads_state)**

This function retrieves the number of heads calculated for the state.

- **byte mirror_state(byte in)**

Create a mirror image of the state. Used to check exiting movement.

9. The main loop

The main loop of the program gives the user some control on the program. The user may change views, redefine the area of interest and so on. The following commands are available:

- Z** - Zero all counters
- R** - Reselect work region (using parallel lines)
- O** - Overlay on/Off
- I** - General information of the IVP-150 use.
- V** - Toggle overlay views
- S** - A demo of a potential statistics window
- F** - Fast flag On/Off (performance over quality – IVP flag)
- T** - Show performance timer (milliseconds per frame)
- 1** - Show the WORK memory and run a cursor menu (IVP style).

In the main loop the whole process of filtering the image and analyzing it is committed. In here the state machine is called and the counter handling and displaying is done.

10. Testing & Results

Using a video shot from a supermarket during a regular working day with a work rate of 3 images per second, the system achieved a success rate of around 80%. There was a noticeable improvement when the frame rate increased (by reducing the size of the area of interest).

Some hard cases were overcome by the software and counted correctly:

- Group of people moving in one direction.
- Two or more people moving in opposite direction
- A man entering the area and changing his mind halfway
- People with shopping carts
- A running kid

11. Conclusion

There are some obvious weak points in this application. It could not handle a rush hour, when many people are moving constantly around the camera in both directions. It won't be able to handle situations when people are walking side to side hiding each other.

A perfect machine would be able to recognize and follow a distinct person and have enough wisdom to distinguish him in a group even when partly obscured. The perfect machine will know to distinguish kids from adults and from shopping crates with 100 percent reliability.

This is not a perfect machine and does not pretend to be one. It is a core idea for a low cost machine that can give good cost-performance to the users.

As a basic implementation there are few uses to this project. In order to give it life, new and improved features may be added. It can be used as a statistic counter in the workday, connected to a global or a local database, and a security guard at night with Internet features.

To make a commercial use the project must aim to be fully automated, finding its own area of interest. Automation will make this product an 'off the shelf' low cost machine with numerous possibilities of use for many potential users.