Universidade do Minho
Escola de Engenharia

Pedro Emanuel Pereira Soares

Computer vision component to environment scanning

2011

Universidade do Minho
Escola de Engenharia

Pedro Emanuel Pereira Soares

Computer vision component to environment scanning

Dissertação de Mestrado
Área de Especialização Informática

Trabalho efectuado sob a orientação do
Professor Doutor Manuel João Ferreira

2011

Computer vision component to environment scanning

Pedro Emanuel Pereira Soares

December 27, 2011

Abstract

Computer vision is usually used as the perception channel of robotic platforms. These platforms must be able of visually scanning the environment to detect specific targets and obstacles. Part of detecting obstacles is knowing their relative distance to robot. In this work different ways of detecting the distance of an object are analyzed and implemented. Extracting this depth perception from a scene involves three different steps: finding features in an image, finding those same features in another image and calculate the features' distance. For capturing the images two approaches were considered: single cameras, where we capture an image, move the camera and capture another, or stereo cameras, where images are taken from both cameras at the same time. Starting by SUSAN, then SIFT and SURF, these three feature extraction algorithms will be presented as well as their matching procedure. An important part of computer vision systems is the camera. For that reason, the procedure of calibrating a camera will be explained. Epipolar geometry and the fundamental matrix are two important concepts regarding $3D$ reconstruction which will also be analyzed and explained. In the final part of the work all concepts and ideas were implemented and, for each approach, tests were made and results analyzed. For controlled environments the relative distance of the objects is correctly extracted but with more complex environment such results are harder to obtain.

Keywords: Computer Vision; 3D reconstruction;

Resumo

A visão por computador é, normalmente, usada como o canal de percepção do mundo em plataformas robóticas. Estas plataformas têm de ser capazes de rastrear, visualmente, o ambiente para detectar objectivos e obstáculos específicos. Parte da detecção de obstáculos envolve saber da sua distância relativa ao robot. Neste trabalho, são analisadas e implementadas diferentes formas de extrair a distância de um objecto.

A extracção desta noção de profundidade de uma cena envolve três passos diferentes: encontrar características numa imagem, encontrar estas mesmas características numa imagem diferente e calcular as suas distâncias. Para a captura de imagens foram considerados dois métodos: uma única câmara, onde é tirada uma imagem, a câmara é movida e é tirada a segunda imagem; e câmaras estéreo onde as imagens são tiradas de ambas as câmaras ao mesmo tempo. Começando pelo SUSAN, depois o SIFT e SURF, estes três algoritmos de extracção de características são apresentados, assim como os seus métodos de emparelhamento de características.

Uma parte importante dos sistemas de visão por computador é a câmara, por este motivo, o procedimento de calibrar uma câmara é explicado. Geometria Epipolar e matriz fundamental são dois conceitos importantes no que refere a reconstrução $3D$ que também serão analisados e explicados. Na parte final do trabalho, todos os conceitos e ideias são implementados e, para cada método, são realizados testes e os seus resultados são analisados. Para ambientes controlados, a distância relativa é correctamente extraída mas, para ambientes mais complexos, os mesmos resultados são obtidos com mais dificuldade.

Palavras chave : Visão por computador; reconstrução 3D;

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We can say that vision is a very, if not the most, important sense when we are moving in a certain environment. Providing crucial information about the environment, one relies, must of the times, only on vision to guide ourselves in a world full of obstacles. The importance of vision in animals and humans can also be seen in robots. In various robotic platforms, the vision system is the main perception device and robots must be able to rely on it to move and detect objects in the world.

When an image is captured by computer vision systems a lot of information is obtained but despite all that information, the depth of a scene point is not directly accessible. It is easily understood that knowing the 3D localization of an object in space is very important when talking about a moving robot. If there is no 3D information the robot that, for example, finds an obstacle in its path does not know if the obstacle is near and it needs to be avoided or if it is far away and there is no need to avoid it at the moment.

In this work we will discuss and implement different approaches to the problem of extracting 3D information from a scene. Different approaches to the image processing step will be analyzed and the results compared. When extracting 3D information we will test two different approaches, single and dual camera system.

1.1 *Motivation*

Focusing merely on vision, the capability that the brain has of extracting information from what our eyes see is simply amazing. There are computer vision systems that are faster and have better performance in certain tasks, but generally, we are still faster and better. One area where the human brain exceeds computer vision is the detection of objects and 3D information extraction. By looking at a room we know, for example, which objects are closer to us and when walking, if see an object, in this case an obstacle in our way, we know if have to move around it or if it is still far away. Regarding moving robots, this vision capability would be a very good thing to have.

In computer vision when an image is obtained from a camera we only have 2D information, that is, x and y position. With this information we can say that a certain point is more to the left/right or higher/lower but we cannot say if it is closer or further from us. Correctly extracting the third coordinate z, allowing us to extract 3D information from a scene, is the objective of this work. Having correct 3D information from a scene will give the robot better moving abilities.

1.2 *Objectives*

In order to complete our main objective of extracting 3D information from a scene, three steps must be accomplished.

- **Point Extraction** - Given images we should be able to extract defining points.
- **Point Comparison** - We must be able to find the same points in two different images.
- **3D Reconstruction** - From the information received from previous steps we have to extract 3D information.

1.3 *Structure of Report*

In Chapter 2 we will talk about the first step of the 3D reconstruction, image processing. In this chapter we will analyze different approaches to the problem of extracting and comparing points in an image. The process of 3D reconstruction will be explained in chapter 3. Starting by explaining why and how to calibrate a camera following on what is the fundamental matrix and what is it used for, and, finally the triangulation. After explaining the ideas and methods used, we will show how this methods were implemented and tested and the results obtained in each method in chapter 4. We will finish this work with a conclusion, explaining the results obtained and the problems encountered.

Chapter 2

Image Processing - point extraction

2.1 Introduction

As mentioned before an image can only give us 2D information. When we look at a photo, for example, we can tell which objects are closer or further away, but, this perception is not only based in the information of the photo, one needs extra information to accurately perceive the distances. One of that extra information is the common size of the object or, by other words, what we think that the size should be. Let us consider the following situation: an oversized beer can and a small car model, in a proportion that gives them similar sizes. If we place the can and the car next to each other and take a photo what will we see? We will see a can and a car with the same size and the brain will assume that the can is closer to the camera (since we know that a can should be smaller than a car).

So, how can we extract 3D information from images (without knowing the size of objects)? We need at least two images of the same scene from different perspectives. This can be done in two ways: with two cameras (stereo correspondence) or with one moving camera (structure from motion). Usually the more cameras being used the better: this project will use two cameras. Even humans need two images to extract 3D information. If we close one eye, after a while, we will lose the 3D perspective, unless we move therefore obtaining images from different perspectives.

The basic idea of 3D extraction from images is matching them and computing the differences. Image matching is a very important step in many computer vision problems such as solving 3D structure from multiple images an stereo correspondence. Matching two images is usually done by finding distinctive points (features) in one and finding the same points in the other. When talking about image matching for 3D reconstruction, especially in moving environments, it must be taken into account that the camera will move and lighting conditions will probably change as well. For that, the image matching algorithm must be prepared to find and match features even in those conditions. In image matching finding the features is

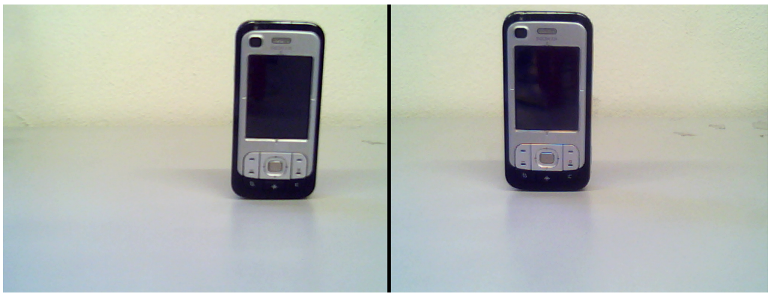


Figure 2.1: Same object from two different perspectives

not a simple procedure. One can have lots of features but if finding them again in the second image is impossible, we will not be able to match both images. An algorithm must provide us a way of differentiating a feature from another (i.e. a descriptor) so we can know if the point we find is the same that we found before.

When classifying an feature detection algorithm for image matching there are some criteria for determining its quality:

- **Quality of detection**
  When finding features in an image the algorithm should produce very few false positives, usually originated from noise.
- **Consistence**
  If an image is tested several times the same feature points should be detected.
- **Speed of detection**
  As for most types of algorithms, the time it takes to produce results should be minimum.
- **Quality of descriptor**
  As mentioned before, what use do we have for lots of points if we can't correctly find and match them in another image.
- **Speed of matching**
  In image matching algorithms a relatively big portion of time is spent in comparing and matching points.

2.2 SIFT-features Algorithm

Scale-invariant feature transform (SIFT) algorithm proposed by Lowe in [22, 23] is a commonly used algorithm used in computer vision to detect local features in images.

Scale-invariant feature transform, as the name suggests, focus on features that are invariant to scaling and rotation and also partially invariant to change in illumination and 3D camera viewpoint. SIFT algorithm also performs rather well when occlusion, clutter, or noise occur in an image.

Fortunately, SFIT found features are also very distinctive making them relatively easy to match in other images. Large numbers of distinctive features can be efficiently extracted from images and then matched with SIFT making it very useful in image matching. In the next section we will present the normal use of the SIFT algorithm.

2.2.1 Steps

In order to generate a set of distinctive scale-invariant image features several steps must be taken. We can organize the process in four major steps. In early steps of the process several points are found and if we were to apply the most expensive operations of the algorithm to all points the algorithm's efficiency would be compromised. So in order to increase efficiency, initial tests are made to the points to determine which ones are worth passing to the next stage. By doing so we guarantee that the most expensive operations are applied only to those points that pass initial tests minimizing the cost of the algorithm.

Interest point localization

The first stage of the SIFT's algorithm is finding scale invariant points, called interest points, so they can be examined in further detail in the next steps. This is done by searching over all scales and image locations and selecting those locations that are stable across all scales. A *difference-of-Gaussian* is applied to obtain different scales.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.1}$$

Function 2.1 defines a scaled image $L(x, y, \sigma)$ obtained from applying to an image $I(x, y)$ a scale variable $G(x, y, \sigma)$. When 2.1 is applied to an input image I we get a different scaled image L.

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{2.2}$$

Function 2.2 defines de difference-of-Gaussian (DoG) between two images where one has a k times bigger scale. After the DoG is applied each point of the image is compared with both neighbors at different scales. Considering a point in a scale k there are 8 neighbors in the same scale and 9 in the $k - 1$ and $k + 1$ scales making a total of 26 neighbors. If the point is either a minimum or maximum of all points the point is considered an interest point.

Key point localization

In the first, step several interest points were found but, to increase efficiency, the number of points must be reduced. This is done by selecting key points based on their stability. Scale and location is determined to each point and those points that have a low contrast (therefore sensitive to noise) are rejected. First for each interest point, its position is determined by interpolation of nearby data, this is done using the quadratic Taylor expansion of the DoG which gives an offset. If the offset is larger than 0.5 the point is close to another interest point so we will change to that point and an interpolation is performed. If the offset is equal or less than 0.5 it will be added to the interest point being analyzed giving an approximated localization. Using the Taylor expansion again, if the offset is less than 0.03 the interest point is discarded for having a very low contrast. Low contrast points are very sensitive to light alterations and therefore, very unstable. Because DoG function is very sensitive to edges, resulting in a lot of interest points in edge lines, points that have poorly determined location but high edge response must be eliminated. After this step all points are now considered key points.

Orientation assignment

This is the step where rotation invariance will be achieved. To each key point one or more orientations will be assigned based on local image gradient directions. The invariance to rotation is achieved because all future operations will be performed relative to the assigned orientation. To find the correct orientation, an histogram of gradient orientations is created based on the point's neighbor. The peak orientation of the histogram is assigned to the point.

Key point descriptor

From the previous steps the algorithm was able to achieve scale, rotation and location invariance. This step aims to achieve illumination and 3D viewpoint invariable. This is done by assigning a highly distinctive descriptor to each key-point. The key point descriptor is created from magnitude and orientation gradients of the area around the key point. This step is performed on the closest scale image.

Key point matching

As we told before, for image matching we need more than scale-invariant features found in an image. If we cannot match the features found in an image to those found in another image, matching is not possible. SIFT gives a highly distinctive descriptor for each key point, making it a reliable method even when large features databases are being used. Normally when features are extracted they are stored in various ways and when a new image is captured and its features extracted, each of the new points must be compared to stored points for a match. [23] shows a good reliability even when large databases are used. Two important determinants of SIFT's algorithm performance are: how the points are stored and how the search is made. Searches in large databases can sometimes be a bottleneck to algorithms performance. KD-Tree [14] is a normally used data structure that allows for points to be organized in space according to the value. This kind of data structure is used in applications where we have to find the values closest to a given value. As in any tree-organized data structure any search for a given value is much more efficient than regular searches.

2.3 SURF-features Algorithm

Speeded Up Robust Feature (SURF) [1] [2] is another feature extraction and matching algorithm also intended to be scale and rotation invariant. Although it is based on SIFT, it is said be faster and have better performance against image transformations. One of the reasons of it's speed is the use of Integral Images as an aid structure throughout the rest of the process. It is also based in Hessian detectors, as it was done by Lindberg [20], for the extraction of interest points, and Haar Wavelet [10] responses as the points descriptors. As for SIFT, SURF as two important stages: interest point detection and point descriptor matching.

2.3.1 Interest point detection

As in many other algorithms, we must start by finding interest points.In SURF this is done in four sub-steps starting by using Integral Images as a helpful way to speed things up. Using a Hessian detector we then find interest points across the image and construct a scale space representation. The final step in interest point detection is to use non-maximum suppression.

Integral Images

Integral Images or Summed Area Table first appeared in [6] and have become very useful in box type convolution filters. When for each pixel we compute something based on a mask, such as circular mask in SUSAN, we have a convolution filter. In order to use integral images, the filter mask must have a box shape because Integral Images use rectangles as the base of calculation.

$$I(x, y) = \sum_{i=0}^{i'<x} \sum_{j=0}^{j\leq y} I(i, j) \tag{2.3}$$

From 2.3 we can see that for each pixel $A = (x, y)$ it's integral image result will be the sum of all pixels from $(0, 0)$ to A. Therefore we can say that integral images represent the sum of all pixels within a rectangle defined by $(0,0)$ and (x,y). The reason why Integral Images are so useful is that for any pixel, we can find the sum of intensities of the surrounding area with just three operations and four memory accesses, as showed in figure 2.2. The time in which we can find a result in an Integral image is independent of filter size, this is very important as SURF usually uses big filters

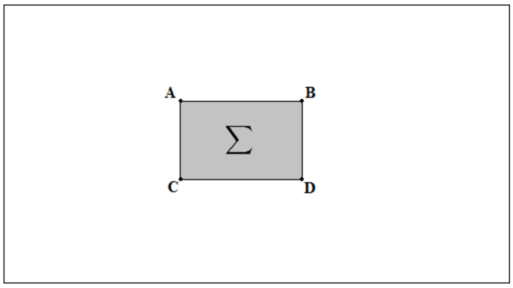


Figure 2.2: Integral Image sum of intensities $\sum = I(A) + I(D) - I(B) - I(C)$

Moreover Integral Images are relatively easy to compute as we only need one pass over the image.

$$I(x,y) = i(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1) \tag{2.4}$$

The fact that Integral Images are fast to compute and make finding the sum of intensities an easy and fast task, is one of the reasons for SURF speed.

Hessian Detector

The previous step was only to give us a way to speed calculations involving sums of intensities. Now we will start the process of finding those points using an Hessian-based detector for every pixel.

$$H(X,\sigma) = \begin{bmatrix} L_{xx}(X,\sigma) & L_{xy}(X,\sigma) \\ L_{xy}(X,\sigma) & L_{yy}(X,\sigma) \end{bmatrix} \tag{2.5}$$

In equation 2.5 $H(X,\sigma)$ defines the Hessian matrix at point X where $L_{ab}(X,\sigma)$ is the Gaussian second derivative in the a and b directions. These derivatives are computed for the smoothing scale σ, which has the same meaning as in equation 2.1 in Section 2.2.1. Box filters are used to approximate the results of these second order Gaussian derivatives and can be evaluated using integral images at low cost. For a real valued Gaussian with $\sigma = 1.2$ the box filter that approximates result will have a size of $9 * 9$.

In a Hessian matrix the value of its determinant is used to classify the maxima and minima by the second derivative test. In SURF the determinant's sign is used to classify the point.

$$det(H) = L_{xx}L_{yy} - (wL_{xy})^2 \tag{2.6}$$

For each pixel we can calculate the determinant and use it to classify the point. We can consider a point as an extremum if its discriminant is positive. We refer to this determinant as the blob response and all responses are stored in a blob response map over different scales. They will later be used to locate interest points.

Scale Space representation

In order for the algorithm to be able to detect the same point over different scaled images, we need to search for interest points across the different image scales. We therefore need a structure that allows us to represent this scale space. In computer vision pyramids are used to implement this scale space. Starting, with the original image at the base level of the pyramid we smooth, using Gaussian, and sub-sample (reduced in size) it to get to the next level of the pyramid. This method of creating an image pyramid is used in SIFT but, with the help of integral images and box filters, this computationally heavy step is improved in SURF.
The use of integral images and box filters allows SURF to apply box filters of any size directly on the same image with a constant speed, instead of reapplying the same filter to a previously filtered image. This is a considerably big difference between image pyramids used in SIFT since we upscale the filter size rather than reduce the image size. As mentioned before "normal" image pyramids are computationally heavy, on the other hand, SURF image pyramids are not. This is another reason why the SURF algorithm is faster.

In SURF, this image pyramid is divided into octaves and each octave refers to a series of response maps. These octaves are then subdivided into a constant number of scale levels. For the first level of the pyramid, a $9 * 9$ sized filter is used and for subsequent layers the filter is up-scaled. At any iteration the next scale the window has to increase a minimum of 2 pixels, since we always need a center pixel.

Points Location

Now that we have the image pyramids constructed for each pixel, we have to find those that are invariant to scale and rotation and therefore points of interest. First we start by using a threshold to eliminate all points that have responses below the predetermined threshold. With this we can control the quantity of points detected and we can choose to increase the threshold and, by this, allowing only the most invariant points to continue. Once all the points are threshold, a non-maximum suppression, as explained in section 2.4.1, is done to find the candidate points. Considering that an image is on a layer of the pyramid and there are two more layers next to it (each one of the scales above and below), we can say that a pixel has a total of 26 neighbors as illustrated in figure 2.3.

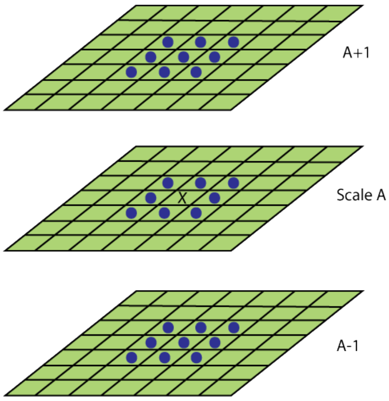


Figure 2.3: Non-maxima Suppression.

Considering point X in figure 2.3, during non-maxima suppression, the point will be considered a maxima if it is greater than his neighbors, represented as blue balls.

After this two steps we have a set of points that are both higher than the threshold and local maxima. From here we interpolate the points using the method purpose by Brown and Lowe in [5] resulting in a set of detected features.

2.3.2 Interest point description and matching

Now that we have a set of points found in an image, we need to find the same points in a different image. Since the same method of finding points will be used in both images, chances are that points found in the first image, will be found again in the second. But there is still the question of knowing if it is the same point or not. As mentioned before we have to find a way of describing the points (i.e. a descriptor). This descriptor describes the content of the area around the pixel in terms of intensity.

Orientation assignment

As said before an image feature should be invariant to image rotation. In SURF, as in SIFT, this is done by identifying an orientation for each interest point. Like explained in section 2.2.1 all future operations will be performed relative to the assigned orientation, therefore achieving orientation invariance. Instead of gradient histograms, as used in SIFT, SURF uses Haar wavelet to find the assigned orientation. One of the uses for the Haar Wavelet is to find gradients, both in x and y directions.

To facilitate the explanation of how an orientation is found, we will divide the process into three steps. For each point we:

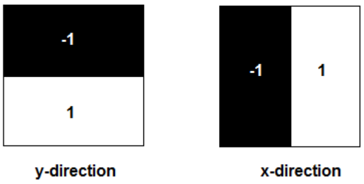


Figure 2.4: Haar Wavelets.

- **Calculate wavelets**
  To do this, we start by choosing a circular area around the point. This area has the radius size of 6σ, where σ is the scale in which the point was detected. For all points inside the area we calculate the wavelet responses. The wavelets used have a size of 4σ. Once again the integral images came in handy as we only need six operations to compute the response in x or y direction at any scale.

- **Map responses**
  All wavelet responses are weighted with a Gaussian centered in the interest point. Again the scale, in which the point was detected, is taken into account and a deviation of 2.5σ is used. With all responses weighted we map them in a $2D$ space.

- **Sum wavelets**
  Imagining a circle in the $2D$ space centered at the origin, we divide this circle in 6 slices. For each slice we sum all responses within, resulting in a response vector. After all 6 vectors are calculated, the longest one is chosen as the interest point orientation. The size of the slice, in this case $\frac{\Pi}{3}$ can be altered although caution must be taken.

This process is exemplified in figure 2.5.

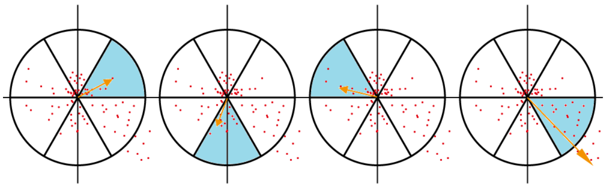


Figure 2.5: Wavelets Sum. In this case the assigned orientation would be the one in last image.

Haar wavelets based Descriptors

The process of calculating a descriptor for each point starts by constructing a square region centered in the point along the point's orientation. This square has a size of 20σ and contains the all pixels which will take part in the calculation of the descriptor vector. The square is then divided into 16 $(4 * 4)$ sub-regions and for each one, we calculate the Haar wavelets (size 2σ), responses obtaining this way a descriptor vector.

$$Descriptor_{subregion} = [\sum dx, \sum dy, \sum |dx|, \sum |dy|] \tag{2.7}$$

From equation 2.7 we can see that each sub-region will have a sized 4 vector containing dx dy which are the wavelet responses in x and y respectively and they're absolute values $|dx||dy|$. Note that these directions are defined in relation to the point's assigned orientation. Therefore we have 4 descriptor values for each sub-region with a total of 16 sub-regions making a total of 64 values. These 64 values are the entries of the point's descriptor vector. Another version of SURF called SURF-128 doubles the number of descriptors values. It does this by splitting the sum according to their sign, we therefore get two descriptors for each of older ones as we can see in 2.8.

$$\begin{split} Descriptor_{subregion} = [\textstyle\sum dx < 0, \sum dy < 0, \sum |dx < 0|, \\ \textstyle\sum |dy < 0|, \sum dx \geq 0, \sum dy \geq 0, \sum |dx \geq 0|, \sum |dy \geq 0|] \end{split} \tag{2.8}$$

If we double the descriptor values with the same 16 sub-regions we get $8 \times 16 = 128$ values. Having a descriptor with twice the number of the values, we get a more distinctive descriptor. This reduces the number of *mismatches* but, on the down side, even though the computing time is not much slower than normal, the matching stage is much slower.

Indexing for matching

In order to help in the process of indexing and matching, SURF uses the sign of the trace of the Hessian matrix, computed in the detection phase, as a way to differentiate points. The sign of the Laplacian (trace of the Hessian matrix) allows us to distinguish between bright blobs on dark backgrounds and the reverse. This simple and "free"(already computed) information gives a way of differentiating the points on a high level, if the points have different signs they are not a match. It is easily understood the advantages this gives us, at the indexing phase we can create 2 areas and when searching for a match we only have in one of the areas. Even in more complex indexing systems such as K-d trees we can split the search area therefore improving the matching time.

2.4 SUSAN Algorithm with Kirsch

In this section we will explain the methods used to attempt the extraction and matching of points. This method starts by finding points using SUSAN's corners and giving each point a descriptor based on Kirsch Gradient.

2.4.1 SUSAN Corners

Smallest Univalue Segment Assimilating Nucleus (SUSAN),proposed by Stephen M. Smith. [26], is a commonly used algorithm in computer vision to extract features from an image. SUSAN, as many other algorithms, follows the method of applying to each pixel's neighboring area a set of rules and getting a response that defines the pixel. This area, in this case, is a circular window and all pixels inside that area are compared in terms of brightness to the nucleus (center of window). This area of similar pixels is called "Univalue Segment Assimilating Nucleus" (USAN) and is later used to determine if the pixel is a corner. Image matching requires that a number of features are extracted from an image. When using SUSAN corners as the extractor of features, we assume that corners are the most important points. When we say that a point is the most important it means that it is an easily detectable point in a sequence of images or images from different perspectives.

The principle

The basic concept of SUSAN is having, for each pixel, a local area of similar brightness, the USAN. The USAN contains important information about the area surrounding the pixel. Considering the area of the mask window and the area of the USAN we can see in figure 2.6 why we can say that if a pixel is in a corner or edge the area of the USAN it will always be less than half the size of the window.

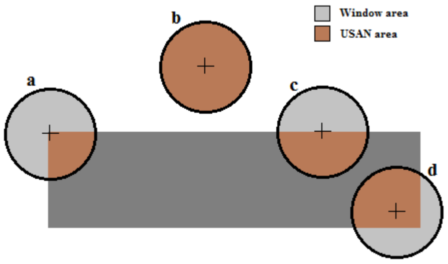


Figure 2.6: USAN area

In 2.6b we can see that when the nucleus is in a flat area (no corners or edges) the USAN area is at a maximum. If the nucleus lies in an edge the area will be half 2.6c and the sharper the corner the less the area of the USAN will be2.6a. The area of the USAN will decrease as an edge is approached reaching a minimum at the correct location of the edge. Looking at the relation between the area of the USAN and where the pixel is, we can say that corners will be the minimum responses to the algorithm.

How it works

Now that we understand the principle behind SUSAN algorithm we will describe the work flow of extracting corners from an image. This process can be divided in five steps.

- **Place a circular mask in each pixel.**
  For each pixel we start by centering a circular mask window, in most cases with a size of 37 pixels, and we compare the nucleus brightness with all the pixels inside the mask. Originally the following equation 2.9 was used to determine if a pixel had an acceptable brightness, meaning, if it's brightness was similar to the nucleus.

$$c(\overrightarrow{r},\overrightarrow{r_0}) = \begin{Bmatrix} 1 & if|I(\overrightarrow{r}) - I(\overrightarrow{r_0})| \leq t \\ 0 & if|I(\overrightarrow{r}) - I(\overrightarrow{r_0})| \geq t \end{Bmatrix} \tag{2.9}$$

  $I(\overrightarrow{r})$ and $I(\overrightarrow{r_0})$ are the brightness's of any pixel and the nucleus respectively, c is the output comparison and t is the brightness difference threshold. With t we can control the acceptable differences in brightness within the USAN. If we increase t, therefore allowing greater differences in brightness, the algorithm will consequently give a greater number of responses. We can say, then, that t allows to control the number of responses. In 2.10 we can see later version of 2.9.

$$c(\overrightarrow{r},\overrightarrow{r_0}) = e^{-(\frac{I(c(\overrightarrow{r}))-I(c(\overrightarrow{r_0}))}{t})^6} \tag{2.10}$$

  With 2.10 we get more stable and sensitive results. While 2.9 gives us binary results, 1 or 0 depending on the threshold, with 2.10 we get continuous results between 0 and 1. Because of this, with 2.10, small variations to a pixel's brightness will not have major effects on c.

- **Define the USAN.**
  Now that we have a way of defining which pixels belong to the USAN, we have to calculate the number of pixels that belong to the USAN. For that we use:

$$n(\overrightarrow{r_0}) = \sum_{\overrightarrow{r}} c(\overrightarrow{r},\overrightarrow{r_0}) \tag{2.11}$$

  Applying 2.11 to each pixel within the mask we get n, the area of the USAN.

- **Geometric threshold.**
  As mentioned before, we can define a pixel has being in a corner based on the relation between the area of the window and the size of the USAN. The corner response, which defines "the level of corner" that a pixel is in, can be defined as:

$$R(\overrightarrow{r_0}) = \begin{Bmatrix} g - n(\overrightarrow{r_0}) & ifn(\overrightarrow{r_0}) < g \\ 0 & otherwise \end{Bmatrix} \tag{2.12}$$

  $R(\overrightarrow{r_0})$ being the initial corner response, is determined by comparing the area of the USAN, $n(\overrightarrow{r_0})$, with a geometrical threshold g. We now have two different thresholds, g and t, while t mostly affects the quantity of responses, g also affects the quantity but most importantly it also affects the quality of the corners. We can define the quality of a corner as it's level of sharpness, for lower values in g only sharper corners will be detected. From figure 2.6 we can define certain values for g to get different results. Let us consider n_{max} the maximum value that n could take, this happens when we are looking at a flat colored area and n will have the same size as the mask window. If we consider $g = n_{max}$ in 2.10 all points would have positive results, for $g = 3n_{max}/4$ only pixels that are on edges or corners. Since an USAN size includes the nucleus, an edge will always have $n > n_{max}/2$ therefore we can say that for corners, g can be defined has $n_{max}/2$.

- **Eliminate false positives.**
  Before we can declare a point as a corner we have to take into account that false positives will be produce. In images extracted from the real world, this can occur where the boundary limits of the image are blurred. In order to eliminate this problem a simple method was implemented and applied the points found in the previous step. If we find the center of gravity of the USAN and then the distance from it to the nucleus, we can reject those which have the center of gravity near the nucleus. A USAN from a corner will not have it's center of gravity near the nucleus. Another problem in real images is the noise. To overcome this problem SUSAN implements a contiguity rule which states that all pixel's, within the mask, in the direction of the center of gravity starting in the nucleus must be part of the USAN. By doing so we ensure that a corner must be part of a bigger part of the image thus eliminating single pixel's usually created from noise.

- **Find corners.**
  The final step on SUSAN corners algorithm is to find the correct location of the corners. At this step of the algorithm, we have several points in the image which are possible corners. A corner will not have only one response in it's area, it will have

all the points which were validated in equation 2.12. But which one of these points is the correct point? As we can see in equation 2.10 the response is 0 if $n(\overrightarrow{r_0}) \geq g$ and $g - n(\overrightarrow{r_0})$ otherwise, which means that the smaller the USAN area is, the bigger the response will be. It is easily understood from figure 2.7 that the closer the nucleus is to the tip of the corner the smaller its USAN area will be and therefore the bigger the response.

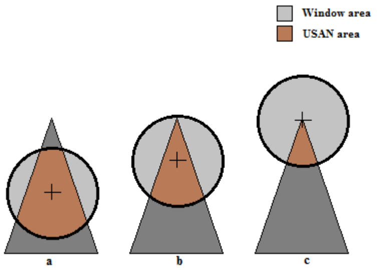


Figure 2.7: Corner USAN area

Because of this, we can state that the corner will be a local maximum in the image. Therefore, to find corners we have to find local maximums in the response image. To do so SUSAN uses a non-maximum suppression. Non-maximum suppression is an algorithm that suppresses all image information that is not part of local maxima, this is done using the gradient direction and will result in a set of corner points.

2.4.2 Kirsch Edge Detector

The Kirsch algorithm [18] is used to detect edges in an image using eight compass filters. Each filter is the gradient of the neighboring area of the pixel in a determined direction. For each direction, 8 in total, we apply the filter to the image and the one which has the highest response is considered as the pixel's orientation. We then can use this maximum response to create a final image that will represent the edges of the contours.

The algorithm uses 3*3 windows for each pixel one for each direction:

+3	+3	+3	+3	+3	+3	-5	+3	+3	-5	-5	-3
+3	0	+3	-5	0	+3	-5	0	+3	+3	0	+3
-5	-5	-5	-5	-5	+3	-5	+3	+3	+3	+3	+3

-5	-5	-5	+3	-5	-5	+3	+3	-5	+3	+3	+3
+3	0	+3	+3	0	-5	+3	0	-5	+3	0	-5
+3	+3	+3	+3	+3	+3	+3	+3	-5	+3	-5	-5

When allying these filters to an image, if the pixel has a strong response it means that the pixel is in an edge. Considering grayscale images, we can say that a pixel belongs to an edge if there is another pixel next to it that has a very different level of color. If we apply the gradient to an area with the same color, not an edge, the window response will always be 0, since the sum of multipliers in the window is also 0 (ex: for the first window $3 + 3 + 3 + 3 + 0 + 3 - 5 - 5 - 5 = 0$) and $0 * X = 0$. In the other hand, if we consider an edge where there is an area where the pixel's color is very different, the window response will be as high as that difference in color. Apart from giving the edges, Kirsch algorithm can also be used as a descriptor for a pixel. We can say that the descriptor for the pixel is an list of the 8 responses to the filters. When comparing the same world point in two different images chances are that pixel will have the same response to the filters.

2.4.3 All together

As mentioned before, in order to extract 3D information from a scene the first step is image matching, and for that we need points in both images and a way to match them. With SUSAN corners providing us with points we can use the Kirsch gradient as a way to give each point a descriptor to compare and match them. When find a match for the point we have to search all the other image points for the one that has a list of Kirsch values closer to the point.

Chapter 3

3D Reconstruction

3.1 Introduction

In this work, when referring to 3D reconstruction, we mean the process of extracting 3D information from scene and interpreting it. We do not really reconstruct a scene based on images since that is not the objective of the work. The objective is to know where, relatively to us, the object is. As mentioned before the depth of a scene point is not directly accessible in 2D images. With at least two 2D images, on the other hand, triangulation can be used to measure the depth of the scene. In order to capture two images from the scene there are two different approaches in robotics, stereo and motion analysis systems. Stereo systems, as the name suggest have two different cameras, normally placed side-by-side with different angles(this way increasing the common viewing area in closer distances). As for motion analysis systems there is only one camera which means that both images needed for triangulation must be obtained with the camera in different positions. This process is normally called *structure from motion*.For this section the most important references are [8] and [11].

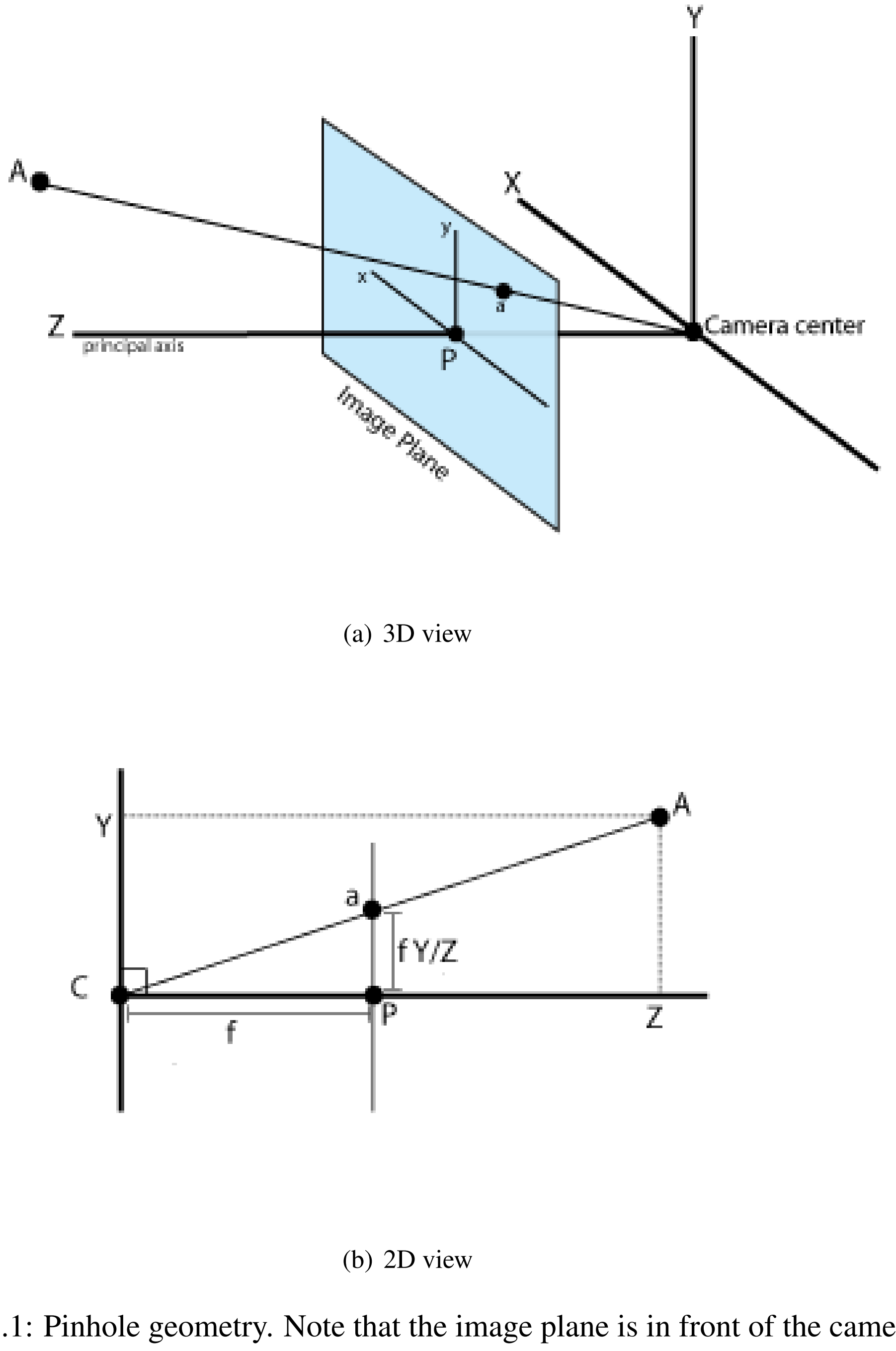
3.2 Camera calibration

Cameras are obviously one of the most important parts of computer vision systems and have a big influence in the system's performance. In order to, correctly determine the $3D$ information of a scene, one must know exactly how an object in $3D$ space is viewed in the image. We can say that the camera maps $3D$ world points to $2D$ image points and knowing how the mapping is done is the objective of camera calibration.

Every camera has it's own parameters therefore, pre-determining and using them with every camera is not a valid solution. As we will show in this section, we can determine these parameters with some mathematics and known patterns.

3.2.1 Camera Matrix

Before we introduce the concept of camera matrix let us start by explaining how an object is viewed by a camera. How $3D$ points are mapped in the image is determined by the camera model. The simplest and most commonly used in computer vision, is the pinhole camera model [9] also known as projective camera. In this model we consider the central projection of $3D$ space points onto an image plane. This geometry is represented by figure 3.1(a).



(a) 3D view

(b) 2D view

Figure 3.1: Pinhole geometry. Note that the image plane is in front of the camera

In figure 3.1(a) let us consider $\boldsymbol{A} = (X, Y, Z)^T$ as a point in the $3D$ space. In the pinhole camera model, $Z = f$ and the plane that this defines is called the *image plane* or *focal plane*. The line from $\boldsymbol{C}$ perpendicular to the image plane defines the principal axis and it's intersection with the image plane defines the principal point p. The intersection between the line that joins $\boldsymbol{C}$ to $\boldsymbol{A}$ and the image plane is where $\boldsymbol{A}$ is mapped, in this case represented by $\boldsymbol{a}$. From figure 3.1(b) we can determine that the point $(X, Y, Z)^T$ is mapped to $(fX/Z, fY/Z, f)^T$ in the image plane. Considering only the coordinates in the image plane, we have

$$(X, Y, Z)^T \rightarrow (fX/Z, fY/Z)^T \tag{3.1}$$

Equation 3.1 describes the mapping from $3D$ to $2D$ coordinates. We can also express this mapping using matrices:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} fX \\ fY \\ Z \end{pmatrix} = \begin{bmatrix} f & & & 0 \\ & f & & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.2}$$

We can now introduce the concept of camera matrix. The mapping between $3D$ and $2D$ is usually represented by a $3 * 4$ matrix called camera matrix or camera projection matrix. Where $\boldsymbol{x}$ represents the point in the image and $\boldsymbol{X}$ the point in the world.

$$x = PX \tag{3.3}$$

From this point on, we will refer to the camera matrix as P, $\boldsymbol{X}$ will be the homogeneous vector $(X, Y, Z, 1)^1$ that represents a point in the real $3D$ world and $\boldsymbol{x}$ the representation of the image point.

Equation 3.1 assumes that the center of the image plane is at the principal point. On the bad side we cannot assume this in practice and it must be taken into account when defining P. But, on the good side, this offset between the principal point and image plane is easily introduced in equation 3.1:

$$(X, Y, Z)^T \rightarrow (fX/Z + p_x, fY/Z + p_y)^T \tag{3.4}$$

and, using matrices:

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} fX + p_x \\ fY + p_y \\ Z \end{pmatrix} = \begin{bmatrix} f & & p_x & 0 \\ & f & p_y & 0 \\ & & 1 & 0 \end{bmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{3.5}$$

Considering

$$K = \begin{bmatrix} f & & p_x & 0 \\ & f & p_y & 0 \\ & & 1 & 0 \end{bmatrix} \tag{3.6}$$

we can say that K represents the camera's intrinsic parameters.
In theory, K, as described in equation 3.6, is sufficient to correctly describe the mapping. But, once again, in practice we have other factors to take into account. In real cameras there is the possibility of non-square pixels. We are not talking about cameras that purposely don't have square pixels, such as those that use hexagonal pixels. We are talking of errors that make the pixel's have a bigger width/height than height/width. If we measure coordinates in pixels we have different scales in the axes. Considering m_x and m_y as the number of pixels per unit of distance in image coordinates, $\alpha_x = fm_x$ and $\alpha_y = fm_y$ represent the focal length in terms of pixels in the x and y direction. We can then define K like:

$$K = \begin{bmatrix} \alpha_x & & x_0 & 0 \\ & \alpha_y & y_0 & 0 \\ & & 1 & 0 \end{bmatrix} \tag{3.7}$$

where $x_0 = m_x p_x$ and $y_0 = m_y p_y$. Even though this rarely happens, we must also introduce the possibility of non-perpendicular image axis. This is done by adding another variable (γ) to K that represents the skew of the image. We can finally define K as:

$$K = \begin{bmatrix} \alpha_x & \gamma & x_0 & 0 \\ & \alpha_y & y_0 & 0 \\ & & 1 & 0 \end{bmatrix} \tag{3.8}$$

We now have the intrinsic parameters of the camera but, P cannot be defined using only K, we also need the extrinsic parameters. Extrinsic parameters are the ones that define where the camera is in the world i.e. its rotation and translation. R is the rotation matrix and defines the camera's rotation in each angle $R = R_x(\Psi), R_y(\phi), R_z(\theta)$ where:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\psi & sin\psi \\ 0 & -sin\psi & cos\psi \end{bmatrix} R_y(\phi) = \begin{bmatrix} cos\phi & 0 & -sin\phi \\ 0 & 1 & 0 \\ sin\phi & 0 & cos\phi \end{bmatrix} R_z(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

$t = (x, y, z)$ is the translation vector and represents the offset between the origin of both coordinate systems. $T = origin_{object} - origin_{camera}$.

Now that we have both the intrinsic and extrinsic parameters we can define P.

$$P = K[R|t] \tag{3.10}$$

which means that

$$x = K[R|t]X \tag{3.11}$$

3.2.2 Methods

Now that we understand what a camera matrix is, we will talk about methods for estimation P therefore, calibrating the camera. There are several algorithms to do this and they do it by corresponding $3Dspace$ and image points. The idea is that, given enough correspondences $X \rightarrow x$, P may be determined. The usually used method to calibrate a camera, it's to obtain several images of a known structure in different positions and rotations. A common structure used in camera calibration is the chessboard plan. Basically we move and rotate the chessboard while the camera is taking images. Some other methods use $3D$ objects (usually covered in markers) with the same purpose without the need to move the object. These methods tend to be harder to work on. Considering the chessboard as the calibration object, from observing a $2D$ image of it, and knowing that all squares have the same size, we can determine it's rotation and relative position. This means that we can match x and X and, up to a certain level, know the third coordinate Z in X.

Since we know both x and X, from equation 3.3, $x = PX$ we can determine P. A rotation matrix can be defined by three angles 3.9 and the translation vector has another three variables giving us six unknown variables. As seen in 3.6, there are four other variables in the intrinsic parameters therefore, making a total of ten values that we have to find so we can define P. At least two images are needed to calibrate a camera, usually, the more you use the better results you get. There are several methods available to find this unknown values. We will analyze the method used in openCV based on Zhang [29, 30] and Borwn[4]. In order to facilitate the comprehension we will, once again, organize the algorithm in steps.

1. **Acquire images**

2. **Chessboard Corners**

3. **Homography**

4. **Constraints**

5. **Closed-form Solution**

6. **Getting the values**

7. **Distortion**

The first obvious step of the algorithm is to acquire images to work. These images must be of a chessboard pattern printed in a planar structure and we need at least two. Between frames, we can either move the pattern or the camera. This movement does not need to be known. From each image of the pattern we then extract the chessboard corners.

The following steps will be explained individually in the following sections.

Homography

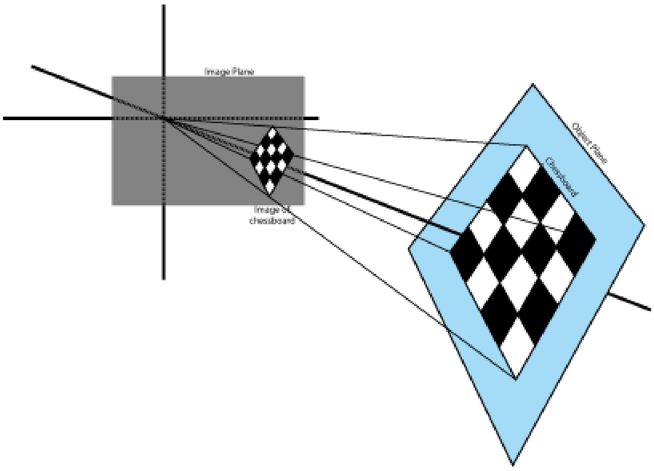


Figure 3.2: Planar Homography

In computer vision the term homography refers to mapping between points on two image planes. When we haev points in a plane surface and we map them to the image plane we have a homography, hence the need to use a planar surface for the pattern. Again using the notation of $\boldsymbol{X}$ and $\boldsymbol{x}$, we can express the action of homography as:

$$\boldsymbol{x} = sH\boldsymbol{X} \tag{3.12}$$

The importance of using H is that it not only includes the physical transformation but the projection, which includes the camera intrinsic matrix, as well. We can express this by splitting H.

Let us define

$$W = [R \;\; t] \;\; \text{and} \;\; M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

allowing us to express equation 3.12 as:

$$\boldsymbol{x} = sMW\boldsymbol{X} \tag{3.14}$$

To simplify H we will define $Z = 0$ therefore allowing us to ignore one column of $R = [r_1, r_2, r_3]$. We can now describe the homography by:

$$H = sM[r_1 \; r_2 \; t] \tag{3.15}$$

To find H, OpenCv uses multiple images of the same planar pattern. From the previous section we know that for each view there are six unknowns (three angles for rotation and three values for translation). Since we get eight equations from detecting the chessboard we have eight new equations and six unknowns for each view which will allow us to compute H (both intrinsic and extrinsic parameters).

Constraints

From the previous step we collect a homography H for each view. If we write H as column vectors, $H = [h_1, h_2, h_3]$ we can rewrite equation 3.15 as:

$$H = [h_1 \; h_2 \; h_3] = sM[r_1 \; r_2 \; t] \tag{3.16}$$

And, considering $\lambda = 1/s$:

$$\begin{aligned} h_1 = sMr_1 \quad &\text{or} \quad r_1 = \lambda M^{-1} h_1 \\ h_2 = sMr_2 \quad &\text{or} \quad r_2 = \lambda M^{-1} h_2 \\ h_3 = sMt \quad &\text{or} \quad t = \lambda M^{-1} h_3 \end{aligned} \tag{3.17}$$

Knowing that r_1 and r_2 are orthonormal, therefore $r_1^T r_2 = 0$, we get to our first constraint:

$$h_1^T M^{-T} M^{-1} h_2 = 0 \tag{3.18}$$

We also know that the rotation vectors have equal magnitude $||r_1|| = ||r_2||$ or $r_1^T r_1 = r_2^T r_2$, yielding our second constraint:

$$h_1^T M^{-T} M^{-1} h_1 = h_2^T M^{-T} M^{-1} h_2 \tag{3.19}$$

We now have two constraints on the intrinsic parameters.

Closed-form solution

In order to make things easier to read and understand we will define B as $B = M^{-T} M^{-1}$ meaning:

$$B = M^{-T}M^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{bmatrix} \tag{3.20}$$

Considering the closed-form solution of B we get:

$$B = \begin{bmatrix} \frac{1}{f_x^2} & -\frac{\gamma}{f_x^2 f_y} & \frac{c_y\gamma - c_x f_y}{f_x^2 f_y} \\ -\frac{\gamma}{f_x^2 f_y} & \frac{\gamma^2}{f_x^2 f_y^2} + \frac{1}{f_y^2} & -\frac{\gamma(c_y\gamma - c_x f_y)}{f_x^2 f_y^2} - \frac{c_y}{f_y^2} \\ \frac{c_y\gamma - c_x f_y}{f_x^2 f_y} & -\frac{\gamma(c_y\gamma - c_x f_y)}{f_x^2 f_y^2} - \frac{c_y}{f_y^2} & \frac{(c_y\gamma - c_x f_y)^2}{f_x^2 f_y^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix} \tag{3.21}$$

As mentioned before γ, the skew of the image is almost always equal to 0. Therefore, if we add this new constraint to equation 3.21 we can simplify it's writing and get:

$$B = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & -\frac{c_y}{f_y^2} \\ \frac{-c_x}{f_x^2} & -\frac{c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix} \tag{3.22}$$

From equations 3.20 we can see that B is symmetric, which means we can define it as a $6D$ vector $b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$. Going back to H, let us define the i^{th} column of H as $h_i = [h_{i1}, h_{i2}, h_{i3}]$ and with this we have:

$$h_i^T B h_j = v_{ij}^T b \tag{3.23}$$

Using this definition for v_{ij}^T, the two constraints previously defined (equation 3.18 and 3.19) can be rewritten as:

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^T \end{bmatrix} b = 0 \tag{3.24}$$

For n images we collect n of these equation which can be stacked in a $2n * 6$ matrix V getting $Vb = 0$.

Getting the values

Once we have V, from the previous step, we can solve the equation ($Vb = 0$) for b. Remembering now that $b = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$ the camera intrinsic parameters are directly calculated from the closed-form solution of B.

$$f_x = \sqrt{\lambda / B_{11}} \tag{3.25}$$

$$f_y = \sqrt{\lambda B_{11} / (B_{11}B_{22} - B_{12}^2)} \tag{3.26}$$

$$c_x = -B_{13}f_x^2/\lambda \tag{3.27}$$

$$c_y = (B_{12}B_{13} - B_{11}B_{23})/(B_{11}B_{22} - B_{12}^2) \tag{3.28}$$

$$\gamma = -B_{12}f_x^2 f_y/\lambda \tag{3.29}$$

$$\lambda = B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23}))/B_{11} \tag{3.30}$$

As for the extrinsic values, we compute them from the equation that we read off from the homography condition.

$$r_1 = \lambda M^{-1}h_1 r_2 = \lambda M^{-1}h_2 r_3 = r_1 \times r_2 t = \lambda M^{-1}h_3 \tag{3.31}$$

To get an exact rotation matrix, we can not simply join together the r-vectors to form $R = [r_1\ r_2\ r_3]$. If we do this, when using real data, the condition $R^t R = RR^T = 1$ will not hold. To solve this problem we use the SVD of R instead of the original R.

Distortion

We now have all the intrinsic and extrinsic parameters, but no work concerning distortion has yet been done. Cameras have distortion, in more correct words, camera's lens cause distortion in the image. Distortion makes us "perceive" points on the image in the wrong place. If we consider (x_p, y_p), the coordinates of a point, where 'p' stands for perfect as if the camera had no distortion, we can also consider (x_d, y_d) as the point's distorted location. With this we can state that:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = \begin{bmatrix} f_x X^W/Z^W + c_x \\ f_y X^W/Z^W + c_y \end{bmatrix} \tag{3.32}$$

Using the results of the calibration without distortion:

$$\begin{bmatrix} x_p \\ y_p \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2(r^2 + 2x_d^2) \\ p_1(r^2 + 2y_d^2) + 2 + p_2 x_d y_d \end{bmatrix} \tag{3.33}$$

To get the distortion parameters a large list of these equations are collected and solved. Once we get the distortion parameters the extrinsic parameters must be re-estimated.

3.2.3 Stereo Cameras

As mentioned before, one way of extracting $3D$ information from a scene is using stereo cameras. Along with the information about the camera's intrinsic and extrinsic parameters, we also need to know how to relate the two (more cameras can be used) cameras in the system. *Stereo Calibration* figure 3.3 is the process of finding a rotation and translation matrix that defines the geometrical relationship between the two cameras. In this section we will also talk about *Stereo Rectification* figure 3.4, which is the process of "making" the cameras optical axes appear parallel. With this, images appear as if they had been taken by cameras with aligned image planes. For these processes the same method of "showing" a calibration is used. Each camera takes, at the same time, an image of the chessboard.

Stereo Calibration

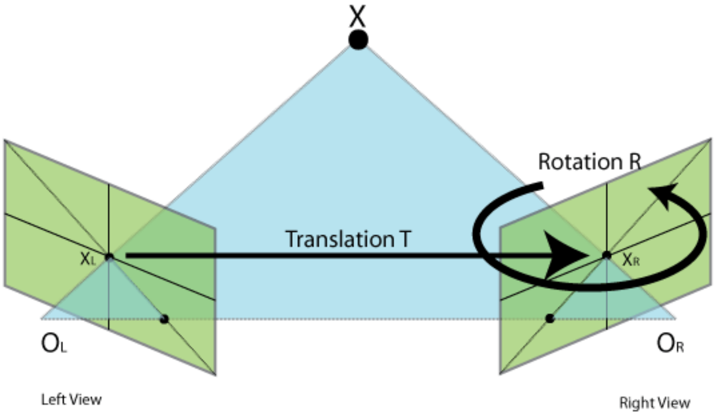


Figure 3.3: Stereo Calibration

So, the objective of stereo calibration is to geometrically relate the cameras but, how can we do this? We need two matrices, R, the rotation and T the translation. Let us denote T_lR_l and T_rR_r as the rotation and translation matrices, obtained using single-camera calibration, of the left an right camera respectively. Being P a point in space, in camera coordinates we get $P_l = R_lP + Tl$ and $Pr = R_rP + T_r$. Keeping in mind the meaning of R and T in stereo calibration:

$$P_l = R^T(P_r - T) \tag{3.34}$$

If we solve this for R and T:

$$R = R_r(R_l)^T T = T_r - RT_l \tag{3.35}$$

For stereo calibration we use the method described in section 3.2 to get the translation and rotation values of the chessboard and use them in the previous equation. This is done for every view and at each one: errors are introduced due to noise and rounding. To get minimum error values, a Levenberg-Marquardt [19, 25, 21] (LMA) algorithm is used. Starting with a median of the values of R and F as an initial approximation, the algorithm then finds the minimum of the reprojection error, returning the solution for R and T.

Bouguet's Algorithm for Stereo Rectification

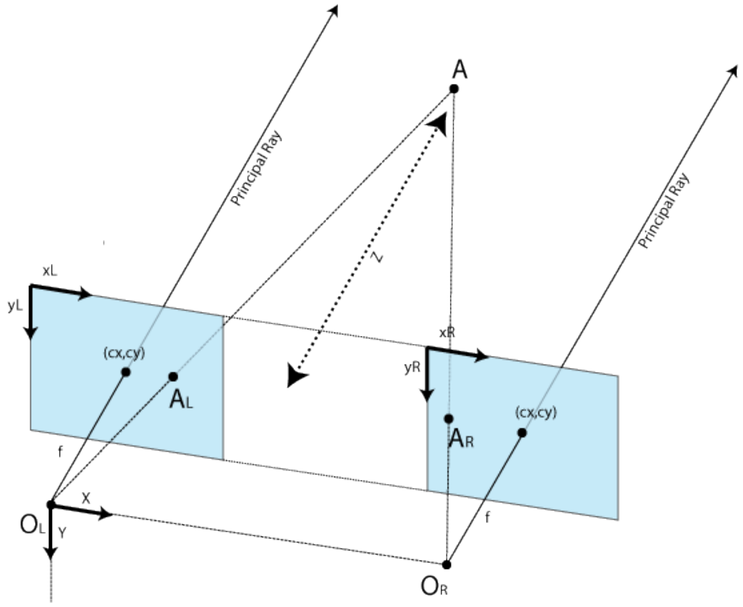


Figure 3.4: Stereo Rectification

As we will see in the development part of the report, it is easier to work with rectified images. In image 3.4 we can see the objective of image rectification. In OpenCv, two algorithms are used for image rectification. Hartley's [12] and Bouguet's algorithm [1]. For Hartley's algorithm we simply need to observe points in an image but there is no sense of image scale, furthermore it tends to produce more distorted images. For these reasons and because we can use calibration patterns, we will choose Bouguet's algorithm.

Bouguet's algorithm works by attempting to minimize of image retro-projection distortion while, at the same time, maximizes common viewing area between the two images. Part of Stereo Rectification is aligning the optical axes. Knowing that we have R that relates both cameras, if in each camera we apply half a rotation ($r_r r_l$ for the right and left camera respectively) we achieve coplanar alignment. To achieve row alignment we need to define R_{rect}, we start by defining three vectors:

$$e_1 = \frac{T}{||T||} \qquad e_2 = \frac{[-T_y T_x 0]^T}{\sqrt{T_x^2 + T_y^2}} \qquad e_3 = e_1 \times e_2 \tag{3.36}$$

[1] Jean-Yves Bouguet never published this algorithm. The algorithm is based in the method presented by Tsai [27] and Zhang[29] [30]

Where e_2 is orthogonal to e_1 and e_3 is orthogonal to both e_1 and e_2. R_{rect} can now be defined:

$$R_{rect} = \begin{bmatrix} (e_1)^T \\ (e_2)^T \\ (e_3)^T \end{bmatrix} \tag{3.37}$$

R_{rect} will make the *epipolar lines* horizontal and locates the *epipoles* at infinity. The concept of epipolar lines and epipoles will be explained in section 3.3.1. To achieve row alignment, for each camera we have:

$$R_l = R_{rect} r_l R_r = R_{rect} r_r \tag{3.38}$$

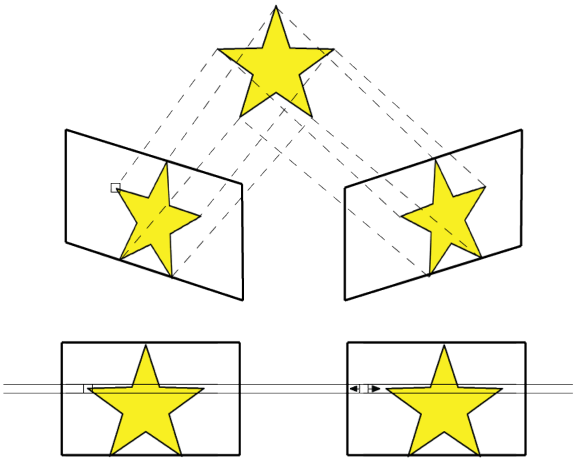


Figure 3.5: Rectified Image

Stereo rectification allows us to find the rectification maps which, when applied to images, aligns them. When we say that the images are aligned we mean that the rows of pixels between the two cameras are aligned as we can see in figure 3.5. It is easily understood the advantages that this brings. Having to find the same point in two images we can restrict the search to a row of pixels instead of searching in the entire image. With this we can increase efficiency and, by restricting the points that are possible matches, we also increase reliability.

3.3 Reconstruction

3.3.1 Epipolar Geometry

When a scene is viewed from two different cameras every point of the scene will have different projections in each image plane. In fig3.6 we have two cameras with their respective focal points O_L and O_R, observing a point X. The projection of X onto each of the image

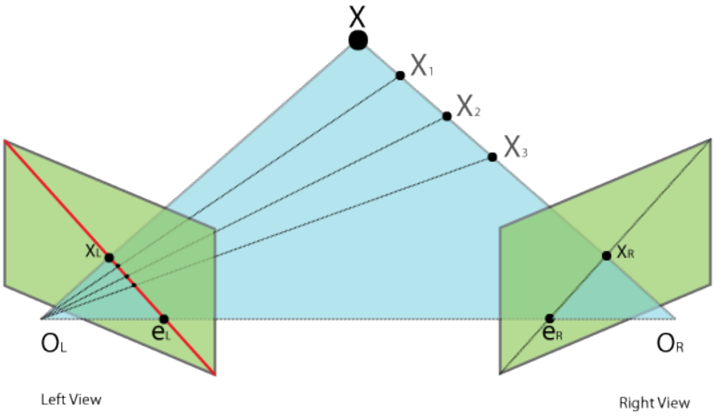


Figure 3.6: Epipolar geometry

planes is denoted x_L and x_R. The line connecting both focal points is called baseline and its intersection with the images planes creates e_L and e_R which are the epipoles. The plane defined by the baseline and point X is called the epipolar plane. The line that contains a projection and an epipolar point on the same image is called an epipolar line. The epipolar geometry of a scene can be defined by a matrix called Fundamental Matrix (F). This matrix depends only on the relative position of the cameras.

$$x'^T F x = 0 \tag{3.39}$$

Eq3.39 defines the relation between the two cameras and the projected points. If the equation is confirmed, the point x_L belongs to the left view epipolar line that corresponds to projection point x_R. Note that a null result is not enough to say that x_L and x_R are projections from the same point in the scene. As we can see from Fig3.6 three example points ($X1$,$X2$ and $X3$) will all have a null result in Eq3.39 even though they are different points in the scene.

3.3.2 Fundamental matrix

The fundamental matrix [24] [11] is a 3 × 3 matrix with rank 2 and 7 degrees of freedom. Even though there are 9 elements, scaling is not significant and $DetF = 0$ removing one degree of freedom. There are two different ways of finding the Fundamental Matrix, we can either find it from known camera matrices or compute it from point correspondences. In single camera systems we will be using only one camera which can be calibrated. However, even if we calibrate the camera we can only use the intrinsic parameters. A camera matrix is composed of intrinsic and extrinsic parameters, using a moving camera makes the use of the same extrinsic parameters throughout the process impossible. For this reason we will rely on extracting the camera matrices from the fundamental matrix, which in turn will be extracted from point correspondences. To extract the fundamental matrix from camera matrices we use the following:

$$F = [e']_x P' P^+ \tag{3.40}$$

Equation Eq3.40 will give the fundamental matrix from two known camera matrices where e' is the epipole of camera P', $[e']_x$ is the *skew-simetric* correspondence of e' and P^+ is the *pseudoinverse* of P.

If camera matrices are not known we can compute F from point correspondences between the images. If there is only one camera this is the most effective way to extract 3D information from a scene. We can also predict where the camera is after a certain movement and define it as the second camera. Predicting where the camera will be after a certain movement is not accurate for various reasons such as unpredicted variations on the floor. The procedure of estimating the intrinsic parameters of two unknown cameras from point correspondence is known as *structure from motion* [11, ?]. Starting with eq 3.39 for any pair of corresponding points x_i and X_i' in the two images we get:

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0 \tag{3.41}$$

For n pairs of points we have:

$$Af = \begin{bmatrix} x_1'x_1 & x_1'y_1 & x_1' & y_1'x_1 & y_1' & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n'x_n & x_n'y_n & x_n' & y_1'x_n & y_n' & x_n & y_n & 1 \end{bmatrix} f = 0 \tag{3.42}$$

Because F has only 7 degrees of freedom only 7 pairs of points are needed to find a solution however 8 pairs are usually used. Solving the equation system $Af = 0$ we can get F. Most of the times more than 8 points are used and each extra point originates an extra line in matrix A. If we have more points than variables in an equation system, the system is said to be over-determined. In most of the cases there is no single solution and for that we must find the one which minimizes the error. The most commonly used methods of solving such equation systems are the RANSAC algorithm [7] and the SVD method [28].

The Ransac (RANdom SAmple Consensus) algorithm is a method to estimate parameters of a mathematical model designed to cope with a large proportion of outliers in the input data, which normally happens when extracting 3D information from a scene. RANSAC is an iterative method choosing in each iteration a random minimum number of points required to solve the model. After solving the model with the chosen points it will determine how many of the remaining points are consistent with the model. If the number of consistent points is enough (determined with a given tolerance) the model will be re-estimated using all consistent points and terminates. If the number isn't enough a new sample of points will be randomly chosen and following steps repeated.

The SVD (Singular Value Decomposition) method is a commonly used method for solving matrix equations of the type $Ax = 0$. The method starts by separating the original matrix A in three matrices $A = UDV^T$. D is a diagonal matrix containing the singular values of A. U and V are orthogonal matrices where V contains the singular vectors of the correspondent to the singular values of D. When the method is applied to A, the last column of resulting V will correspond to a minimizing solution of f. With f the matrix F can then be built.

3.3.3 Reconstruction

Triangulation, sometimes also referred as reconstruction, is the method of finding a point in a 3D space given at least two projections of the same point in images. The work-flow of 3D reconstruction is the following:

1. Find points

2. Match points

3. Fundamental matrix F

4. From F compute P_A and P_B

5. Compute X in the world for each x and x' matching pair.

Up to this point, steps 1, 2 and 3 were already explained and, in the previous section we, explained that the camera matrices can be either extracted from the fundamental matrix or built from known camera parameters. Now we will show how to extract camera matrices from the F as well as how to triangulate a point.

Camera matrices

The fundamental matrix encodes information of both the intrinsic and extrinsic parameters of the cameras. Let us start by defining P_A and P_B as the camera's projection matrices and remembering the basic form of a camera matrix:

$$P = K[R|t] \tag{3.43}$$

Now let us consider that P_A is in the origin of the world coordinates, meaning that $t = [0, 0, 0]^T$, and that P_A's focal length = 1. With this we can write 3.43 for P_A as:

$$P = [I|0] \tag{3.44}$$

Then, because:

$$x'^T F x = X^T P_P F P_A X \tag{3.45}$$

and:

$$P_B^T F P_A = [SF|e']^T F[I|0] \tag{3.46}$$

Where e' is the epipole and S a skew-symmetric matrix, we can say that:

$$P_B = [SF|e'] \tag{3.47}$$

A good choice for S is $S = [e']$, which means that we can once again rewrite P_B as

$$P_B = [[e']F|e'] \tag{3.48}$$

All we need now is to find the epipoles from F. Two of the properties of F is that $Fe_1 = 0$ and $e_2^T F = 0$. Using SVD we can easily calculate the epipoles.

Triangulation

In image fig3.6 we can see that point X is the intersection of the lines that start in O_L and O_R and pass through X_L and X_R respectively. These lines are called projection lines. This intersection along with both focal points creates a triangle which is the base for triangulation. In theory these lines should always intersect however for various reasons such as camera distortion this intersection can never happen. If the lines did intersect, finding an intersection in space between two lines is a trivial procedure.However an intersection, in most cases, won't be found. Different triangulation methods deal with this problem differently [13].

The common method to deal with the problem of non-intersecting lines is the Mid-Point method [3], which as the name suggests will choose the middle point of the smallest line connecting both projection lines. This method will not give the best results since a lot of approximations are made. Other methods [13, 15] are based in moving the corresponding pixels of the point's projection point so that the projection lines intersect. Such methods are called *optimal correction methods*. The main goal of these methods is to minimize 3.39. For an optimal solution the movement of the pixels must be the minimum needed i.e. minimizing the error given by the squared sum of the movements. While in [13] the error is minimized by solving a grade 6 polynomial, [16] chooses a different approach that can be found in [15, 17]. [16] has proven that [15] method is more efficient for its faster results and numeric results similar to [13]. With these methods an intersection is guaranteed and a simple triangulation can be made.

Must of the work is now done and the only step remaining is to find the point's world coordinates. Let us go back to section 3.2.1 where the meaning of a camera matrix was explained:

$$x = PX \tag{3.49}$$

Considering that there are two cameras "looking" at the same object we have:

$$x = P_A X, x' = P_B X \tag{3.50}$$

The strategy to find X is combining 3.50 into a form $AX = 0$.

$$x = PX \leftrightarrow x(PX) = 0 \tag{3.51}$$

$$PX = \begin{bmatrix} P_1^T \\ P_2^T \\ P_3^T \end{bmatrix} X = \begin{bmatrix} P_1^T X \\ P_2^T X \\ P_3^T X \end{bmatrix} \tag{3.52}$$

$$AX = \begin{bmatrix} xP_{A3}^T & -P_{A1}^T \\ yP_{A3}^T & -P_{A2}^T \\ x'P_{A3}^T & -P_{B1}^T \\ y'P_{A3}^T & -P_{B2}^T \end{bmatrix} X = 0 \tag{3.53}$$

Solving A with SVD, the last column of V will be X. Note that the solution for X is not homogeneous, $X = (x, y, z, w)$. In order to get the $3D$ coordinates of the point, one has to convert them to homogeneous. This is done by dividing all the three coordinates by w

$$\begin{aligned} x' &= x/w \\ (x, y, z, w)n \longrightarrow (x', y', z') where y' &= y/w \\ z' &= z/w \end{aligned} \tag{3.54}$$

For each of the points used to compute F we solve equation 3.53 and then convert the point into homogeneous.

Chapter 4

Development process

4.1 Introduction

Now that we have all the concepts explained, we reached the development stage. We can divide the work into two main parts: point extraction and 3D reconstruction. For the point extraction we will describe the implementation and results of each of the algorithms used (SIFT SURF and SUSAN as described in section 2). We will start by explaining the common work to the three algorithms and then analyze each in detail, first the finding of the points and then the matching.

For the $3D$ reconstruction part we will first analyze the implementation and results of a single camera approach followed by the stereo camera approach.

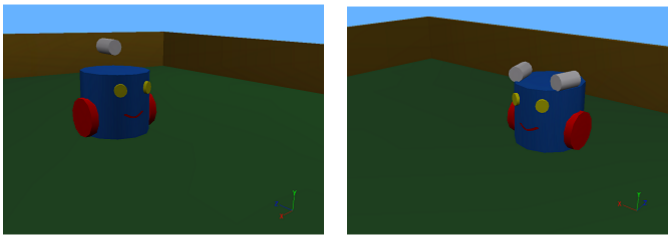
In all of the work we used the programming library OpenCV (Open Source Computer Vision). OpenCV provides several, already implemented, concepts of 3D reconstruction as well as basic computer vision functions. All shown tests were performed in a "Compaq 6720s" laptop with "Intel Pentium Dual CPU T2390 1,86GHz" with 4Gb memory running "Windows 7 32bits".

In early stages of the development we used Webots as a simulation platform. With Webots we have a controlled world which gives some advantages such as:

- **Knowledge**
  We now the exact position of every point in the world.

- **Perfect Cameras**
  Cameras do not have distortion or noise. In stereo systems we also know the geometric relation between both cameras.

- **Control**
  If only one camera is used we can have a repetitive controlled motion in one camera and take frames in pre-defined locations.



(a) Single camera (b) Stereo camera

Figure 4.1: Simulated robots in Webots.The grey blocks represent the cameras

The results in WebBots will be presented later.

4.2 Common work

Before we can implement the concepts described in section 2, there is some common work to implement.
The first step is to calibrate the cameras. For one or two cameras we start by taking images of a known-sized chessboard and find the chessboard corners in the image with *cvFindChessboardCorners*. We do this a predetermined number of times (must be more than 2 we used 15) . To increase accuracy and refine corners location one can use *cvFindCornerSubPix*. By defining the size of the board in terms of $board_{size} = board_{width} \times board_{height}$ we know how many corners we must have in each view, therefore we consider the view a success if $corners_{found} = board_{size}$. All points belonging to a successful view are stored for later use. After we have enough successful views we can finally use *cvCalibrateCamera2*, which returns the camera's parameters. If we are using a stereo system there is more work to be done. First at each view we must store two images (one for each camera, taken at the same time) and two sets of points. Instead of using *cvCalibrateCamera2* we use *cvStereoCalibrate* which takes the information from both cameras and returns each camera's intrinsic matrix along with their geometrical relation. Using *cvStereoRectify* we get each camera's projection matrix along we other matrices that are used in *cvInitUndistortRectifyMap* in order to get the rectification maps. Now we have the P's (camera matrix, section 3.2) and some other useful data such as the rectification maps. The process of camera or stereo calibration is done only once during the process and data obtained can be reused as long as the cameras do not change.

For 3D reconstruction we need two images, those images can be from the same camera, after it moves, or from a stereo system. Either way two images will be used and for that reason we start by creating two *IplImages* with a size of 320 × 240. We choose this size merely for reasons of speed. Since the first objective is to extract points from an image we need some structure to store them. Every algorithm uses their own structures inside, but,

to facilitate switching between algorithms, they all return a list of *CvPoint* for each image. At the matching stage this list is created by the algorithms in a way that the matches are in order, meaning that $CvPoint_{left}[i]$ is the match for $CvPoint_{right}[i]$. We then converted the lists to *CvMat*. Each image, at each frame, as a matrix with size $correctmacthes \times 2$.
Now that we have the cameras calibrated and the data structures we need to define, we can implement all algorithms and concepts previously explained.

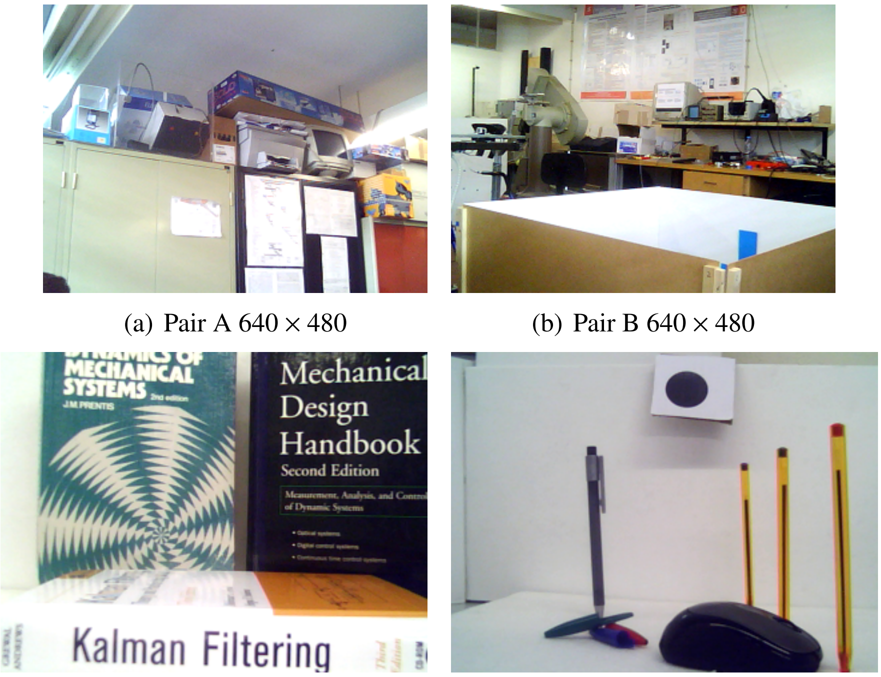
4.3 Point extraction

In this section we will show the development process to extract points from images. Three algorithms were explained and we will now show how they are used. Each algorithm has its own structure that defines a feature point with different components but they follow a general rule. A point has:

- Location
- Descriptor

Both components have clear meanings and they will be used to match, compare and locate points.
Every algorithm starts by extracting points and when both images are analyzed they search for the points from $image_1$ in the points from $image_2$. Both stages in feature extraction are important in terms of speed and reliability.

For tests we used a set of pictures 4.3 and ran them in each algorithm. For each image in 4.3 there is another image taken at the same time from a different position. These images were taken by a stereo camera. The speed is easily calculated and we present the results in each algorithm in form of a table. This table presents the number of points found in each image as well as the number of successful matches. It also presents information about time such as, how long it took to find the points, to match them, and how long in total the algorithm took.In order to eliminate the possibility of a background process influencing the time results, tests were repeated 20 times and we present the average of the results obtained. Testing for stability is not as linear as speed: if an algorithm is stable the same points should be found in the same image in every test. Running the algorithm in real-time (i.e at each frame we find and compare points) if we see a lot of inconstant points, it means that the algorithm is not stable. Robustness can be measure by the amount of false positives the algorithm returns. For each algorithm we will present the time of execution, number of features found and a comment about it's robustness and stability.



(a) Pair A 640 × 480

(b) Pair B 640 × 480

(c) Pair C 320 × 240

(d) Pair D 320 × 240

Figure 4.2: Test images

4.3.1 SIFT Algorithm

Methods

In SIFT a feature is represented by a structure that has several components. From that structure those components with greatest importance are:

- Location. X coordinate.
- Location. Y coordinate.
- Descriptor. A lists of *double* that describes the point.

The algorithm uses a list of these structures to store and manage the points. To find points in an image we use the following algorithm:

```
For image_left and image_right:
1 - Build Gauss Pyramid
    1.1 - For each wanted level Smooth previous Level
2 - Build Dog Pyramid from previous pyramid
    2.1 - For each level in the pyramid:
          Create level in DoG pyramid using cvSub
3 - Scale Space Extrema
    3.1 - For all levels in the DoG pyramid:
          Search for local maxima or minima
4 - Compute descriptors
5 - Sort Features
```

The first step is to build a Gaussian pyramid for each image. A pyramid is basically a list of *IplImage*, for the Gaussian we start with the original image, convert it to 8-bit grayscale and Gaussian-smooth it. To smooth an image we use *cvSmooth* with $smoothtype = CV\_GAUSSIAN$. To get a new level in a pyramid we re-smooth the previous level. A difference of Gaussians pyramid is built by subtracting adjacent intervals of the Gaussian pyramid. The DoG pyramid is created using $cvSub\ DoG_i = cvSub(GaussP_{i+1}, GaussP_i)$. With the DoG pyramid built we search the pyramid for points that are either local maximum or minimum discarding those with low contrast. Up to this moment, for each point, we have a location and the scale in which it was found. We now compute a histogram of gradient orientation around the point and convert it to an array. We now have a descriptor.
Finding the points is the first part, now we need to match them in both images.

```
2 - Build a kd-Tree with features from image_right
3 - For each feature in image_left
    3.1 - Find its nearest neighbors in the built tree using Best Fin F
    3.2 - Square distance between point's and result's descriptor.
    3.3 - If distance is less than a threshold
          Points are a match
```

As we can see, a KD-Tree is built only with the points of the right image. Having an organized structure of points, from $image_{right}$, for each point from $image_{left}$ we are going to search the tree for its match.

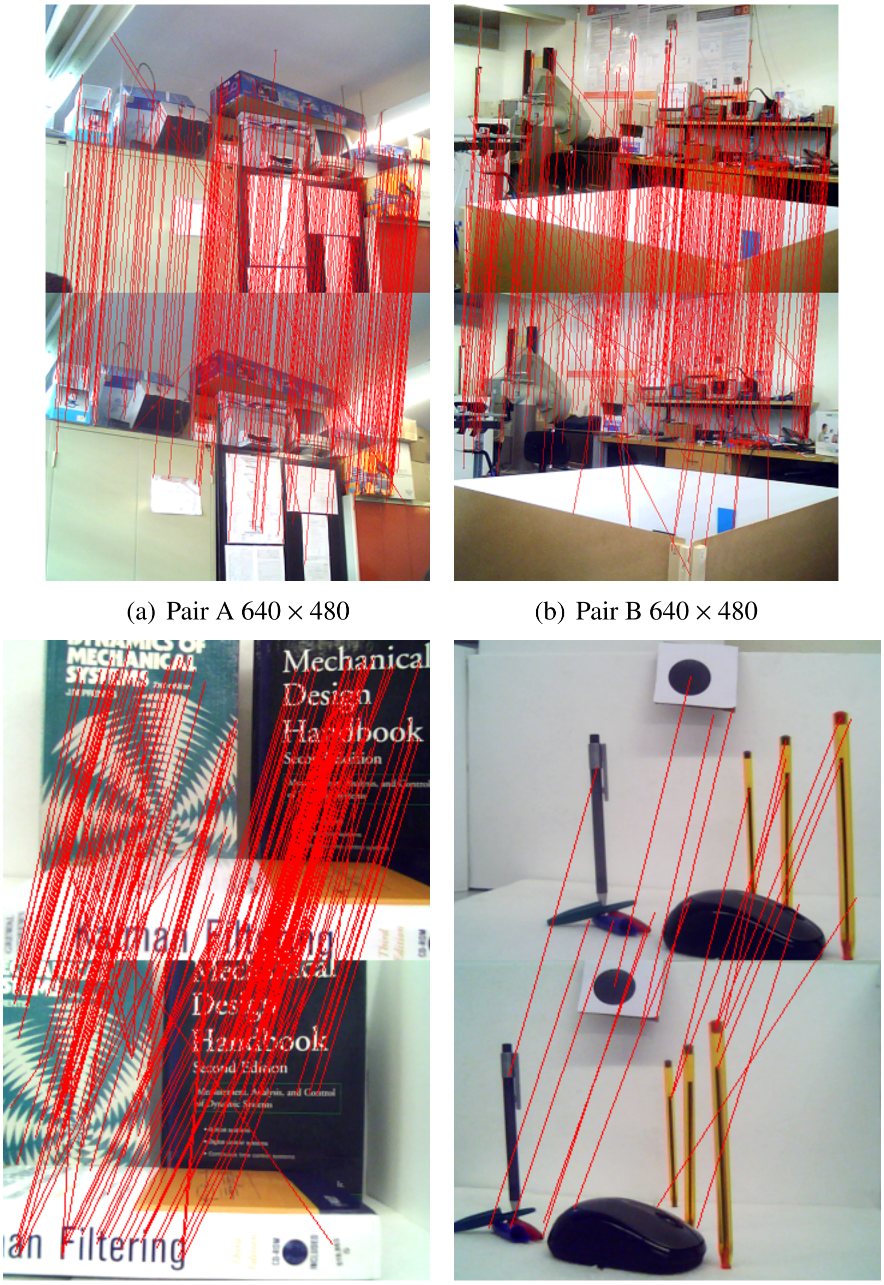
Results

After running the algorithm in the image set we have come to the results found in image 4.3.1 and table 4.3.1.

	Left Points	Time(ms)	Right Points	Time(ms)	Matches	Time(ms)	Total Time(ms)
Pair A	895	1016.66	779	944.116	327	128.831	2091.41
Pair B	1154	1181.35	988	1058.89	321	170.939	2413
Pair C	721	632.845	630	550.311	246	130.458	1320
Pair D	83	180.269	75	187.602	25	5.12	376.819

Table 4.1: SIFT results

From table 4.3.1 we can easily obtain information about the speed of the algorithm. As expected, in higher resolution images, the algorithm takes longer. A big difference in time between Pair C and D, which have the same size, is visible. This is due to the fact that C has



(a) Pair A 640 × 480

(b) Pair B 640 × 480

(c) Pair C 320 × 240

(d) Pair D 320 × 240

Figure 4.3: SIFT Image Results

a more complex texture. Complex textures tend to have more features, therefore more points are found and in consequence more descriptors need to be calculated. Looking at the time it took to find the points in the left and right image, we can see that the more points you find the longer it takes. This does happen in pair D where it took longer to find lesser points, such a small difference in time is considered as not significant and it is probably due to an outlier in the average results. In the image results in 4.3.1, lines connect matches and as we can see they all tend to follow one direction disregarding inclination. This is a good sign. If we

consider the objective of the work we either have a moving camera or a stereo system with this, we can say that both systems' images are related by a movement in the camera. If the *match lines* tend to have the same direction it means that all pixels were "moved" the same way between the images. Different directions in the lines means that some pixels "moved" to the right, while others "moved" to the left. Being this impossible for us[1], different direction lines show us matching errors.
Running the algorithm in real time we can see that most line keep a constant direction but there are some blinking points. The algorithm appears to have a good stability.

4.3.2 SURF Algorithm

Methods

OpenCV already has an implementation of SURF. Nevertheless we will explain how this was done. As for SIFT, there is a structure for SURF points. In OpenCV we have: *CvSURFPoint*

- CvPoint2D32f representing it's location
- The sign of the Laplacian (section 2.3.2).
- Size of the feature
- Orientation of the feature
- Value of the Hessian

When looking at the components of a *CvSURFPoint*, one might ask where is the descriptor for this point? In OpenCV the descriptor is not a part of the point's structure, when finding features we get a list of points and a list of descriptors.

The first step is to define *CvSURFParams* which establishes the Hessian threshold and the size of the descriptors (64 or 128). A common choice for a threshold is 500 and we use 128 descriptors, since bigger descriptors mean better matching. Even though the matching stage is slower when using 128 sized vectors, as we will see later on SURF is still much faster than SIFT.
To extract features from an image we call:

```
void cvExtractSURF (const CvArr∗ image, const CvArr∗ mask,
                    CvSeq∗∗ keypoints , CvSeq∗∗ descriptors ,
```

[1] Different line directions are not impossible to happen. If we consider high rotation in an image (ex: turn the camera upside down) different line directions are expected, but, since we do not expect that such cases should occur with our system we consider different direction an error in the matching. Furthermore the direction of the lines is not tested when using a single camera.

```
                    CvMemStorage* storage, CvSURFParams params)
```

In *cvExtractSURF* we have an image, an optional mask (can be used to define an area where to search for features), the keypoints represented by a sequence of *CvSURFPoint*,the descriptors where each descriptor is a list of 128 *floats*, the memory storage and the parameters. Therefore the algorithm has the following workflow:

```
For image_left and image_right:
Call CvExtractSURF
    1 - CvIntegral
    Hessian Detector
    2 - For every octave in image scale space
        2.1 - For every layer in the pyramid
            2.1.1 - Calculate and store Hessian Matrix Determinant.
            2.1.2 - Calculate and store trace of the Hessian matrix.
        2.2 - Find Interest points. For every point in every layer:
            2.2.1 - Hessian threshold
            2.2.2 - Non-maxima suppression
            2.2.3 - Calculate wavelet center coordinates
            2.2.4 - Interpolate maxima location
```

The first step, as described in section 2.3, is to create an integral image where we can work. This is done calling *CvIntegral*, and for the steps described above we only use the integral image. The determinants and traces are stored using lists of *CvMat* where we have an entry in the list for every layer. Each matrix on the list represents the determinants of the pixels on that layer. When the determinants are calculated, those that have a value less than the threshold are ignored. Then tests are made to see if the point is a local maxima, this is done by comparing the point to it's 26 neighbors. After calculating the center of the wavelet and interpolate it we have the interests points.

Now we have to assign descriptors to the points. The generalization of the algorithm for this is:

```
For all points previously found:
    3 - Orientation Assignment
        3.1 - For all points inside wavelet calculate direction
        3.2 - For all possible directions
            3.2.1 - All previous directions grouped
        3.3 - Choose direction with highest response
```

Assigning a direction to the point is the first step. For that we start by defining a round area around the point. For all pixels inside the area we calculate their directions in X and Y. Assuming that there are 360 possible directions, we have to test all of them to choose which one will be assigned to the point. In this step we can choose the size angle we are testing. Starting $angle = 0$ at each step we can increase the angle $WindowSize$ degrees, $angle+ = WindowSize$, let us call this angle of search, search area. For all search areas we sum all directions (found in step 3.1) that fall inside it and, if the result is higher than the previous search area, we choose it as the assigning direction. In the end we will have the direction with the highest response vector.

```
4 - Descriptor
    4.1 - Define a square area of pixels around the point
    4.2 - For all pixels calculate gradients in x and y with wavele
    4.3 - Construct the descriptor. For each sub-region:
        4.3.1 - Store the 8 values
```

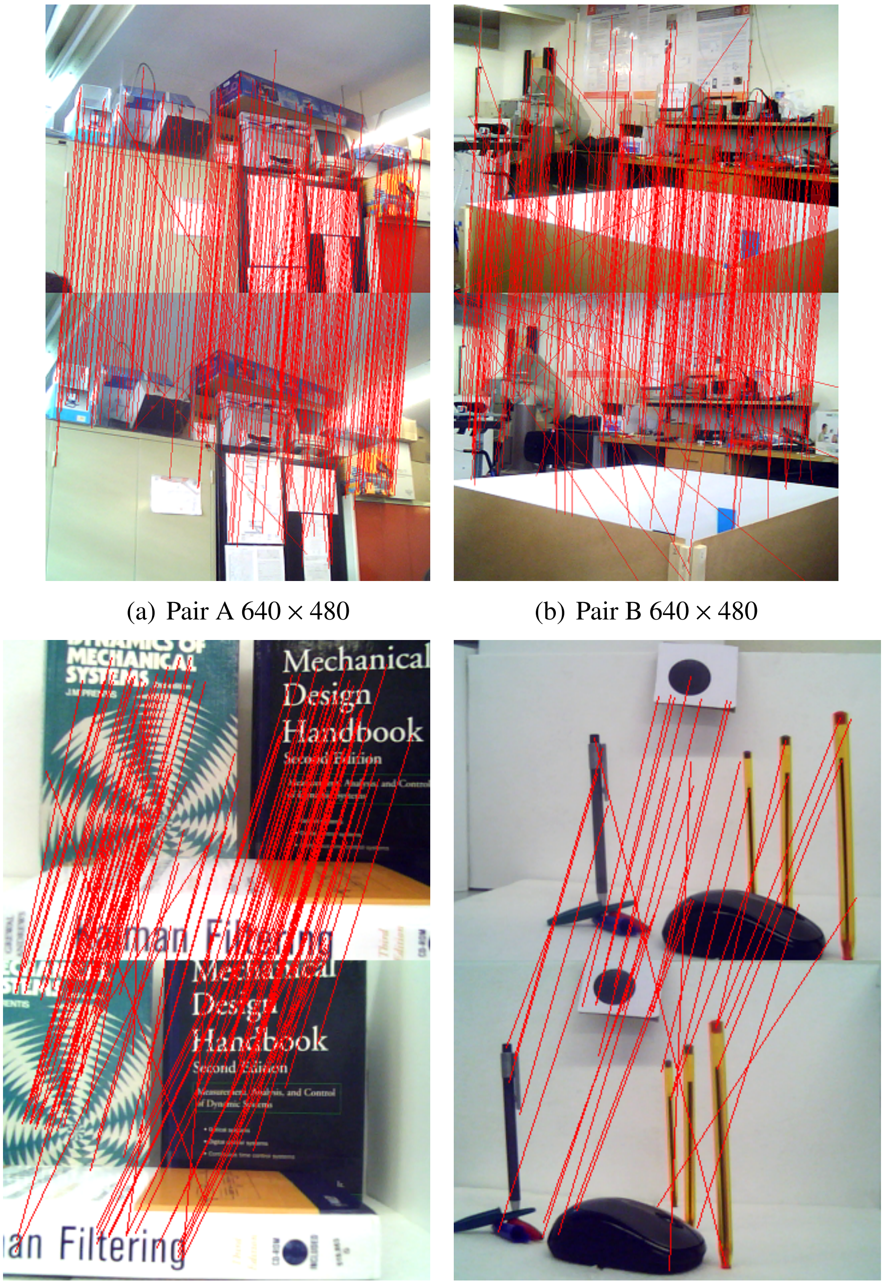
With step 3 we have defined an orientation for the pixel and, in step 4, we create the descriptor. In step 4.2 we use two lists of $float$, one for each direction, to store the wavelet responses, then in step 4.3 we use these values to define each sub-region's values. Basically for each sub-region we have a vector with size 8 and each element of the vector is a value from the stored wavelet responses. If we are using 64 descriptors instead of 128 we use a vector with size 4.

We now have a list of *CvSURFPoint* but there is still work to be done. We still need to match these points. OpenCv does not match points automatically and therefore we need to implement a way of doing it. For that we used FLANN (Fast Library for Approximate Nearest Neighbors) which is an already implemented library with algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. In the end we get a vector of indexes where i is a match for $i + 1$.

Results

After running the algorithm in the image set we have come to the results found in image 4.3.2 and table 4.3.2.

When comparing these results with the ones obtained with SIFT, one immediate conclusion is that SURF is faster than SIFT. With SURF we have less than half computation time in all tested images. Analyzing the algorithm in terms of robustness and stability, the results showed that SURF has an equivalent performance to SIFT. By considering only these two



(a) Pair A 640 × 480

(b) Pair B 640 × 480

(c) Pair C 320 × 240

(d) Pair D 320 × 240

Figure 4.4: SURF Image Results

	Left Points	Time(ms)	Right Points	Time(ms)	Matches	Time(ms)	Total Time(ms)
Pair A	687	400.15	657	399.924	320	59.6798	864.348
Pair B	845	466.553	795	432.178	355	72.98	976.343
Pair C	423	138.305	312	121.467	123	35.653	299.547
Pair D	74	61.433	94	67.80	31	7.74	141.592

Table 4.2: SURF results

algorithms we can say that SURF is the obvious choice.

4.3.3 SUSAN Algorithm

Methods

For SUSAN corners there is no default implementation in OpenCV therefore structures had to be implemented as well as the main algorithms. The first step is to be able to represent a point, for that, the structure *CORNER* was created and it is formed by:

- x location
- y location
- Descriptor Vector
- info
- Brightness
- First order derivatives

The first three components have obvious meanings. Info component in its turn is used as an auxiliary variable used to, for example, mark the end of a list or mark the point for deletion. The following components are, the brightness of the pixel, and first order derivatives. These components are used along with the descriptor in the matching stage. For SUSAN corners a basic list of *CORNER* will be used to represent the image features. Having a structure to represent a point we can implement the algorithm.

```
For each image:
1 - Setup brightness
```

The first step for extracting SUSAN corners is to implement the brightness of the image as a LUT (Look-Up-Table),the LUT and is defined $uchar * bp$. This step allocates enough memory for $bp*$ and repositions the pointer to the center of the allocated space. The reason why a LUT is used is speed of computation. Using memory pointers alongside with bp we can facilitate some computation. Remembering the USAN area, as described in section 2.4.1:

$$c(\overrightarrow{r},\overrightarrow{r_0}) = e^{-(\frac{I(c(\overrightarrow{r})-I(c(\overrightarrow{r_0}))}{t})^6} \tag{4.1}$$

$$n(\overrightarrow{r_0}) = \sum_{\overrightarrow{r}} c(\overrightarrow{r},\overrightarrow{r_0}) \tag{4.2}$$

Where n is the area of the USAN.From equation 4.1 we can see that a lot exponential calculations have to be done in SUSAN. For all possible image brightness differences we calculate e^{-x^6}, convert the result to a *uchar* in the range of 0-100 and store it in the LUT. Therefore $bp[26]$ is the result for 26 levels of brightness difference. But how can we use this LUT in the algorithm? Let us consider:

$$P = Img * +(y - 3) * img_{width+x-1} \tag{4.3}$$

$$CP = bp + Img[y * img_{width} + x] \tag{4.4}$$

Where P is an image pointer and CP is a LUT pointer. In this case P points to the first pixel in the circular mask surrounding point (x, y) and CP points to a position in the LUT corresponding to the brightness of the center pixel(x, y). If we want to find n, the area of the USAN, we just have to, for every pixel inside the mask:

$$n+ = *(CP - *P + +) \tag{4.5}$$

By moving CP an amount equal to the value of the pixel pointed by P, we subtract the two brightness values and perform the exponential function.
Up to this point no work has yet been done in the image, we only prepared the LUT as a helpful tool in following steps. Continuing on the algorithm:

```
2 - Smooth image using 3x3 window
3 - For all pixels Compute USAN area
    3.1 - If USAN area to big ignore point
4 - For all pixels find local maximas
    Use 7x7 search mask
5 - For all local maximas
    5.1 - Store point
    5.2 - Compute brightness
    5.3 - First order derivatives
6 - For all corners compute descriptor
```

The process of finding the corners starts by calculating the USAN area for all pixels in the image. This is done using the concept represented by equation 4.5. If the USAN area passes a certain size, the point is no longer considered a corner and therefore is ignored. Using a 7×7 window we search all image for local maxima, this is done by comparing the point to all it's neighbors and, if it is bigger, it is considered a local maximum. Different sized windows can also be used as long as they have odd size (ex: 5 × 5). If it is consider a local maxima it is also consider a corner and therefore an image feature. The final step of finding a point is assigning it a descriptor. For this, using the concept of Kirsch described in section 2.4.2, for each point we assign a *int[8]* vector as a descriptor. Each member of the vector is a direction of the Kirsch detector.

Now that we have the points, there is still the step of matching the points between the images. Both in SFIT and SURF a single approach was used to match the points; however in SUSAN two different approaches were used. The first approach was the straight-forward method of comparing points one-by-one. Since result were not as good as expected, a different method was then used. Since the second method is to be used only with stereo systems it will be explained later on section 4.4.2.

As we already said the first implemented method has a straight-forward workflow:

```
7 - For all points in image 1
    7.1 - Find the best match in points from image 2
```

Considering that we have a vector and three variables that describe a point, for every point in the first image we search all points in the second image for the one that has the lowest squared distance between the values.

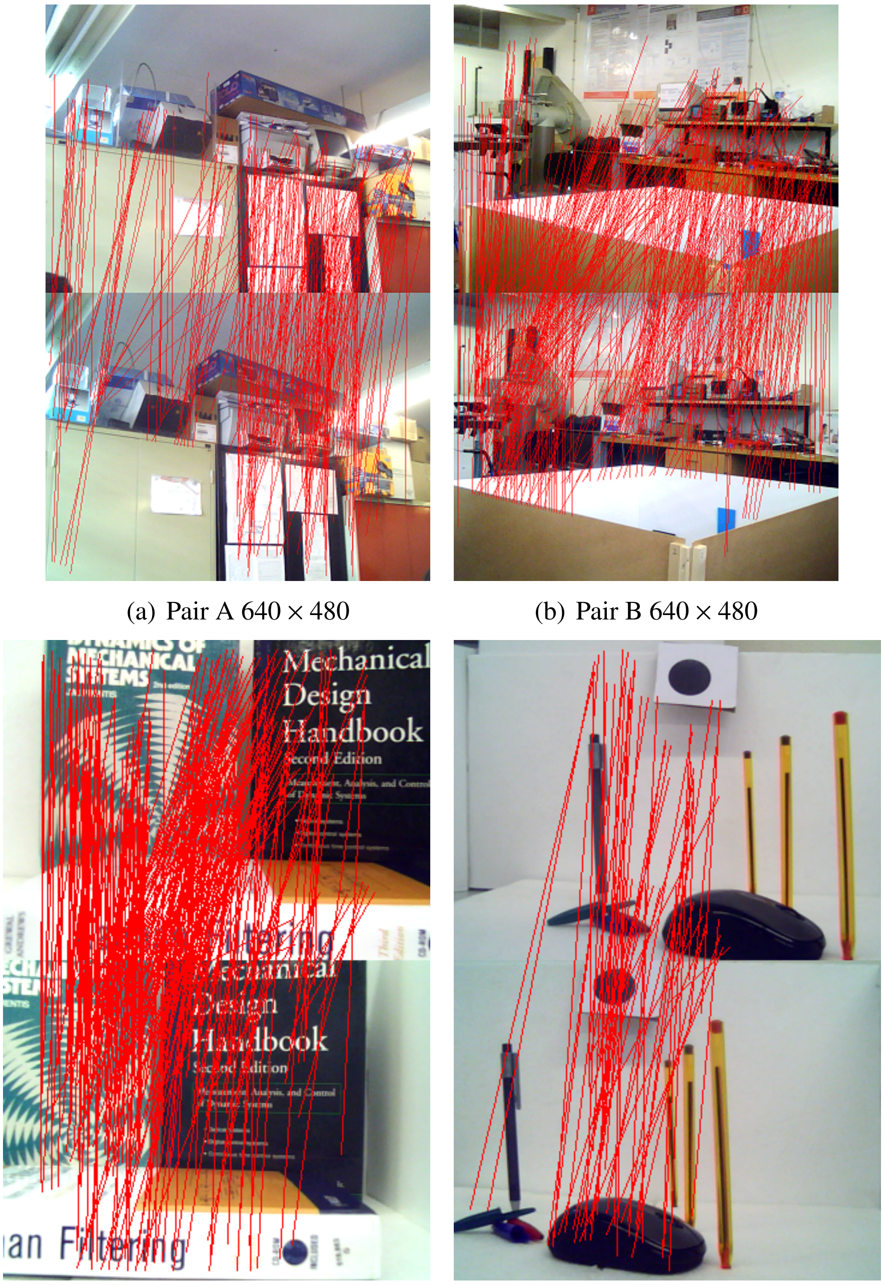
Results

After running the algorithm in the image set we have come to the results found in image 4.3.3 and table 4.3.3.

	Left Points	Time(ms)	Right Points	Time(ms)	Matches	Time(ms)	Total Time(ms)
Pair A	291	110.15	277	113.57	204	6.12	317.33
Pair B	447	111.65	430	115.5	340	10.2	368.49
Pair C	528	30.93	437	34.96	252	10.08	77.1
Pair D	112	28.56	110	30.8	51	4.59	69.23

Table 4.3: SUSAN results

Looking only at the table results, we can say that SUSAN has a far superior performance than SURF or SIFT. But, when looking at the image results, we can see that this speed as a consequence. While the *match lines* appear to be correct both in SURF and SIFT, in SUSAN there appears to be no order. This shows us that although there is a fast computation time the results are reliable. In real time there is also no stability, and with a closer look at the image, matches appear to be incorrect. When using stereo systems an improvement will be done to the matching step.



(a) Pair A 640 × 480

(b) Pair B 640 × 480

(c) Pair C 320 × 240

(d) Pair D 320 × 240

Figure 4.5: SUSAN Image Results

4.4 3D Reconstruction

In the previous section we showed how points were extracted from an image, now we will focus on how to use those points for $3D$ reconstruction. As it has been mentioned several times in this work, we need at least two images to extract $3D$ information from a scene. In this work two approaches were taken to obtain those images: Single camera and Stereo systems. Even though, both will produce two images, the differences between the systems are

obvious. One of those differences and probably the most important for us is that we know that when there is a movement both cameras move. This might not seem like a big deal but let us consider the following: In a single camera system, the difference in the images after a movement is as the movement itself, unpredictable. With a stereo system, in the other hand, no matter what movement the stereo camera does, we know the difference in the images. The advantage that this brings us is that a found point in the left image will always be more to the left on the right image: furthermore the vertical difference can also be threshold. We will explore this advantage when we are implementing the stereo camera approach.
Starting with the single camera approach, we will explain how the found points were used to extract $3D$ information and present the results obtained. Then, we will do the same for the stereo camera approach. In both approaches we will receive two *CvMat*s, one for each image, representing the found points. The organization of the points in both matrices allows to say that the first point in one matrix is the match for the first point in the other, the second is a match for the second and so forth. The first step of both approaches is to have both cameras calibrated. Using uncalibrated cameras is possible, in single camera systems, but results are better when cameras are previously calibrated. In the case of stereo cameras, stereo calibration must be done.

4.4.1 Single Camera Approach

Methods

When using a single camera approach we still need two images. Let us start by defining two *IplImage*s *imageA* and *imageB* and, for the points, two *CvMat*s *pointsA* and *pointsB*. The general workflow is the following:

```
1 – Capture imageA
2 – Find pointsA
3 – While program is running:
    3.1 – Capture imageB
    3.2 – Find pointsB
    3.3 – Match points
    3.4 – 3D reconstruction
    3.6 – pointsA = pointsB
    3.7 – Move camera
```

As we can see, in fact, we do not use two images in each iteration; instead, we only use the extracted points of both images. This is all the information we need from an image making it unnecessary to recalculate the points in both images. Regarding step 3.7, this is an

important step and it must always be done. If one does not move the camera between iterations in step 3.2 the points extracted will be equal to the points extracted in the last iteration. Having equal points means that we have equal images which is almost the same as trying to extract $3D$information from only onw image. For this reason results of the $3D$ reconstruction will not be correct if step 3.7 is not done.
With the process of obtaining two "images" in a single camera system, the process can be divided in 4 implementation steps:

```
1 - Find fundamental matrix cvFindFundamentalMat
2 - Correct Points cvCorrectMatches
    2.1 - For all correspondences
        Compute Corrected correspondence
3 - Projection matrices
4 - Triangulate points cvTriangulatePoints
    4.1 - For each point solve AX=0 using SVD
5 - Convert points cvConvertPointsHomogeneous
```

From the steps above we can see that OpenCV has a method for all the needed main steps. We start with:

```
int cvFindFundamentalMat( CvMat* pointsA ,
                          CvMat* pointsB ,
                          CvMat* fundMatr ,
                          int    method ,
                          double param1 ,
                          double param2 ,
                          CvMat* status=0);
```

Apart from the obvious first three parameters, when using *cvFindFundamentalMat* we must also define a method. We chose *method = CV_FM_RANSAC* which means that the algorithm will use RANSAC. The following two parameters are used as maximum epipolar distance and confidence level thresholds respectively. The last parameter is a matrix created by the algorithm. If the point was rejected the matrix element is set to 0 and 1 otherwise. Rejected points are outliers that usually represent errors and are ignored through the rest of the process. The result will be the fundamental matrix represented by *CvMat* fundMatr*. If no fundamental matrix is found the method will return 0.
After the fundamental matrix is found we need to correct the points according to it. For that we use:

```
    void cvCorrectMatches(CvMat *F,
                              CvMat pointsA ,
                              CvMat *pointsB ,
                              CvMat *new_points1 ,
                              CvMat *new_points2)
```

This method uses the fundamental matrix F to correct all correspondences $pointsA[i] \leftrightarrow pointsB[i]$ according to Hartley's method "The Optimal Triangulation Method" proposed in **??**. The objective of the method is to compute a new correspondence $new\_pointsA[i] \leftrightarrow new\_pointsB[i]$ that satisfies the epipolar condition $new\_pointsA[i]*F*new\_pointsB[i] = 0$. The chosen solution is the one that minimizes the geometric distance between the new and old points.

Next step is getting the cameras matrices. At this step there are two possibilities: we are either using calibrated or uncalibrated cameras. Note that even though we are using only one camera we consider that there are two projection matrices. Since images were taken from different locations and a projection matrix is composed from intrinsic and extrinsic parameters, after a movement the "camera" will be different. When referring to "cameras" we mean the same camera before and after the movement. Using calibrated cameras means that we know the intrinsic parameters (which do not change between movements) of the cameras. Using the method described in section 3.3.3 we first use SVD to calculate the epipoles and then compute the camera projection matrices.

The fundamental matrix was found, the points corrected and we have the projection matrices. Next step is to triangulate the points. Recurring once again to OpenCV we use:

```
cvTriangulatePoints(CvMat* projMatr1 ,
                    CvMat* projMatr2 ,
                    CvMat* projPoints1 ,
                    CvMat* projPoints2 ,
                    CvMat* points4D)
```

Which for each point solves the $AX = 0$ type system using SVD and returns a vector with $4D$ points.

The final step in $3D$ reconstruction is to convert the point homogeneous. This is easily done with:

```
void cvConvertPointsHomogeneous(CvMat* src ,
                                CvMat* dst)
```

Which converts points from $4D$ to $3D$. This list, with $3D$ coordinates, is the main objective of the work. With it we can plot the points to a simulated $3D$ world, create a depth map.

Results

For this section we used the SURF as the point extraction algorithm. We chose SURF for its speed and higher stability and robustness. Extracting 3D information from a scene with only one camera, implies that the camera must be moving. This means that no information is known about the camera after the movement. Looking again at the list of steps:

```
1 – Capture imageA
2 – Find pointsA
3 – While program is running:
    3.1 – Capture imageB
    3.2 – Find pointsB
    3.3 – Match points
    3.4 – 3D reconstruction
    3.6 – pointsA = pointsB
    3.7 – Move camera
```

We can see that there is a cycle in the process that is always running. However, in early stages of the development we started by simulating only one step. For that we only needed two images, of the same scene, taken by the same camera, at different positions. This was done merely as a test, using stable images allows to test the concepts of 3D reconstruction. The first tested images were from Webots[2] and then real world images. The two sample images taken from Webots are shown in figure 4.4.1.

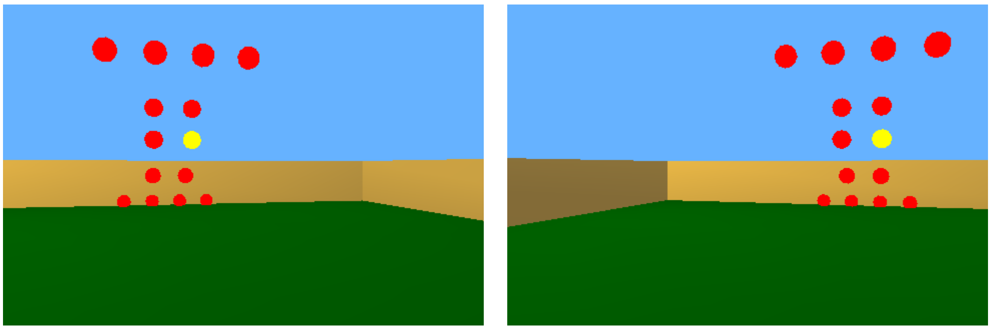

(a) Left camera image (b) Right camera image

Figure 4.6: Webots images

The points obtained were plotted in a simulated world (figure 4.4.1). As we can see, although there are errors, the structure of the real world is recognizable. We can easily see which points are closer to us.As a visual help, the closer the point is to the origin, the whiter

[2] All the positions in Webots are known, which allows us to have a precise "real measure" of the points that we can compare to the results obtained.

it is.

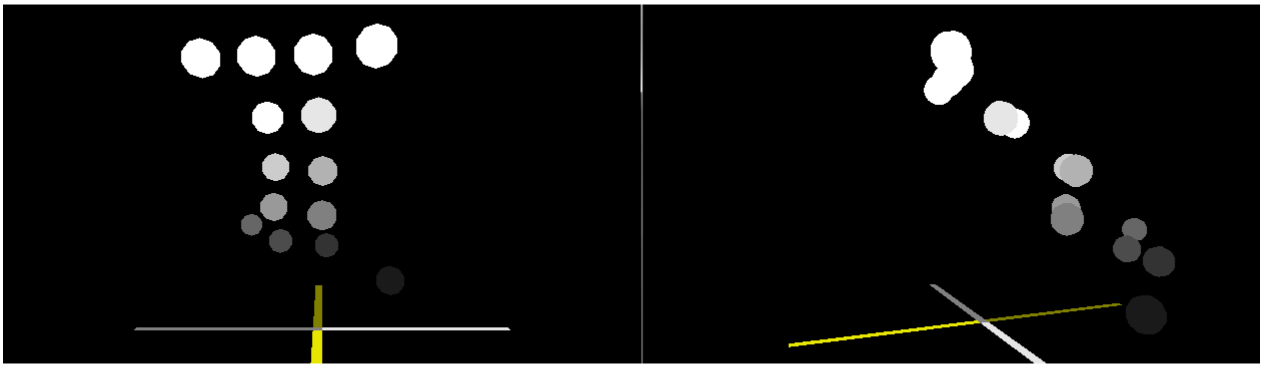
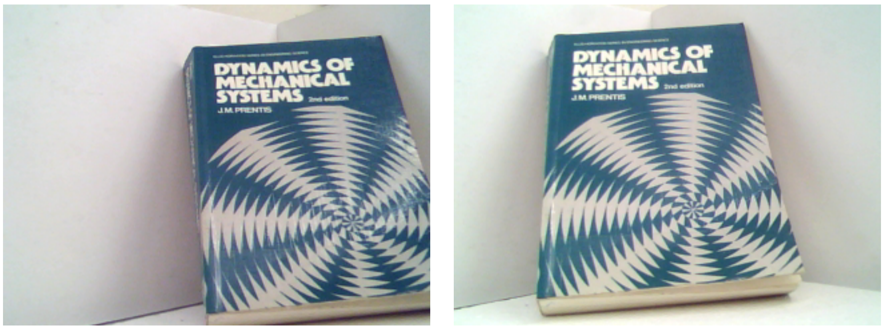


Figure 4.7: 3D results from Webots images

Real images were then used:



(a) Left camera image

(b) Right camera image

Figure 4.8: Real images

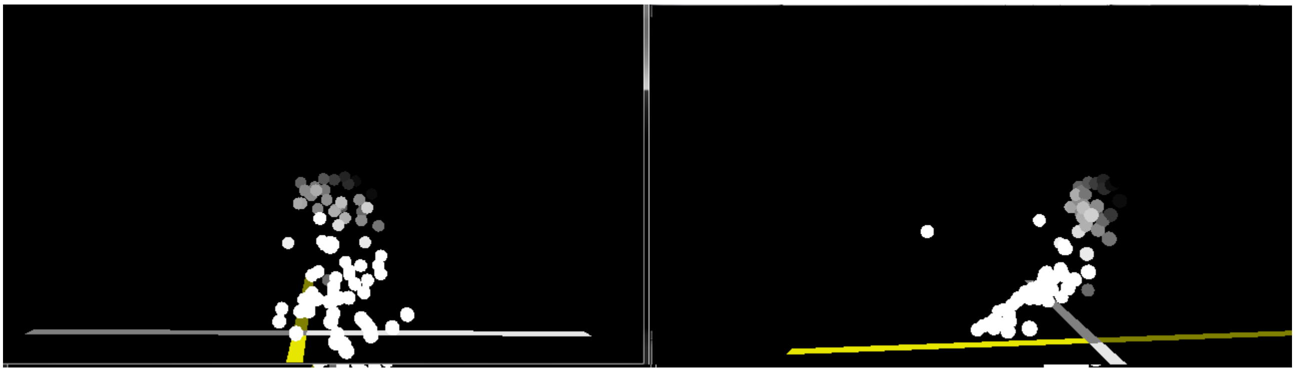
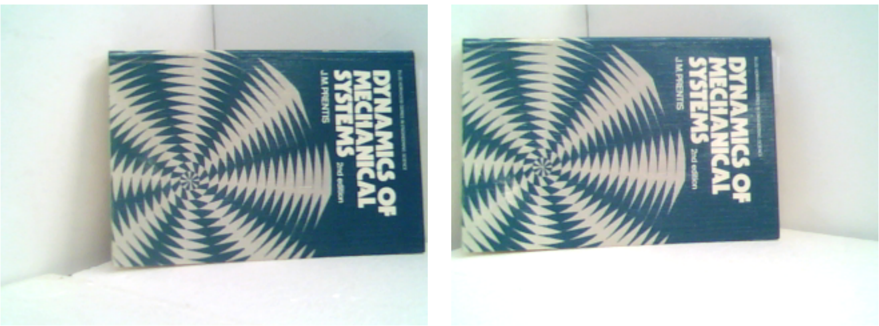


Figure 4.9: 3D results from Real images

Again, even though there are errors, the structure is still visible.

The next step was to try the concept using two equal cameras. The cameras were part of a stereo system but, again, no information about the geometric relation between the cameras was given. Images were taken alternately from left and right cameras simulating a repetitive movement of a camera.



(a) Left camera image (b) Right camera image

Figure 4.10: Stereo camera images

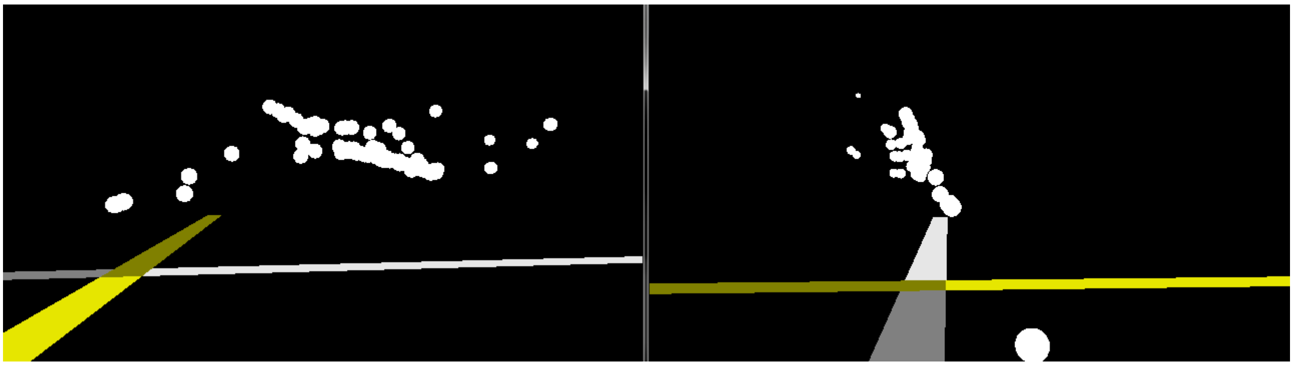


Figure 4.11: 3D results from Stereo cameras images

Figure 4.4.1 shows a sample of the results obtained. The point's location is not stable throughout the running time. At each frame they are plotted at different positions without showing a specific structure. Results showed that, when using real time images, the $3D$ reconstruction performance decays. After a carefully analysis of the process, we have noticed that finding the fundamental matrix from point matches is very sensitive. Errors caused by image noise,mismatches or lack of stability in the SURF algorithm, tend to greatly influence the fundamental matrix which in turn, influences the $3D$ triangulation.

The final stage of the development consisted in using a moving camera. This time the cycle was running completely and without stopping. The results obtained are very similar to the ones using the stereo camera, again there is no stability in the plotting and no structures are defined.

4.4.2 Stereo Cameras Approach

Methods

When showing the results of the SUSAN algorithm, we said that different matching method was used with stereo systems. The method is based on the fact that we know the geometric relation between the two images. They are both vertically aligned and on the same plane. This arrangement of the cameras allows to say that a point viewed in the left image will always be more to the right on the right image. Also the point will have approximately the same y coordinates in both images, due to distortion errors one cannot say that $y_l = y_r$, hence the need for image rectification. Therefore the first step is stereo calibration.

The result of running stereo calibration is each camera's projection matrix alongside with four rectification maps. These rectification maps are obtained using:

```
void cvInitUndistortRectifyMap(CvMat* cameraMatrix,
                               CvMat* distCoeffs,
                               CvMat* R,
                               CvMat* newCameraMatrix,
                               CvMat* mapx,
                               CvMat* mapy)
```

For each camera we have the camera matrix, the distortion coefficients, and the rectification transformation which can be obtained by using *cvStereoRectify* or *cvStereoRectifyUncalibrated*. The result of using *cvInitUndistortRectifyMap* is a pair of rectification maps. When these maps are applied to a pair of images they will be projected to a common plane.

In theory, with rectified images one can say that a point viewed from both cameras will have the same height in both images. However there are always errors therefore we can only say that the point will be around a certain area.
The steps in 3D reconstruction with stereo cameras is the following:

```
3 - While program is running:
    3.1 - Capture imageB and imageA at the same time
    3.2 - Find pointsA
    3.3 - Find pointsB
    3.4 - Match points
    3.5 - 3D reconstruction
```

Both methods, of using either a single or stereo camera, are very similar. Apart from having to capture two images at each cycle, there is also a difference in the 3D reconstruction process. From what we have explained, we know that the matching lines will always point to the same direction in stereo systems. However the inclination of these lines is not the same for every point. With the inclination of the lines, which is the x difference between the projections of both point, we can tell which points are closer to us. As a simple experiment to understand this, if one places an object, like a pencil, in front of the nose and alternatively blinks each eye the object will appear to "move" a lot. If the pencil moves away from the face this movement will be reduced and if it moves closer the movement will increase. The same goes for stereo cameras. We can then say that a point that is closer to the camera will have a big difference in x and vice-versa. The $3D$ reconstruction step is therefore simple measurement obtained from $pointR_x - pointL_x$. This measure is not by any way metric and we can for example threshold it to a certain range of distances. In our tests we used:

```
diff = pointR_x - pointL_x
if(diff<0)
    delete point
if (diff==0)
    dist = img_width
else
    dist = img_width/diff
```

An initial test is done to eliminate those point whose matching lines point to the left. Then we calculate the distance. By dividing *ima_width* by the difference, we will get values between 1, for the closest possible to the camera and 320 to the points furthest away. This method, sometimes called "disparity maps", cannot be used with a single camera since vertical alignment in the images cannot be guaranteed.
With SUSAN we also used a different work flow:

```
3 - While program is running:
    3.1 - Capture imageB and imageA at the same time
    3.2 - Find pointsA
    3.4 - Match points
    3.5 - 3D reconstruction
```

With SUSAN, points are only found in one of the images, more precisely, on the left image. In the matching step, we use the knowledge we have from stereo cameras to find a match for the left points. We start by defining a search area represented by a rectangle. This search area has a top-left corner $rect_x$ $rect_y$ and a size $rect_w$ $rect_h$. Since the correct match will always be more to the right we can start by saying that $rect_x = point_x$ and we either limit the disparity, for example $rect_w = 100$ or give no limits, $rect_w = img_w - point_x$. For the height it would be ideal to say that $rect_y = point_y rect_h = 1$, but since the points will not be perfectly aligned we need to give a certain margin. Therefore $rect_y = point_y - margin$ and $rect_h = margin \times 2$. Having a reduced portion of the image to search for the match point we can introduce heavier search algorithms. We choose to use OpenCV's

```
public void cvMatchTemplate(
         IntPtr image,
         IntPtr templ,
         IntPtr result,
        TM_TYPE method)
```

Where we provide an image where the search will run and, a template, which is a variable sized window around the point. This algorithm slides through the image comparing overlapped patches with the template and saves the results. The more similar matches are found by locating either global maximum or minimum using *cvMinMaxLoc* in the result image. As a form of making the matching process more reliable one can also compute the Kirch descriptor for both the point and result's points selecting the one that has the most similar descriptors.

Results

We start by showing the results of using the new SUSAN matching algorithm (figure 4.4.2). Note that the images are rectified. 4.3.3.

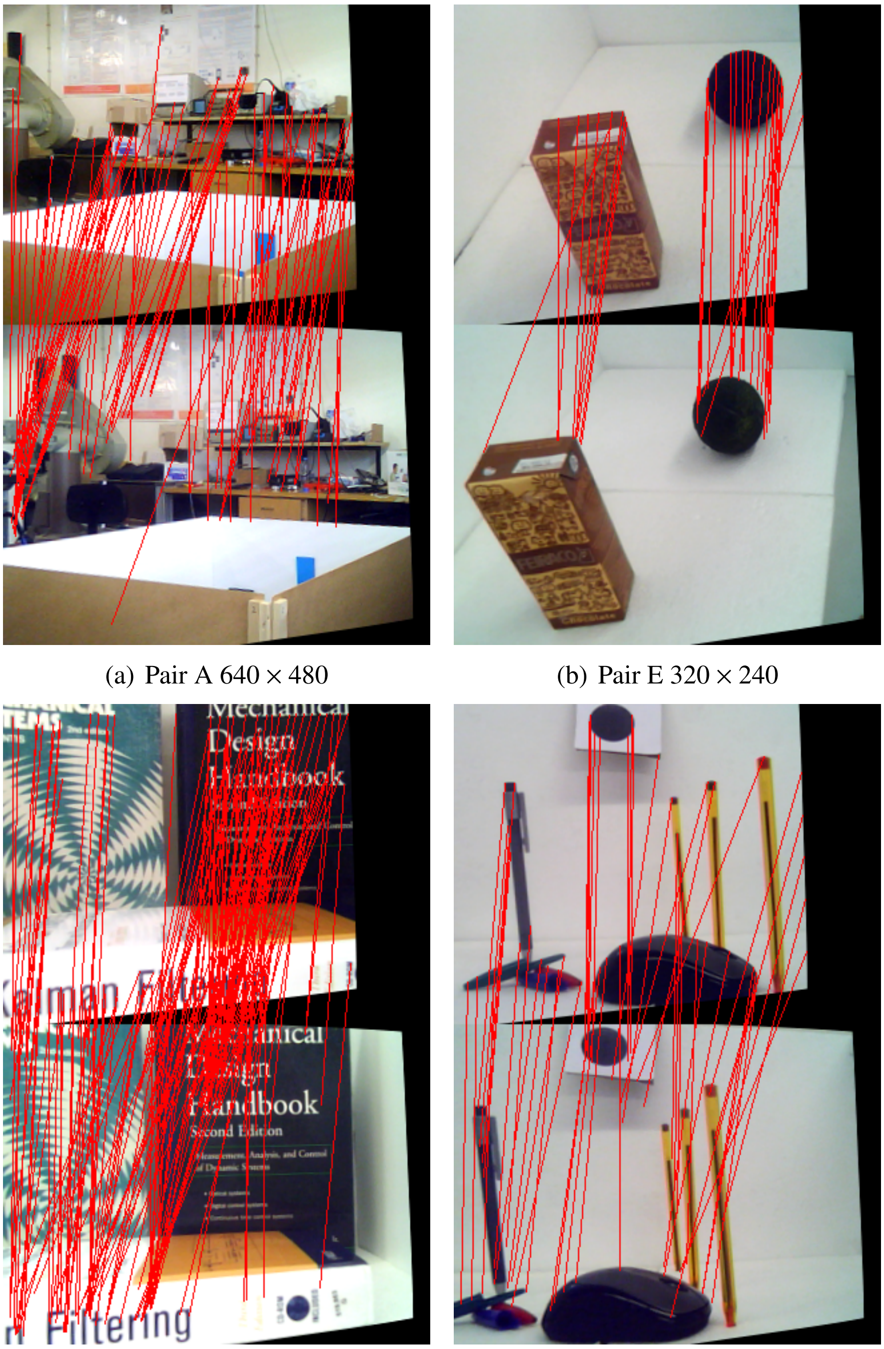
From figure 4.4.2 we can see that there was some improvement in the matching. Although results are still inferior to the other algorithms, specially in more complex and higher resolution images, in simpler images, such as figure 4.12(b), the results are acceptable. If speed is the most important aspect of the algorithm or if the system has small computational power (such as the simplest robots), SUSAN is a valid option. In controlled environments for basic obstacle detection, results obtained with SUSAN are very similar to results obtained with SURF or SIFT. We can see an example of such environments in figure 4.4.2.

Now we will present the results obtained with the stereo implementation showed above. The same test images, used when testing single camera reconstruction, will be used again. This allows some comparison between both methods. As when using a single camera, tests started by using Webots. Points were plotted in a $3D$ world where their coordinates were simply the images coordinates, for x and y and the disparity between both images for z.

Next with real images:

As we can see from figures 4.4.2 and 4.4.2 results, are very similar between stereo and single cameras.

Moving to a real time reconstruction, at each test we change the the positions of the objects in the scene. This time, instead of plotting the point into a simulated world, we will place the points in the image and change their color according to their distance.



(a) Pair A 640 × 480

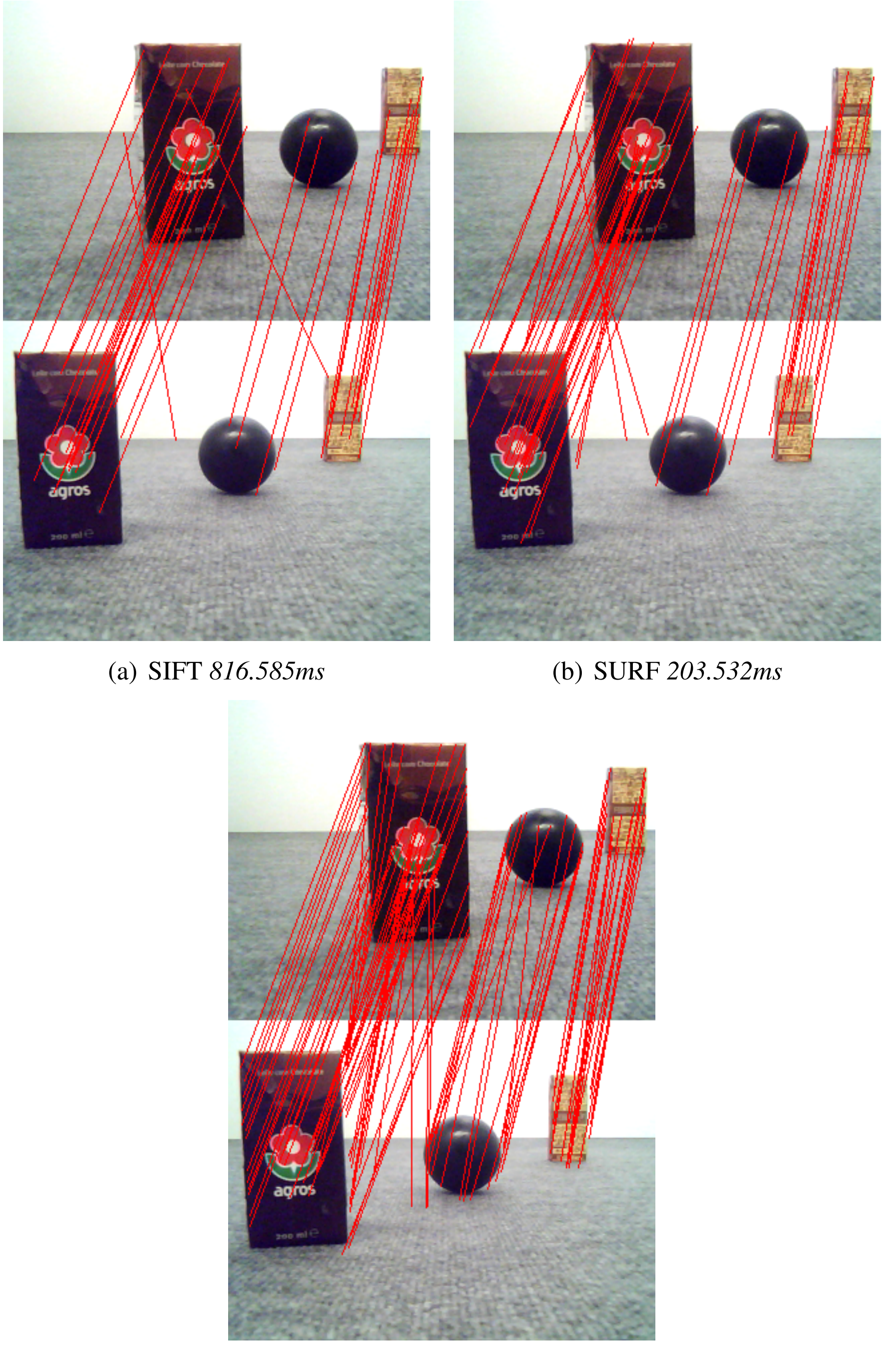
(b) Pair E 320 × 240

(c) Pair C 320 × 240

(d) Pair D 320 × 240

Figure 4.12: SUSAN Image Results with better matching algorithm

In figure 4.4.2 we can see three frames where one of the objects changes position while the other does not. The matched points are marked in the image as red dots. Depending on the distance of the point to the camera, it's color will go from red to white, where white represents the closest points to the camera. Let us consider the stopped object in the left *Object A* and the moving object in the right *Object B*. At the first frame object *B* is clearly further away than *A* therefore the points in *A* have a whiter level. At the second frame both



(a) SIFT *816.585ms*

(b) SURF *203.532ms*

(c) SUSAN *99.66ms*

Figure 4.13: Control environment SIFT SURF and SUSAN

objects are at similar distance as we can see in the similar color level in the points. Finally at the last frame, object B comes closer than A, points in B become white while, points in A gain a red color.

In figure 4.4.2 we can see another sequence of frames. This time, all objects changed their position and we used another method of showing the $3D$ information. We start by find-

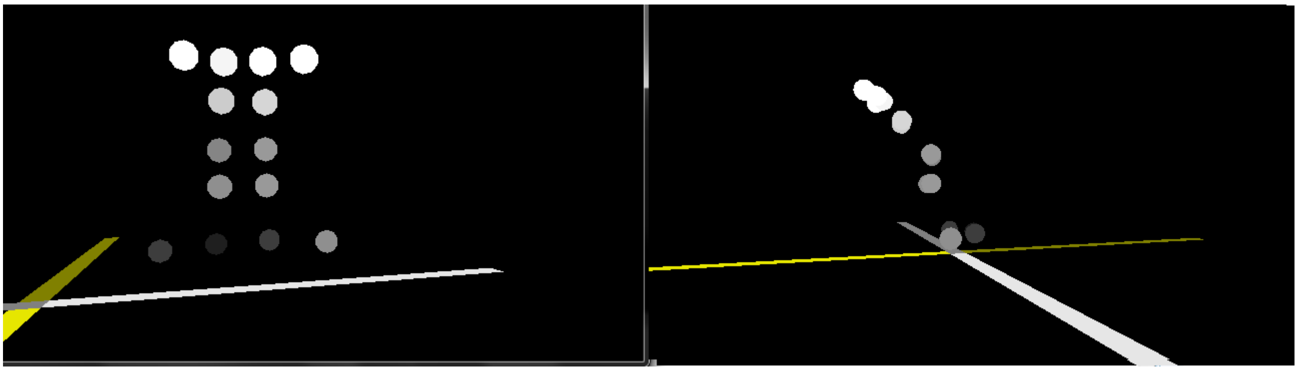


Figure 4.14: Stereo reconstruction with Webots images

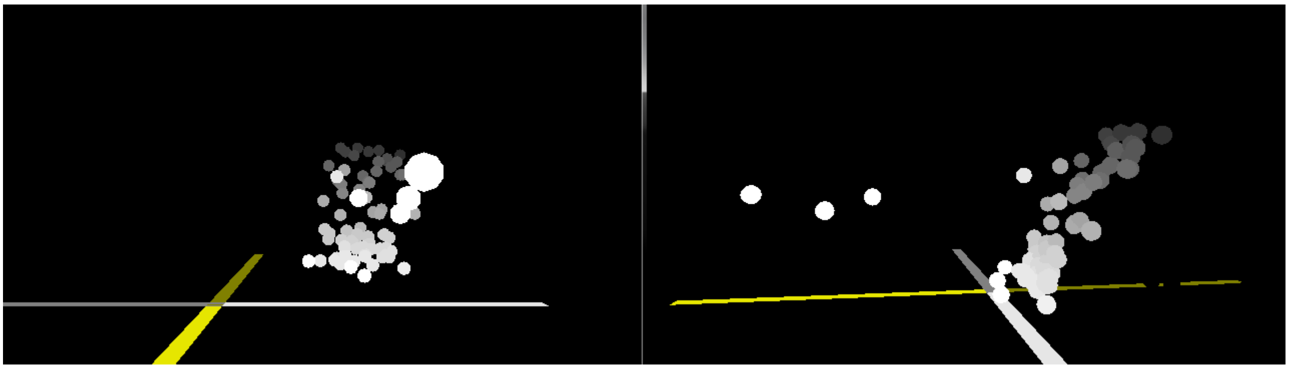
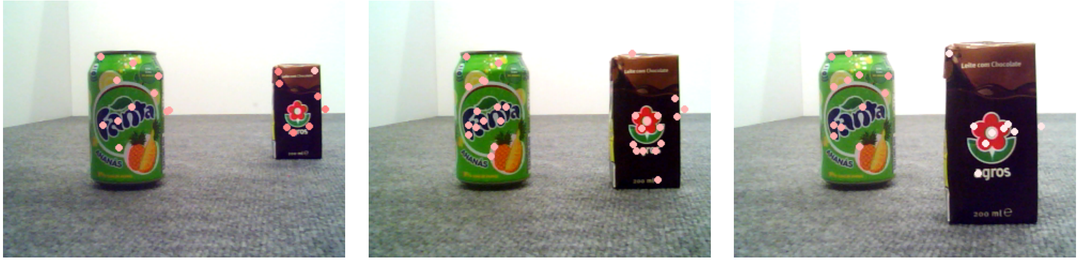


Figure 4.15: Stereo reconstruction with real images



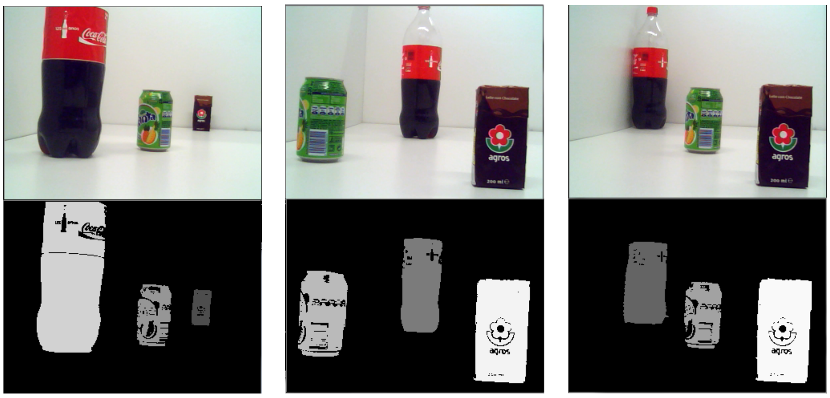
(a) Frame 1

(b) Frame 2

(c) Frame 3

Figure 4.16: Distance plotting in moving object.

ing the contours of the image, then, assuming a contour is an object, for each object we see how many matched points are inside the area of the object. Averaging the disparity of the found points we get the estimated distance of the object. In all images we can see which object is closer to the camera (whiter color) and further (darker color).



(a) Position 1 (b) Position 2 (c) Position 3

Figure 4.17: Distance in detected objects.

Chapter 5

Conclusion

At the beginning of this work three main objectives were proposed and different solutions to the various problems analyzed. For the first objective of extracting defining points from an image, three algorithms were analyzed. Then, for each algorithm at least one matching method was implemented. Finally, with the points extracted and matched, two different ways of extracting $3D$ information were also proposed.
For point extraction SIFT, SURF and SUSAN algorithms were tested. For SIFT, Kd-trees were used for the matching stage and results showed that, considering only the quality of the matching, SIFT has the best results. Even though SURF has similar results both in stability and quality of matching, in some cases, SIFT has more stability. With SURF, optimized algorithms in nearest neighbor searches were used to find the matches. In terms of results, they were similar to SIFT but much faster. In terms of speed, SURF produces results in less than half of the time of SIFT. Finally, two different methods for matching the SUSAN points were tested. While the first method produces un-usable results, the second method, even though it can only be used in stereo systems, produces better results. Generally, SUSAN has inferior matching results but, in certain cases we can say that results are as good as SURF or SIFT. The advantage of SUSAN is its speed. Even with a more complex matching process SUSAN is still faster than any other tested algorithm. We have also learned that, for all algorithms, the more complex the texture is, the more points we get. This is the reason why most of the objects used in tests had a complex texture. If only "simple color" objects were used, the number of found features would be minimum, which is a problem for RANSAC. Remembering the objective of RANSAC a high number of points is important. To say which of the algorithm is better depends on where it is used. If, for example, computational power is not a problem and neither is the speed, we would recommend the use of SIFT. Has an opposite, if we have low computational power and real time results are mandatory, SIFT is not an option. In the same situation, if talking about a controlled environment, SUSAN produces valid results at far higher speeds than SURF. For generic-situations SURF is recommended.

For the $3D$ reconstruction, when using a single camera, valid results were not achieved. While with still images, the $3D$ information was understandable, with real time images no valid information was extracted. We can say that one of the most important steps is the extraction of the fundamental matrix, which in turn, has proven to be very sensible to errors. Again, there are always matching errors and, even when using RANSAC with a higher minimum confidence, we can say the fundamental matrix is never correctly extracted. The matching errors do not always fall on the algorithm, unsteady camera images has been a problem throughout the project which is due to a provable inferior quality in the camera. The last part of the project was the implementation of $3D$ reconstruction with stereo cameras. Using a simple measuring procedure, we are able to extract valid results from an image. Even though we cannot measure exactly were a point is, we are able to tell which object is closer to the camera, and an approximated knowledge of how far it is.

Bibliography

[1] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, pages 404–417, 2006.

[2] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: speeded-up robust features. In *9th European Conference on Computer vision*, volume 110, pages 346–359, 2008.

[3] P. Beardsley, A. Zisserman, and D. Murray. Navigation using affine structure from motion. *Computer VisionŮECCV'94*, pages 85–96, 1994.

[4] D.C. Brown. Close-range camera calibration. *Photogrammetric engineering*, 37(8):855–866, 1971.

[5] M. Brown and D.G. Lowe. Invariant features from interest point groups. In *British Machine Vision Conference, Cardiff, Wales*, pages 656–665. Citeseer, 2002.

[6] F.C. Crow. Summed-area tables for texture mapping. *ACM SIGGRAPH Computer Graphics*, 18(3):207–212, 1984.

[7] K.G. Derpanis. Overview of the ransac algorithm. 2005.

[8] D.A. Forsyth and J. Ponce. *Computer vision: a modern approach*. Prentice Hall Professional Technical Reference, 2002.

[9] D.A. Forsyth and J. Ponce. *Computer vision: a modern approach*, volume 54. Prentice Hall, 2002.

[10] A. Haar. On the theory of orthogonal function systems.)). *Mathematische Annalen*, 69:331–371, 1910.

[11] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*. Cambridge Univ Pr, 2003.

[12] R.I. Hartley. Theory and practice of projective rectification. *International Journal of Computer Vision*, 35(2):115–127, 1999.

[13] R.I. Hartley and P. Sturm. Triangulation. *Computer vision and image understanding*, 68(2):146–157, 1997.

[14] H.M. Kakde. Range Searching using Kd Tree. *Journal of*, 2008.

[15] K. Kanatani. *Statistical optimization for geometric computation: theory and practice*. Dover Publications, Incorporated, 2005.

[16] K. Kanatani, Y. Sugaya, and H. Niitsuma. Triangulation from two views revisited: Hartley-sturm vs. optimal correction. In *Proc. 19th British Machine Vision Conf*, pages 173–182. Citeseer, 2008.

[17] Y. Kanazawa and K. Kanatani. Reliability of 3-d reconstruction by stereo vision. *IEICE Transactions on Information and Systems*, 78:1301–1306, 1995.

[18] R.A. Kirsch. Computer determination of the constituent structure of biological images* 1. *Computers and biomedical research*, 4(3):315–328, 1971.

[19] K. Levenberg. A method for the solution of certain problems in least squares. *Quarterly of Applied Mathematics*, 2:164–168, 1944.

[20] T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision*, 30(2):79–116, 1998.

[21] M.I.A. Lourakis. A brief description of the levenberg-marquardt algorithm implemented by levmar. *matrix*, 3:2, 2005.

[22] D.G. Lowe. Object recognition from local scale-invariant features. In *iccv*, page 1150. Published by the IEEE Computer Society, 1999.

[23] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[24] Q.T. Luong and O.D. Faugeras. The fundamental matrix: Theory, algorithms, and stability analysis. *International Journal of Computer Vision*, 17(1):43–75, 1996.

[25] D.W. Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the society for Industrial and Applied Mathematics*, 11(2):431–441, 1963.

[26] S.M. Smith and J.M. Brady. SusanŮa new approach to low level image processing. *International journal of computer vision*, 23(1):45–78, 1997.

[27] R. Tsai. A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses. *Robotics and Automation, IEEE Journal of*, 3(4):323–344, 1987.

[28] M. Wall, A. Rechtsteiner, and L. Rocha. Singular value decomposition and principal component analysis. *A practical approach to microarray data analysis*, pages 91–109, 2003.

[29] Z. Zhang. Flexible camera calibration by viewing a plane from unknown orientations. In *iccv*, page 666. Published by the IEEE Computer Society, 1999.

[30] Z. Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000.