AS-0.3200 AUTOMAATIO- JA SYSTEEMITEKNIIKAN PROJEKTITYÖT

CEILBOT

FINAL REPORT

Jaakko Hirvelä

GENERAL

The goal of the Ceilbot-project is to design a fully autonomous service robot moving in a roof instead of a floor. This makes the moving and energy supply much easier. This project is implemented under the research unit GIM. Project is being implemented as part of TKK's digitalization and energy-related research - Multidisciplinary Institute of Digitalisation and Energy (MIDE).

In the beginning of the project this semester two students – me and Juhana Leiwo - focused in the development of the mapping algorithm for the robot. Juhana is developing the 3D mapping and I will be focusing in a few 2D machine vision algorithms.

OVERVIEW OF USED ALGORITHMS

My part of the mapping algorithm consists of three different algorithms: texture segmentation, object matching and motion detection. The goal of texture segmentation is to be able to distinguish objects with different textures from their backgrounds. This makes the robot able to know whether there are different objects on a table or a floor and make further decision accordingly. For instance grab a found object.

Object matching takes a reference image of an object which is been sought from the room and if it finds a similar looking object further decision can be made accordingly. The main idea of this method is to be able to find things a person has lost. For instance if we are talking about an older person the robot can help him or her find important objects and give them to him or her.

Motion detection – as the name suggests – detects motion in the room. The purpose of this method is to detect and segment motion so that the robot knows where movement has occurred and stay out of those areas to avoid collision. The next chapters will focus in detail to these methods and describe their methods and problems.

TEXTURE SEGMENTATION

This method consists basically of three different parts: pyramid segmentation, flood filling and segmentation. Pyramid segmentation is the easiest part since openCV has a function called cvPyrSegmentation which uses pyramid segmentation to segment the image. If the following image (Image 1) is used as the basis and pyramid segmentation is applied to it we receive the following image (Image 2).

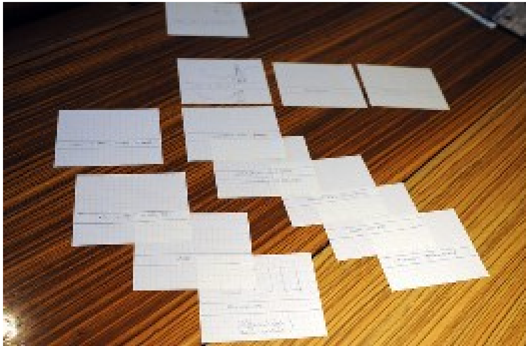

Image 1.

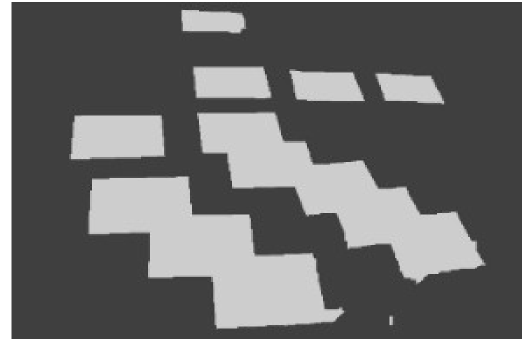

Image 2.

Since cvPyrSegmentation is openCV's own function I won't be going into details how the function works. After the segmentation one problem arises, which is parameter adjustment. This is kind of a problem with an autonomous robot since the robot can't know by itself when to alter the parameters. As soon as brightness changes parameters need to be adjusted so this function is very sensitive for variations in the surroundings. One solution to overcome this problem is to add reference images to every wall in the room. After this, the robot is tought the positions of each of the reference images. Finally initial parameters need to be fed for the algorithm to produce a clear segmented image from the reference images. These images can now be used as a reference in further segmentation. Since the robot is now thought how each of the reference images need to look like, it can adjust the parameters by itself as long as the resulting image looks exactly the same as the reference image that was manually tought to the robot.

After the pyramid segmentation, flood fill is used to define a distinct grayscale value for each individual area in the image 2. OpenCV has a function called cvFloodFill which fills a distinct area with a predefined grayscale value. With this kind of a method we can calculate how many different

objects there are in the image and their locations. After cvFloodFill the following image is received (Image 3).

Flood filling is used so that the image will be gone through pixel by pixel starting from the upper left corner – co-odinates are 0,0. Since distinct objects are marked as white (Image 2) the method knows that as soon as a pixel has a grayscale value of 255 it knows that a flood fill operation needs to be applied to the area containing this pixel co-ordinate. However, since noise always occurs we need to be sure that there is a larger area of white pixels to remove the effect of noise. This is why the method always scans the area of the found white pixel and calculates the proportion of white pixels relative to the scanned area. If the proportion is greater than a predifined value, flood fill is used.

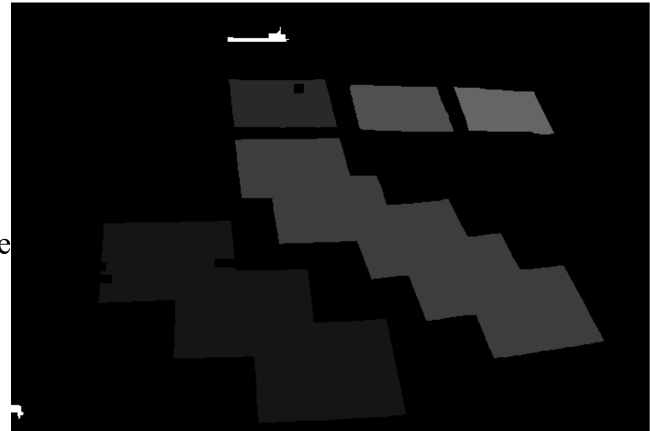

Image 3.

The next and final part consists of calculating the co-odinates of each distinct area with a unique grayscale value. Since we now know the grayscale values of different areas filled with flood fill, the algorithm uses threshold to make only one part of the segmented image visible as shown in the image 4 and image 5.

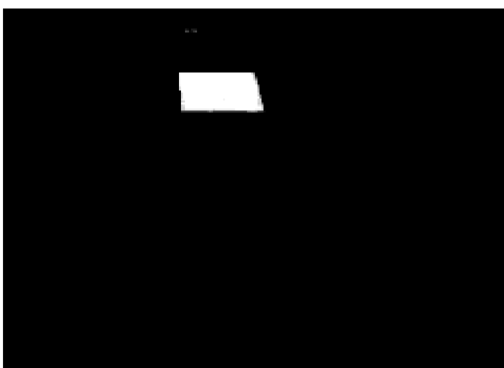

Image 4.

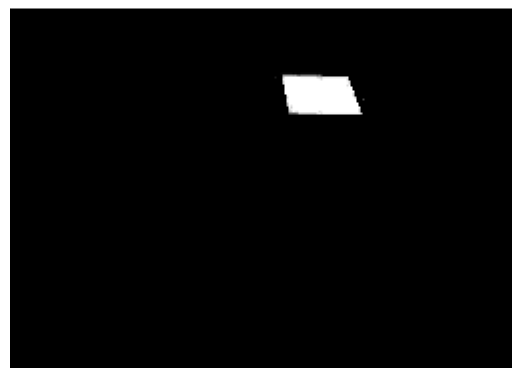

Image 5.

After each threshold we calculate the co-odinates of each of the areas. This is a simple method described below. First the method starts scanning the image from left side of the image and as soon as it finds a first white pixel it takes the x-co-ordinate of this pixel. Then it scans starting

from the right side of the image, stops as soon as it finds a white pixel and takes the x-co-ordinate of the pixel. If we calculate now the average of the two x-co-ordinates we have the x-co-ordinate of the area. The same idea is applied in calculating the y-co-ordinates of the object. The method is shown in image 6.

Blue circles correspond to found y-co-ordinates and red cirles correspond to found x-co-ordinates. The green circle in the middle of the white area is the calculated co-ordinate of the object. This value is then stored to a list which is used for further processing. After all of the previously described methods a following image is produced containing the middle co-ordinates of each of the distinct areas (Image 7). According to this algorithm six objects were found in the image. This method is not capable of distinguishing between overlapping objects, such as overlapping papers in the image. The 7th paper on top of the image was not found. However by altering the parameters it is highly possible to find it as the 7th object.

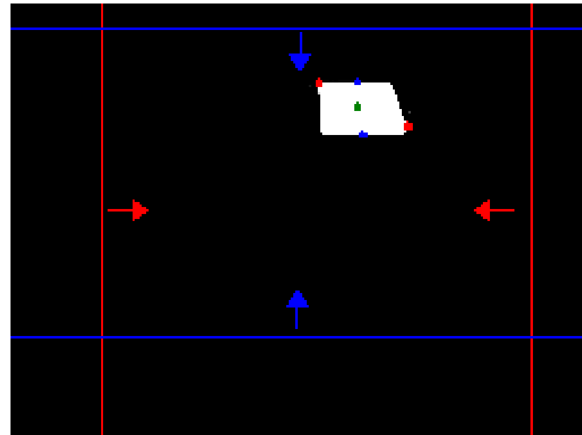

Image 6.

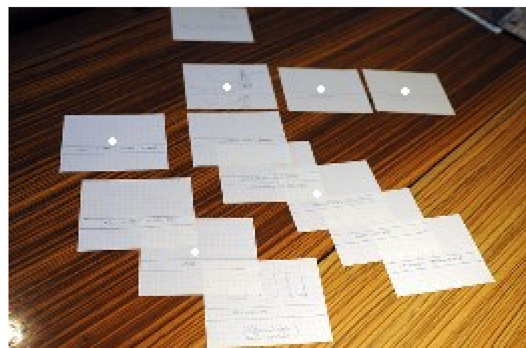

Image 7.

OBJECT MATCHING

First a reference image of the object - which is being searched - is fed to the algorithm. Next the method uses openCV function called cvEctractSURF which finds robust features in the image. For each feature it returns its location, size, orientation, direction, laplacian, hessian and optionally

the descriptor, basic or extended. In image 8 all the found features are circled in red, this is the reference image.

Function CVExtractSURF is applied both to the reference image and the video image. After these features have been extracted the method starts to look for similar features which have similar orientation, size, hessian and location relative to other features. When the method has found all the features in the video image it starts to go through them one by one. The method now starts to compare the feature found in the video image with the features found in the reference image. It compares the size, direction, laplacian, hessian and location relative to nearby features. If this feature matches with one of the reference features (size and orientation) then its surrounding features will be checked against the surroundings in the reference image. If they all match a similar feature has been found. The most important part is the checking against relative location between different features. This prevents finding similar features that actually don't belong to the object to be searched. The white lines correspond to matching features. If a certain precentage amount, in this case 30%, of original features are found, then the algorithm calculates the co-ordinates of the found object (Image 10, white cirlce). However it must also be mentioned that this method is far from robust since the reults vary quite drastically betwwen adjacent frames. For instance if the method finds the reference

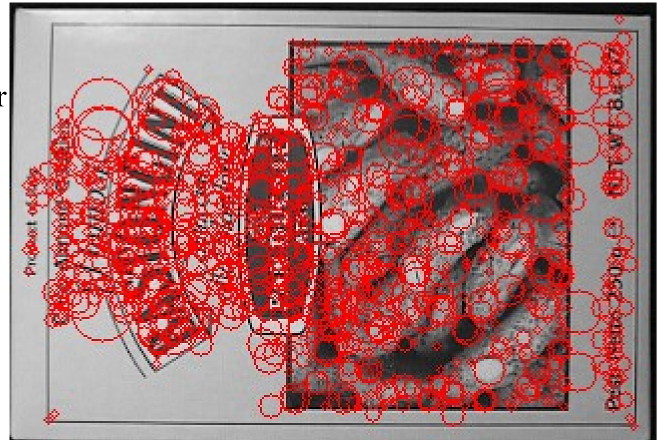

Image 8.

Image 9.

object in one frame it might not find it all in the next frame.

Image 10.

MOTION DETECTION

Motion detection consists also of three different sections: the difference between adjacent frames, contour extraction and motion segmentation. The basic idea in detecting motion is to calculate the difference between the current and previous frame. If motion has occurred in some parts of the image it is constituted so that the grayscale value of that pixel area changes (Image 11).

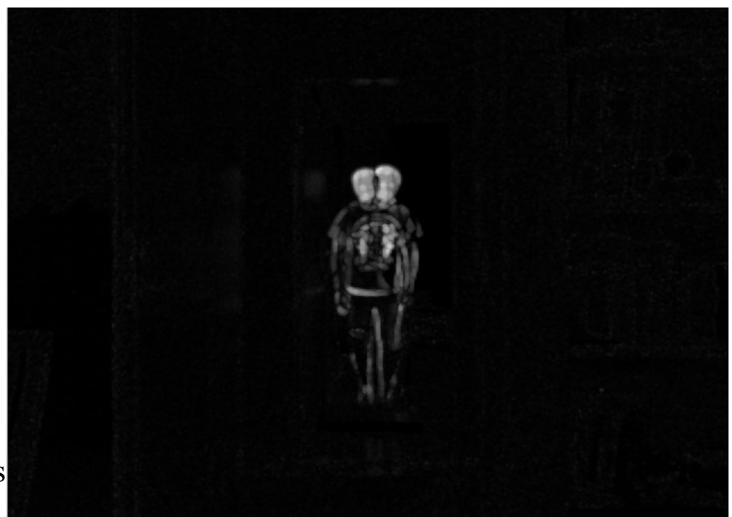

Image 11.

The white pixels represent moving objects in the resulting image. OpenCV function called cvAbsDiff is used in calcuting the pixel differences. However only the outliers of the moving objects will be noticed. This is because the grayscale values of the moving object change only in the edges and the grayscale values in the middle of the moving object stay the same. After this some morphological operations needs to be performed to smooth out the noise and connect the motion areas.

Next part is making a motion history image by using openCV function called cvUpdateMotionHistory. The basic idea is that the function takes a time value as a parameter which is used to build the motion history image. For instance if the time parameter is 30ms all the motion images (image 11) happened 30ms before will be summed to the motion history image. This method is needed to smooth the resulting image. Remember that only the outliers of the moving object will be recognized. When using this kind of motion history function the motion image will have much more solid motion regions instead of scattered.

Image 12.

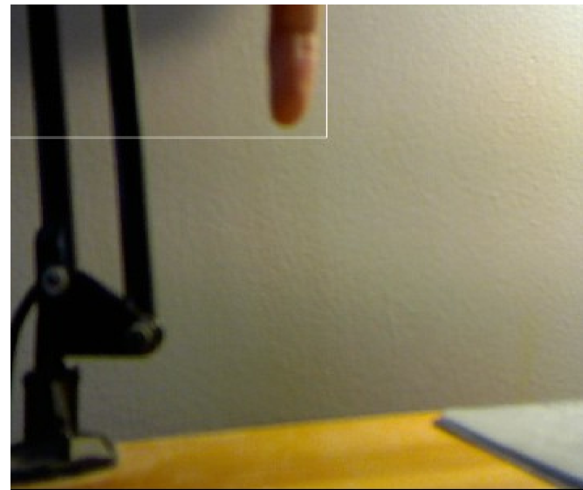

Image 13.

The final part consists of calculating the co-ordinates for each of the distinct motion areas as well as segmenting the motion into distinct areas. OpenCV function called cvSegmentMotion is used in segmenting the areas. It defines grayscale intensity values 1,2,3.... for each unique motion area (white area in image 12). Since noise always occurs in machine vision the method always checks that the areaof the found motion areas is large enough. If the area is small then we can safely assume that it is most likely noise and remove it from motion segmentation.

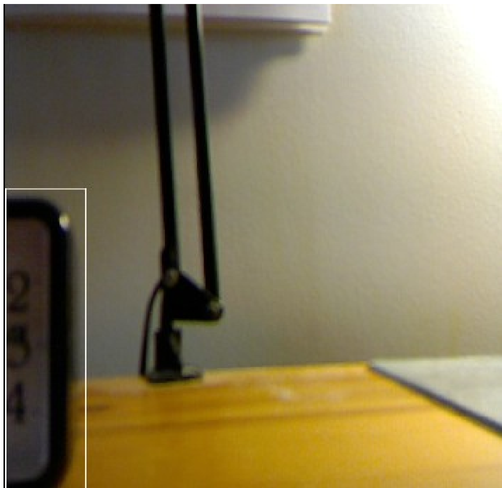

Image 14.

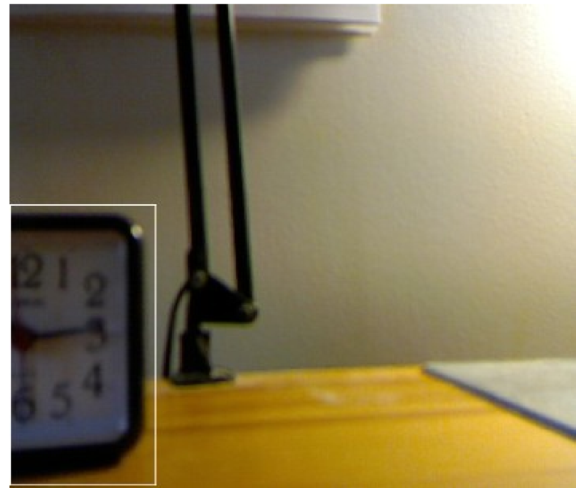

Image 15.

Since this kind of algorithm is capable of only detecting motion in 2D-co-ordinates it can't predict whether the moving object is closing is moving away from the robot. That is why a separate distance sensor is needed to monitor the moving object. The method could first calculate how many moving objects it founds in the room. Then it calculates the average co-ordinates for each of the moving objects. If we know the relative angle and location of the robot – and hence the camera – we can calculate the location of the object in the room and give this information to the distance sensor which could then monitor the object. Due to time limits I wasn't able to develop this kind of calculation to the algorithm.

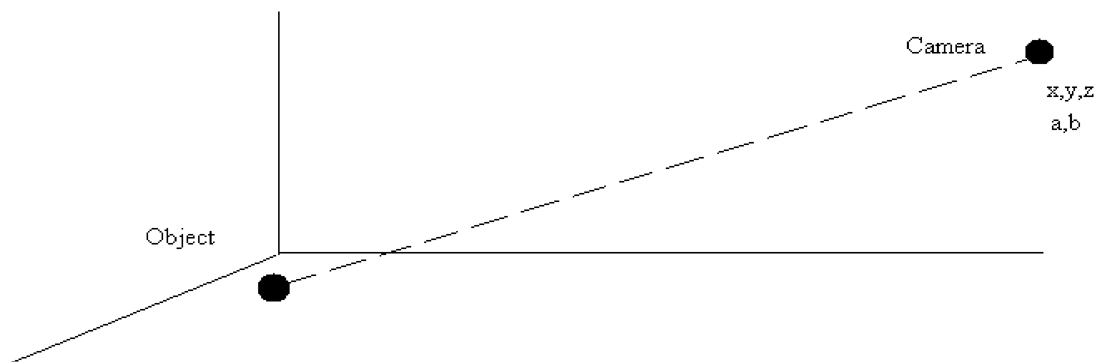

Image 16.

In the above image is an example how this calculation could be performed. Since we know the position of the camera (x, y, z) relative to a known zero point (some corner in the room) and the horizontal and vertical angles (a,b) of the camera we can calculate that the moving object is somewhere in the dotted line. Now based on this information the distance sensor can start to

monitor the location – starting from the dotted line - of the moving object.

GUI

I have also designed a GUI for the mapping team. However Juhana and I didn't have enough time to put our codes together so this job will be left for the next semester. The GUI has separate parts for the 2D and 3D mapping. The GUI looks the following (Image 15).

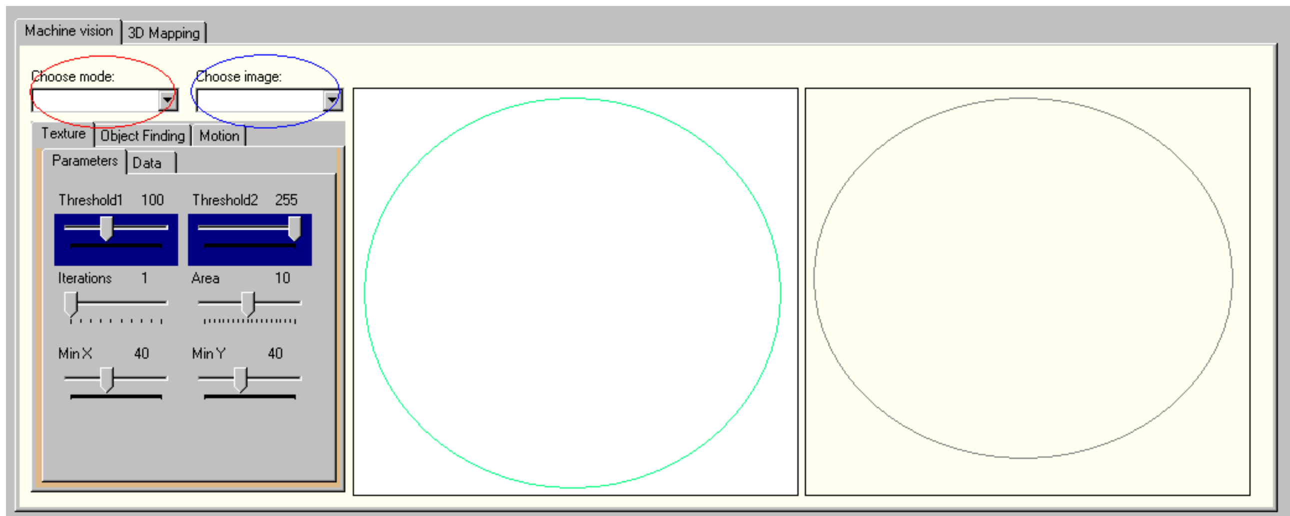

Image 17.

First of all one needs to be sure that a web cam is connected to the computer!! Otherwise the system won't run and error will occur. The mode will be chosen from ”Choose mode” combobox (red circle). It has four different modes: Texture segmentation, Object finding, Motion detection and Normal. In the normal mode only the normal camera image is shown and no mapping methods are performed. The resulting image will be shown in picturebox1 (green circle). When a mapping method is chosen one can choose two alternative images from ”Choose image” combobox (blue circle). That image will be shown in picturebox2 (gray circle, image 15). The purpose of this method is that the user is able to see pre-processing images from the methods in order to see that everything works as they are supposed to. For instance, if one chooses Texture segmentation as mode, he has two pre-processing images to choose from: image1 and image2. Image1 constitutes to an image right after cvPyrSegmentation and image2 constitutes an image right after the flood filling operation. These images help the user to change the parameters and see why the algorithm can't find

any specific objects (Image 16.).

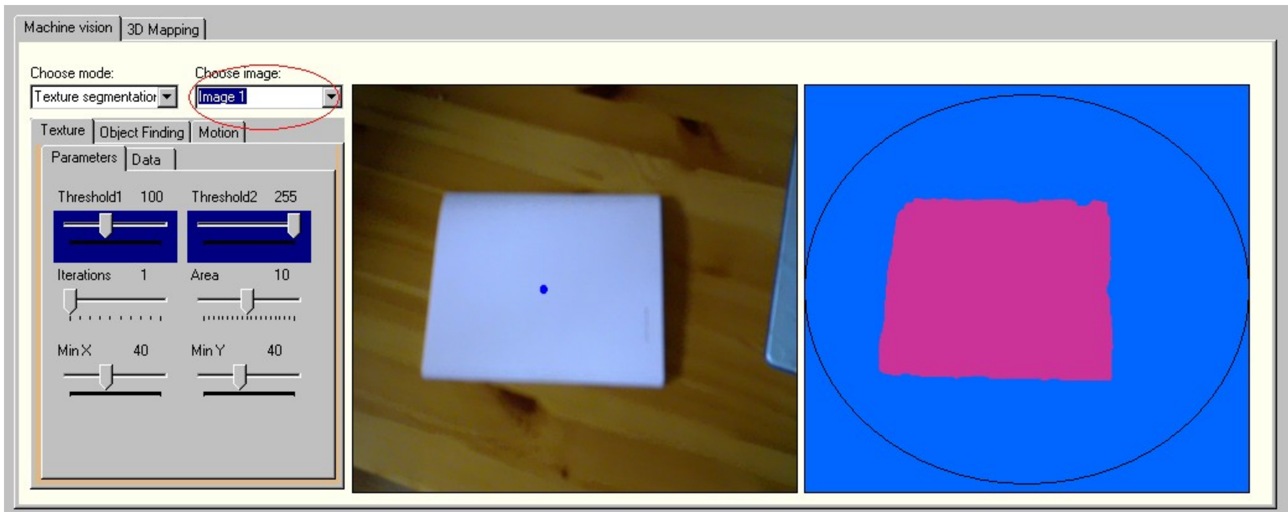

Image 18.

Each of the modes also has separate tab page where one can adjust parameters and see the results (Image 17 and 18). In image 17 one can see that there are three different tab pages (red circle): Texture, Object Finding and Motion. If one want's to alter the parameters or see the results the right tab needs to be chosen first. Each tab has two other tab pages inside it(blue circle): Parameters and Data. In the Parameters tab one can adjust the parameters the used method uses.

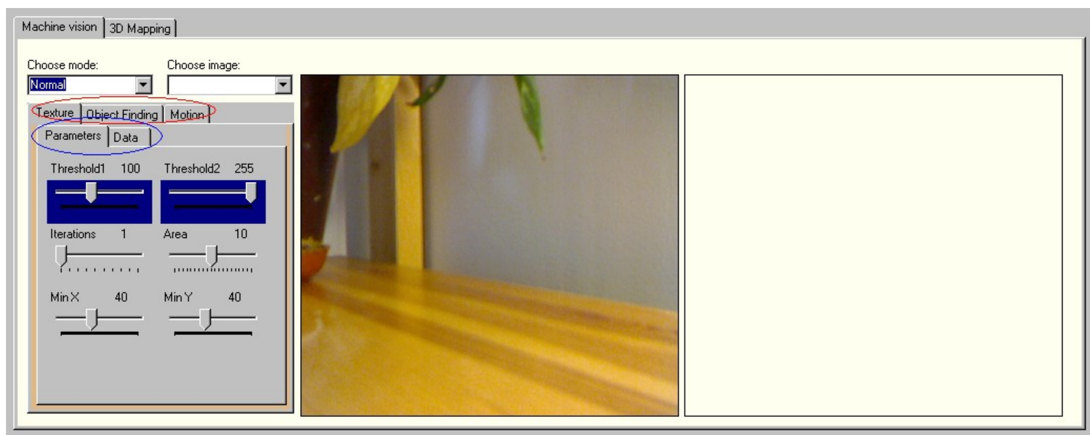

Image 19.

In the Data tab one can see the results – or co-odinates and the number of found objects (red circle in image 18). The co-ordinates list shows the location of every found object and displays them as x- and y-co-ordinates. The left upper corner of the image is the zero point (0, 0).

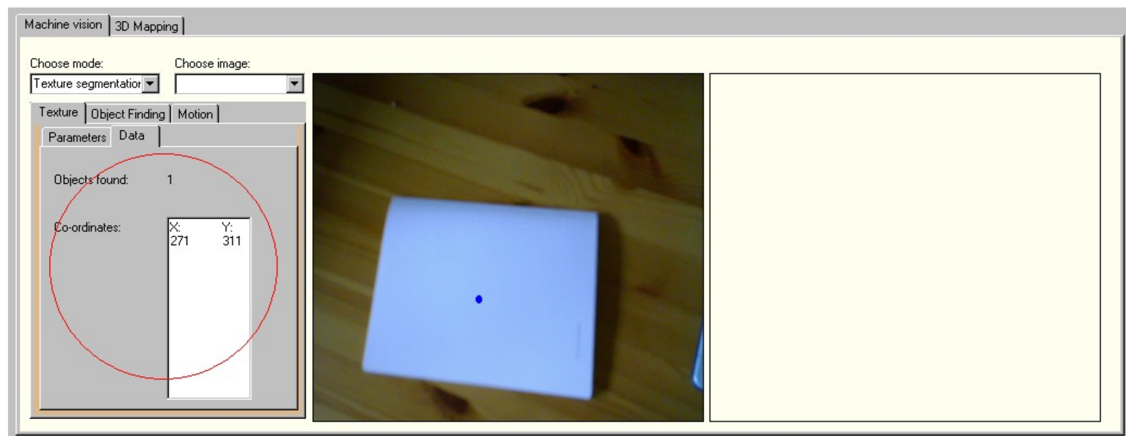

Image 20.

In image 19 one can see the parameters tab for texture segmentation. It has six different parameters. Next will be explained the meaning of each parameter. The parameters highlighted in blue are the most important ones.

- Threshold1 – error threshold for establishing the links in pyramid segmentation. The links between any pixel a on level i and its candidate father pixel b on the adjacent level are established if p(c(a),c(b)) < threshold1. After the connected components are defined, they are joined into several clusters.
- Threshold2 - error threshold for the segments clustering. Any two segments A and B belong to the same cluster, if p(c(A),c(B)) < threshold2.
- Iterations – how many times erode (morphological operation) will be used in the image after pyramid segmentation. This is used to remove the effect of noise.
- Area – this defines the size of the pixel area which will be checked for white pixels. It is used in FloodFill-function. Whenever a white pixel is found it checks a wider area to see that the found white pixel isn't just noise.
- Min X – the smallest difference between the found x-co-ordinates in function FindEdges. It basically removes every found object which x-co-ordinate differences is smaller than this parameter.
- Min Y - the smallest difference between the found y-co-ordinates in function FindEdges. It

basically removes every found object which y-co-ordinate differences is smaller than this parameter.

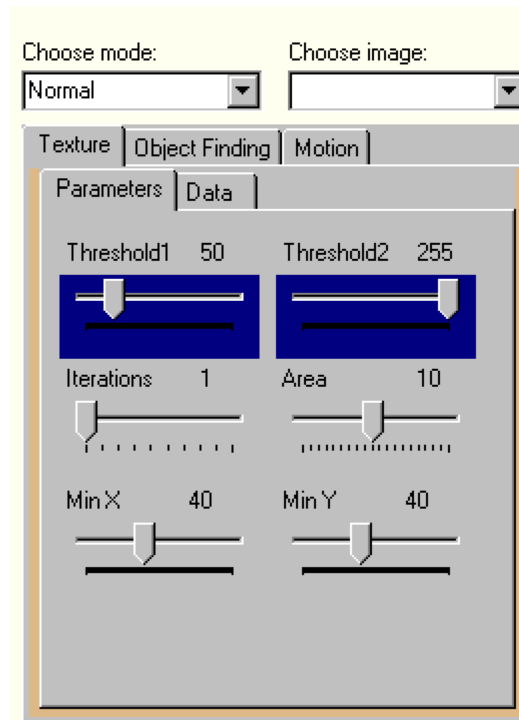

Image 21.

In image 20 one can see the parameters tab for object matching. It has four different parameters. Next will be explained the meaning of each parameter.

- Set percent – the value has to lie between [0 and 1]. Defines how many features the method has to find relative to the reference image.
- Set direction – the value has to lie between [0 and 360]. Defines how a big absolute angle difference there can be between the reference feature and a found feature.
- Set size – the value has to lie between [0 and 200]. Defines how a big absolute size difference there can be between the reference feature and a found feature.
- Set hessian – the value has to lie between [0 and 500]. Defines how a big absolute hessian difference there can be between the reference feature and a found feature.

Send button needs to be pressed to update the three previous parameters.

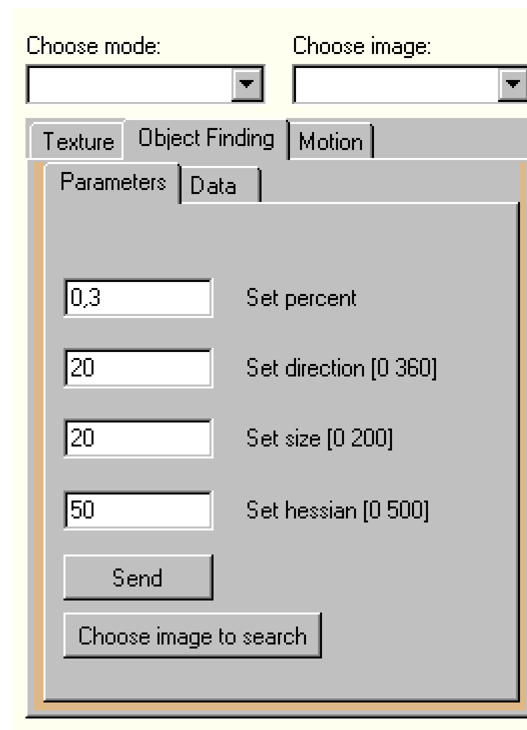

Image 22.

In image 21 one can see the parameters tab for motion detection. It has three different parameters. Next will be explained the meaning of each parameter.

- Threshold – the threshold value which will be used in thresholding the image right after calculating frame differences. Removes small moves.
- MinArea – the size of the found motion area needs to be larger that this.
- Time -defines the time value for udpateMotionHistory function. (explained in Motion Detection chapter)

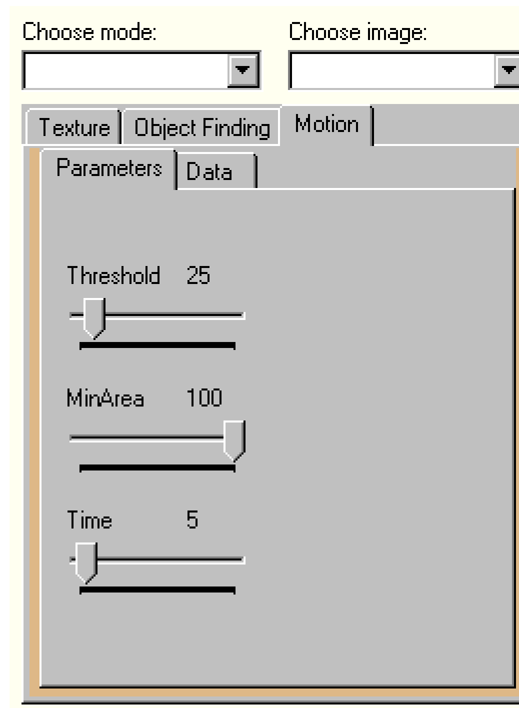

Image 23.

EXPLANATION OF THE CODE

The class diagram of the source code is depicted below.

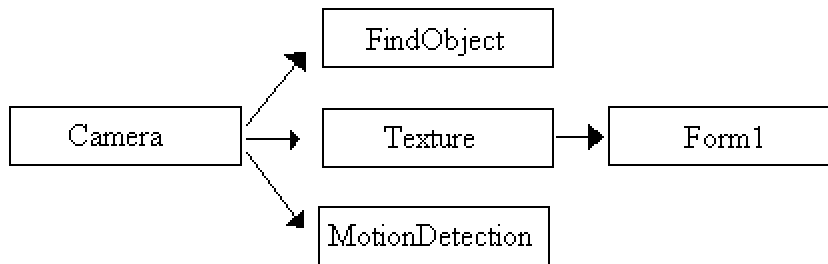

Image 24.

Camera class contains the code to acquire video image from the web cam. These frames are then given to FindObject, Texture and MotionDetection classes when these classes ask for it. The final class is Form1 which calls the methods from the three previous classes and shows the output.

Camera class has the following methods:

- Camera(void) – this is the constructor which initiates the camera image
- getImage(void) – this returns the next captured frame from the web cam used

- getSize(void) – this return the size of the video image

Texture class contains the following main methods:

- Texture(Camera* camera) – this is the constructor which inits all the Texture class variables, notice that the camera object is given as parameter for this constructor
- segment(void) – this method starts to segment the image, it basically just calls cvPyrSegmentation and adds some morphological operations
- process(IplImage* src, IplImage* dst) – this is where the actual processing happens, this method calls FloodFill and FindEdges to process the image. Parameter src is the source image and dst is the image where additional information will be drawn
- FloodFill(IplImage* src) – this method starts from the left upper corner of the image and goes through pixel by pixel the whole image. Since a white pixel corresponds a found object a floodfill operation is used to fill that area with a unique graysacale value. When a white pixel is found CheckWiderArea is first called
- CheckWiderArea(IplImage* src, int ref_x, int ref_y) – checks that a white pixel found by FloodFill function isn't just noise but there is larger area of white pixels nearby the found white pixel
- FindEdges(IplImage* src, int threshold) – the final method called by process. It finds the object corners – method described in Texture segmentation chapter, image 6 – and calculates the averages of those co-ordinates

MotionDetection has the following main methods:

- MotionDetection(Camera* camera) – the constructor which inits all the class variables. Notice that camera object is given to this constructor as a parameter.
- init(void) – initiates the images used in the class
- process(void) – starts processing the image. It first calculates the frame differences of two adjacent frames and thresholds the resulting image. After that it extracts contours from the thresholded image and finally draws all the found contours. Finally FindEdges is called to

find the co-ordinates for each of the found contour areas.

FindObject class contains the following main methods:

- FindObject(Camera* camera) – the constructor initiates the class variables. Notice that the camera object is given as parameter.
- Find(void) – this is the main method which calls other methods in processing the image. First image features are extracted using cvExtractSURF. After that it just compares the found features in video image and reference image.

I won't be going through the functions in the class Form1 since they are irrelevant in this report. However in the beginning of the Form1 class new objects from Camera, Texture, FindObject and MotionDetection classed are created and the camera object is given to the three latter objects as parameter.

INSTALLATION GUIDE

1. Download and install Visual Studio C++ 2008 Express Edition from http://www.microsoft.com/express/Downloads/#2008-Visual-CPP
2. Download and install OpenCV 1.0 from http://sourceforge.net/projects/opencvlibrary/files/
3. Download and install Microsoft Visual C++ 2008 Redistributable Package (x86) from http://www.microsoft.com/downloads/details.aspx?FamilyID=9b2da534-3e03-4391-8a4d-074b9f2bc1bf&displaylang=en
4. Download and install Microsoft Visual C++ 2005 Redistributable Package (x86) from http://www.microsoft.com/downloads/details.aspx?familyid=32bc1bee-a3f9-4c13-9c99-220b62a191ee&displaylang=en
5. Add OpenCV paths to Visual Studio. http://opencv.willowgarage.com/wiki/VisualC%2B%2B
6. Further information about openCV cen be found from http://opencv.willowgarage.com/wiki/
7. Install Logitech WebCam C300 drivers.

8. Add the Ceilbot GUI to Visual Studio Project folder.
9. Debug the program.

WORK FOR THE NEXT SEMESTER

Since there were two students participating in the development of the mapping algorithm our goal was to combine our codes. Since Juhana's code is in Java the first task would have been to translate it to C++. The next part would have been to add Juhana's part of the code to my GUI. In the GUI there is a blank tab page called 3D Mapping which is reserved for Juhana's code. The idea would have been that in that tab page Juhana could have added his part of the 3D mapping. Next task would have been to add get and set function for every variable Juhana would have wanted to alter in his GUI. Because we didn't have enough time to implement the combination, these tasks will be left for the next semester. The combination is a very crucial part that needs to be done!

I have also thought of combining the texture segmentation and object matching algorithms to make the object matching more robust. The basic idea would be first to use texture segmentation to segment the area and then use object matching only in those areas where texture segmentation has found objects. One other thing that must be implemented next fall is to make the texture segmentation more autonomous. One solution is to use a reference image as depicted in Texture Segmentation chapter or something similar idea. This kind of idea might help to overcome the brightness variation issues. Here is a short to-do-list for next semester:

- Translate Juhana's code to C++
- Construct the 3D mapping part GUI to 3D mapping tab page
- Combine texture segmentation and object matching
- Add intelligence to parameter adjustment in texture segmentation