



ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS  
DEPARTMENT OF INFORMATICS



***DRAFT / WORK IN PROGRESS***  
**Author : Ammar Qammaz**  
**Supervisor : Georgios Papaioannou**

Athens , April 2012

# **Introduction and motivation**

## **Goal**

## **Overview**

### **1 Mathematical Framework**

*1.1 Camera Pinhole Model*

*1.2 Camera Calibration*

*1.3 Image Rectification*

*2.1 Image Processing*

*2.2 Corner and Feature Detection*

*2.3 Template Matching and Integral Images*

*2.4 HAAR Wavelet based Face Detection*

*3.1 Epipolar Geometry*

*3.2 Binocular Disparity Depth Mapping on a parallel camera setup*

*4.1 Homography Estimation*

*4.2 RANSAC*

*4.3 Optical Flow*

*5.1 Dead Reckoning*

*5.2 Simultaneous localization and mapping*

*5.3 A\* Pathfinding*

*6.1 First Order Logic and a Wumpus like World*

*6.2 The big picture*

### **2 Hardware**

*2.1.1 Overview*

*2.1.2 Camera Sensors and Synchronization issues*

*2.1.3 Motor System and Peripherals*

*2.2.4 Embedded System Notes*

*2.2.5 The Energy – Weight - Heat – Cost Problem*

*2.2.6 Guarddog Part list / Specifications*

### **3 Software Stack**

*3.1.0 Overview*

*\*Unified String Interface*

### **4 Future Work**

*\*Network Connectivity – Encryption over RF*

*\*NLP – AI Knowledge Base*

*\*Face / Speech Recognition*

*\*Physics Simulation*

*\*Commercial Personal Robots*

*\*Low Level Assembly ( MMX/SSE3 ) optimizations*

*\*CUDA / VLSI acceleration*

*\*Car sized guarddog or "CardoG"*

## **Acknowledgements**

## **Bibliography/References**

START

## **Introduction and motivation**

### **A few opening remarks**

Humans increased their physical power during the industrial revolution using machines. They were able to create giant dams , factories , cars , airplanes and skyscrapers to make their everyday life easier . Technology has continued to improve exponentially and in the current age , labeled by some as the age of informatics or the internet , mental capabilities where multiplied. Merging the following two revolutions we can finally partly replace ourselves from dull and repetitive tasks of day to day life that will gradually stop to trouble the human kind leading to a more pleasant life. The GuarddoG project is about making machines that can see and act as a futuristic private guard .

The process of creating an autonomous robot that can perceive its environment and react and interact with it took nature millions of years. From the first bacteria to multi cell organisms , the wolf then the dog and the human , enormous evolutionary differences created beings of immense complexity and perfection. For someone to build something that took such a great amount of time in even a quarter of a lifetime is over-ambitious. An extra observation that is thought provoking is that while humans in complex decision making such as chess playing or tactic games with a limited set of rules have been surpassed by computers. In contrast in simple things for humans such as perceiving space , time , and “natural logic” every human has an innate superiority a result of the millions years of natural selection with these characteristics as a basis.

That being said GuarddoG does not attempt to create a dog ( with everything a dog implies ) , because this is practically impossible. Its goal is replacing a specific function of a dog as a guardian. I am very optimistic that with time robots will eventually be improved enough to be able to perform a multitude of tasks approaching something that will surely be different than a real dog , better at some things , and worse at some others.

Even though the future will offer even more tools , even now thanks to the marvelous technology and work of all the scientists , mathematicians , physicists , chemists , engineers and computer scientists ( we are literally standing on the shoulders of giants ) I was able to construct something very close to my original target , spending a fraction of the money and time that would be required before 15 or even 10 years.

A better way for someone to visualize the small subset of functionality that is attempted by computer vision algorithms and in this case GuarddoG , is to compare it to the holy grail of cognition and intelligence , the human brain. Though computers for many years have managed to surpass human experts on tasks like playing chess , remembering sequences of numbers , performing arithmetic calculations on large data sets and recently even guessing questions to answers ( IBM Watson on the Jeopardy TV Show ) , things that everyone can do without even thinking about , like walking , identifying 3D objects and faces , and coordinating his head , eye and body movement are currently unachievable by machines at least to the extent of human performance .

This is a good indicator of the level of optimization that has taken place through the millions of years of evolution , because there is no doubt that if playing chess was a trait that leaded to natural selection the human brain would be totally different and have a much greater affinity towards these kind of activities. On the other hand , if breathing , beating the heart or walking and identifying objects wasn't crucial for the survival of the human species , to master these kind of activities could may well be as difficult and time consuming to be achieved as mastering chess .

\*

## **Project Goal**

### **The goal of the "Guard Dog" Project**

The goal of the Guard Dog Project is to build a robotics platform that can act as a guard , traverse a known path and fend off intruders. In case of a security breach it would signal the alarm and begin to follow the perpetrator and after a set distance would resume its previous path.

The goal of this document is to give a clear and concise look of the algorithms , methods and parts that should be employed to achieve this. Needless to say it was a very challenging effort discovering these first hand during development , but compiling them in this document in a coherent form of reasonable size is also proving to be very difficult. For many of the things that are mentioned in a paragraph or a page , one could go on writing hundreds of pages or even books to fully justify and explain them , and indeed hundreds and thousands of pages have been written for them. Wherever needed , the provided references will help the reader better understand the concepts or study them in more detail.

Robotics and computer vision are not a new domain of computer science and electrical engineering. It was especially shocking for me to see video footage of experiments in the AI Lab of Stanford ( for example Les Earnest and Lou Paul and the Rancho Arm ) circa 1971 that perform object detection , complex decision making and that actually use more or less the same algorithms as current robotics projects do. The major difference is not so much about the methods used , but the exponential improvement on computer hardware , popularly coined as Moore's Law.

We are living in times where many high-end mobile phones actually have more complex processors than the satellites of the first mission to the moon and that experiments such as those that required equipment that cost millions of dollars in 1971 and could only be done in universities or government research centers can be reproduced with consumer electronics readily available everywhere. Unfortunately the consistent computation of the world around a robot is still a very difficult and expensive task with a generic CPU and no specialized hardware , but yet it seems almost feasible when you achieve even something that can work 10 times slower than a human.

Of course an additional goal of the project is to perform guard duties using only cheap building blocks but not passive sensors as most commercially available security systems do. Instead building a semi-intelligent agent that can do this job the way humans would do it. It is an exploration of the possibilities and limits of current technologies along with software that can leverage the capabilities of computer hardware in an efficient way, to achieve it.

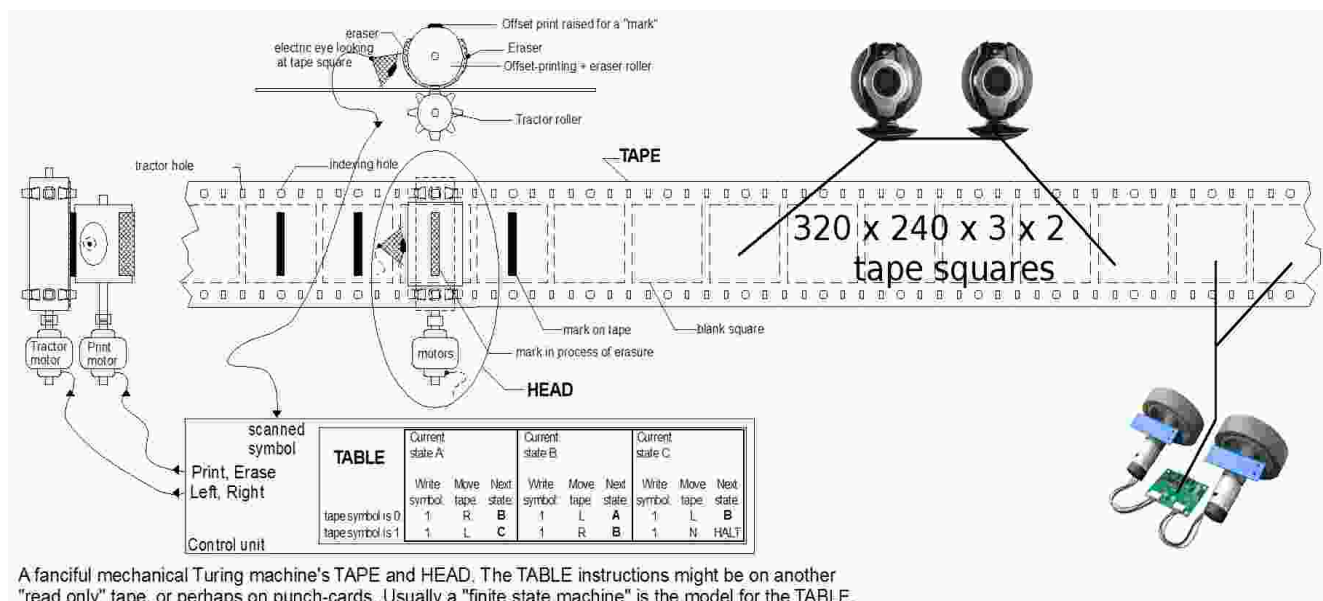
It is also interesting to note that the same computer vision libraries , could in principle and with some necessary adjustments be fitted for tasks like driving cars in city streets to helping blind people find their way or any task that involves using optical information of ones surroundings to achieve a related goal. The main difference would be the risk/cost and risk/performance ratio since a computer driving a car at full speed can do much worse damage when compared to a small robot bumping on a wall.

# Overview

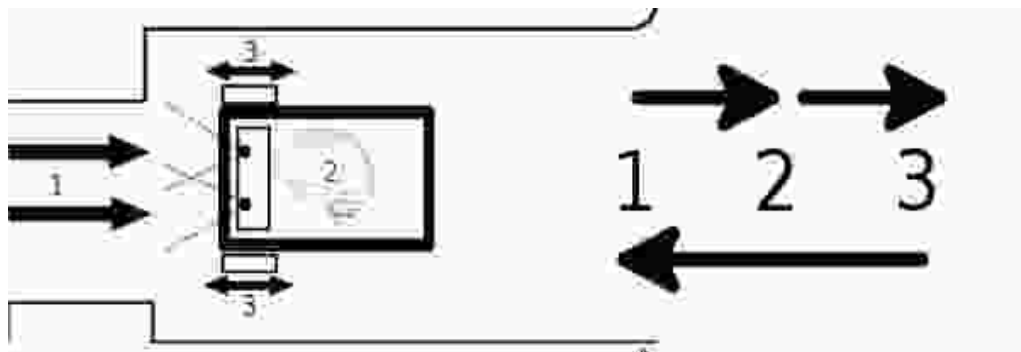
## An outline of this text to help the reader

The ease with which humans sense the world makes the problem of computer vision seem “easy” to solve. In fact the way we see is so natural and persistent that even scientists in the field made biased over-optimistic predictions about it. The fact is that despite the exponential growth in computational speed , and although there is a very big market that could certainly use vision algorithms to automate tasks , there is still no defacto algorithm that can compare to what human vision performs. Moreover from simple reflexes as maintaining focus and coordinating ones gaze , reading text , to tracking your position in an unknown city , vision seems to be “AI-Complete” , since understanding and combining what is seen is an altogether different task than the small building blocks which are presented here.

A robot that can see and interact with the world , is basically a Turing machine on wheels. Therefore the whole model presented here is an adaptation of different mathematical concepts and a fusion of them together. The strip of tape in this Turing machine is constantly filled with symbols of light intensity as the light gets reflected and activates the camera sensor elements. When the control algorithm decides that the robot has to move it writes it to the according tape elements and the motors move , producing a new view of the world.



The first thing to take into consideration beginning to approach this problem , is how the physical world is being represented by the cameras. They are , after all , the means with which the GuardddoG/RoboVision algorithm collection , a “meta”physical entity can take a peek into reality. The data acquired must then be filtered to remove deformations and distortions that may corrupt the whole process. These steps are described in the Camera Model , Camera Calibration and Image Rectification parts of this document. Once two corresponding images of the projection of the world on the camera sensors are aquired , they are examined for optical cues that reveal the details of the world in three dimensions and also the robot's position. This is also discussed extensively , and the Disparity Mapping algorithm used by GuardddoG is a new implementation. When all these steps are finished , the next one is tracking the position of the robot ( LK Optical Flow , RANSAC Homography ) and the combination of the successive 3d Views together ( SLAM , Obstacle Detection ). The final piece of the algorithm is a knowledge base that will set its goals and keep the state of the world , and steer the robot towards achieving them. For GuardddoG , its goal is the traversal of a standard path , and raising the alarm if a breach is found.



*Illustration 1: The chicken and egg problem nature of an autonomous robot , that with its action changes its perception of the world , and with the changing perception of the world it changes its action..!*

Beginning to make a system that sees , one can make many choices about the way with which to gather input. As nature teaches us , and by bringing to mind various insects and animals that have been optimized through a process of millions of years to see one might use anything from ultrasonic sounds , to millions small eyes of insects up to human stereoscopy. With the world represented through the camera being so chaotic , and as this project does not deal with a fixed environment in which to be operated , while also having economic restrictions applied the best choice was a human like stereoscopic camera input. It is true that commercial RGB+depth cameras such as Microsoft Kinect can bypass a very big portion of the computational complexity of this project , but they still have their own drawbacks. The stereoscopic setup wasn't chosen by accident by nature , and the nature of a robot that uses stereoscopic vision makes it closer to the human experience as a mode of viewing the world.

Illustration 2: What computers see

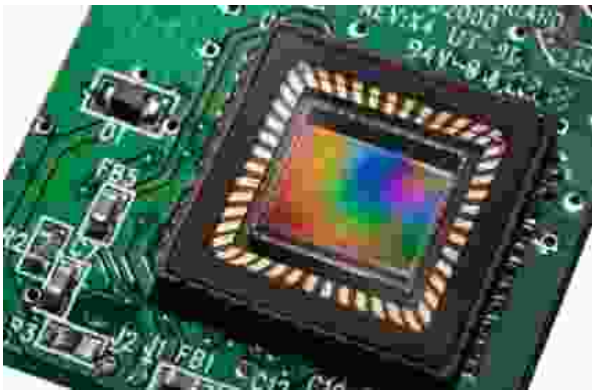
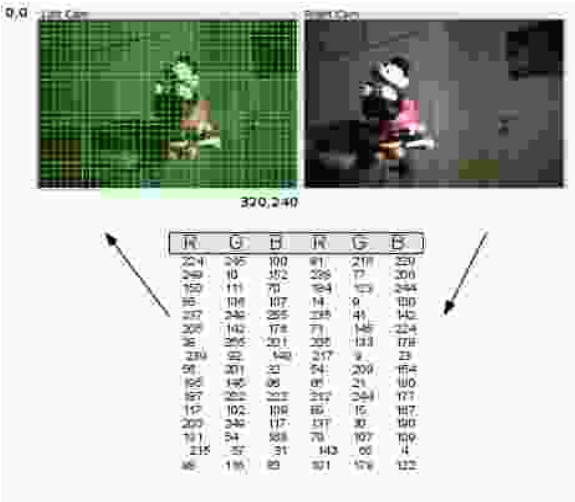


Illustration 3: CCD sensor

Trying to approach the computational limit of a dense stereoscopic method for two frames sized 320x240 pixels in order for a full search from an image patch sized 40x40 pixels on the left eye to all the possible matching patches along the epipolar line on the right eye , we have to make  $320 \times 320 \times 240 / 40 = 24576000 / 40 = 614400$  operations in the worst case each time we get a depth map. In order to achieve a “human like” response time from the vision system this has to be done at a rate of 25 frames per second , or with a delay of 40 milliseconds per scan..

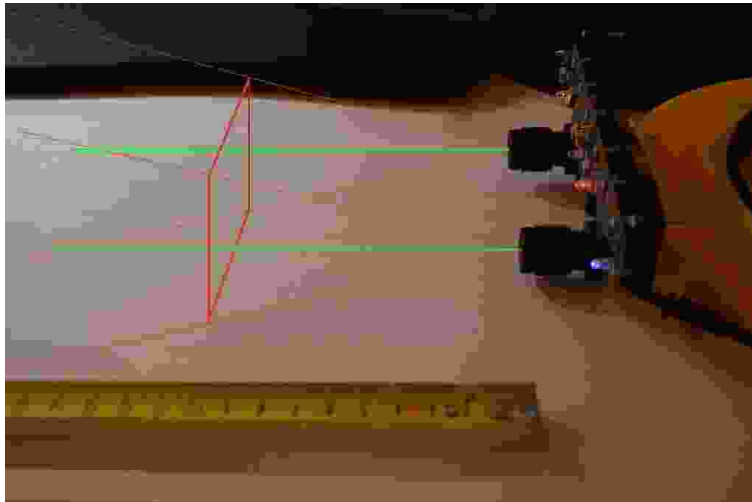
The number of operations per second increases exponentially as the image size becomes larger

SIZE	IMAGE RESOLUTION	OPERATIONS	OPERATIONS PER ms
QVGA	320 x 320 x 240 / 40	24,576,000	614,400 operations / ms
VGA	640 x 640 x 480 / 40	196,608,000	4,915,200 operations / ms
XGA	1024 x 1024 x 768 / 40	805,306,368	20,132,659 operations / ms
...	Other Configurations	...	...
WUXGA	1920 x 1920 x 1024 / 40	3,774,873,600	94,371,840 operations / ms



This exponential increase , of course , impacts all the algorithms used on the project , and for every operation there are numerous sub operations implied so the total maximum number of operations ends up being many times larger than the numbers on this table. All the algorithms on the other hand do a better job than this worst case scenario , and specifically the disparity mapping algorithm of GuarddoG , which is one of its novel aspects and is briefly presented in this text . To reduce the number of operations by design , and as an early measure to compensate for the cheap hardware that is used by the onboard computer the resolution of images used by default is QVGA ( 320x240 pixels ) .

Manufacturing a physical stereo rig for the experiments which is perfectly aligned has a crucial effect on the calculations. Not only it increases computational efficiency and reduces errors but it also removes mathematical ambiguity about instances of the world that can be interpreted in many ways.. The relative position of the GuarddoG cameras is supposed to be constant and the two cameras always have a coplanar alignment with a fixed distance between the optical centers. The cameras are also never allowed to change their focus ( nor could change it as they do not have an automatic focus control ).



*Illustration 4: The fixed parallel camera rig , that GuarddoG uses*

To avoid re calculations and use of the CPU for reasons avoidable by better designed algorithms or a smarter implementation , the whole vision library uses a pipelined architecture , so that the same image will not have to pass a processing stage twice once it enters and according to the needs of the Robot Hypervisor the different stages try to be combined , or operations stay pending for the next frame.

The pipeline itself , a term that is frequently mentioned in this document , is an abstract term meaning the whole library collection and the final program which when executed receives input from the cameras , channels it and processes it and then using the motor system steers the whole platform to achieve the set goal .

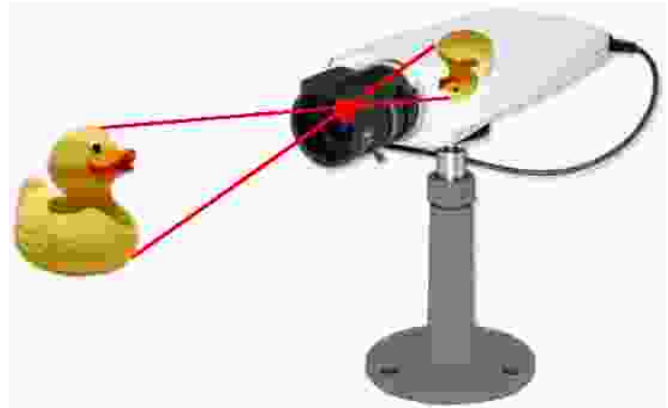
The purpose of this document is to describe and analyze this pipeline and it is organized in five parts , each of which is dedicated to a certain aspect of it. The first stage is to analyze the mathematical background of the algorithms , why they were chosen and why they should in theory work discussing performance issues from a complexity viewpoint .

The second part focuses on hardware and technical details , along with performance statistics for different hardware setups. The third one explains the various tactics followed writing the software and how everything fits together on the resulting software stack. Part four discusses about the system in practice , its performance and limitations on actual deployment , and the fifth and the last chapter for future plans for an even better implementation and self-reflection ( 反省 [citation needed] ) about the project.



# Mathematical Framework

## 1.1 Camera Pinhole Model



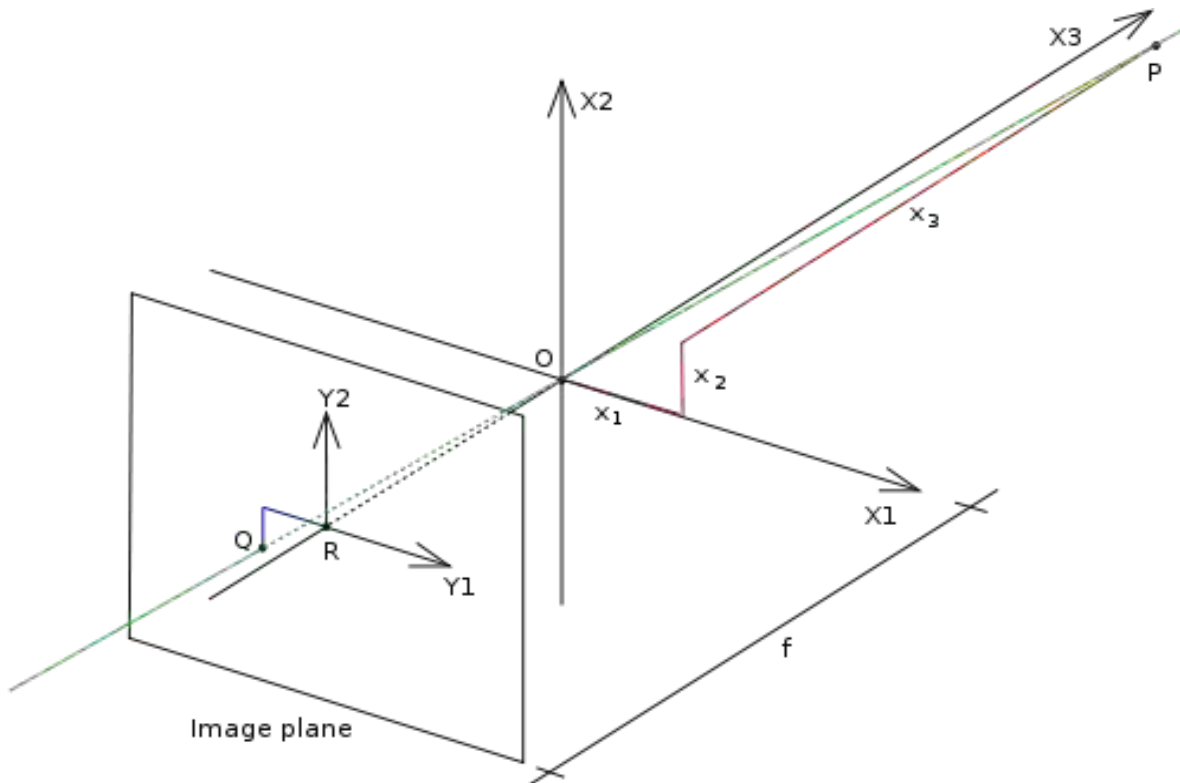
A pinhole camera is a light capturing device without lens and a very small aperture. Regardless of the imaging sensor , the shutter system , or the integrated circuit on camera , it is fundamental to understand the physical model and how light gets projected on the sensor , in order to start to reverse engineer the physical process that creates the data we must later process.

The smaller the hole of the camera aperture , the less light rays pass through it and the sharper the image gets , but as the hole size decreases , so does the total number of photons that pass through it, resulting in a dimmed image for short exposures. Regardless of the number of picture elements or the mechanism that captures the light the most fundamental laws that govern vision are mathematics and in our case Euclidean geometry.

Of course besides mathematics , optics and physics explain other implied principles such as why light propagates in a straight line the way it does , why objects radiate a specific electromagnetic spectrum frequency and other interesting details that govern the procedure. Since a domestic robot will not encounter gravitational lens , refractions due to heat or liquids these are details that can safely be omitted .

However a light bending phenomenon , although far less exotic than the previous stated ,does indeed impact the procedure. Due to manufacturing inefficiencies in the shape and substance of the camera lens , the projection on the camera sensor gets distorted. The distortion , depending on the quality of the methods employed by the factory that makes the camera can be so great that the generated image may become unusable without additional processing. This problem is discussed in the calibration and re sectioning part of this document ( the next topic ) , that aims to measure and repair the distorted image making it fit to the ideal pinhole camera model described here.

The pinhole camera model applies to most consumer grade web cameras and it is a very useful tool both for this usage scenario as well as for understanding more complex camera topics such as zoom lens focus changing and many others



*Illustration 5: The pinhole camera model , illustration from Wikipedia , public domain*

The point O is where the camera aperture is located , and the start of the axes. The three axes of the coordinate system are referred to as  $X_1$ ,  $X_2$  and  $X_3$ . Axis  $X_3$  points in the viewing direction of the camera and is referred to as the optical axis, principal axis, or principal ray. The 3D plane which intersects with axes  $X_1$  and  $X_2$  is the front side of the camera, or principal plane.

The reflected rays from the world end up on the image plane , a two dimensional plane which is parallel to axes  $X_1$  and  $X_2$  and is located at distance  $f$  from the origin O in the negative direction of the  $X_3$  axis. A practical implementation of a pinhole camera implies that the image plane is located such that it intersects the  $X_3$  axis at coordinate  $-f$  where  $f > 0$ .  $f$  is also referred to as the focal length of the pinhole camera.

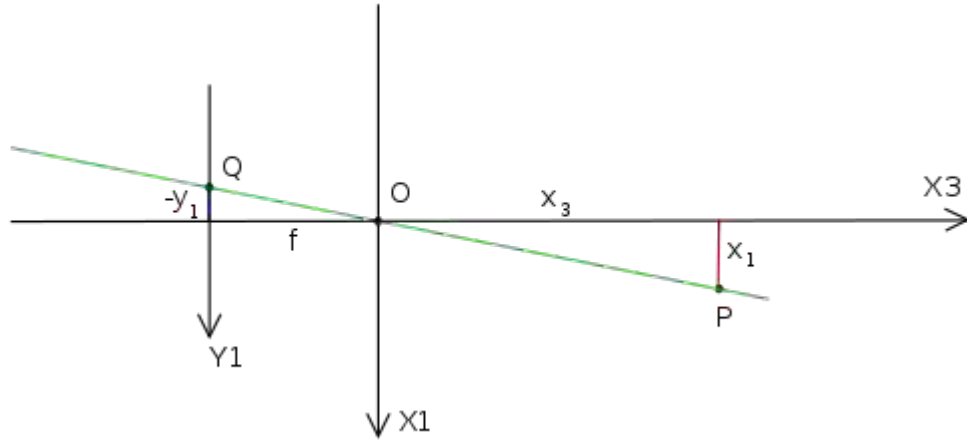
A point R at the intersection of the optical axis and the image plane. This point is referred to as the principal point or image center.

A point P somewhere in the world at coordinate  $(x_1, x_2, x_3)$  relative to the axes  $X_1, X_2, X_3$ .

The projection line of point P into the camera. This is the green line which passes through point P and the point O.

The projection of point P onto the image plane, denoted Q. This point is given by the intersection of the projection line (green) and the image plane. In any practical situation we can assume that  $x_3 > 0$  which means that the intersection point is well defined.

There is also a 2D coordinate system in the image plane, with origin at R and with axes  $Y_1$  and  $Y_2$  which are parallel to  $X_1$  and  $X_2$ , respectively. The coordinates of point Q relative to this coordinate system is  $(y_1, y_2)$ .



*Illustration 6: The pinhole camera model , viewed from the side ( from the X2 axis ) , illustration from Wikipedia , public domain*

The geometry of the pinhole camera viewed from the side , and on two dimensions. The calculations performed are based on similar triangles that are created with the point O as their intersection.

The mathematical equations that condense are the following :

$$\frac{-y_1}{f} = \frac{x_1}{x_3} \vee y_1 = -f \frac{x_1}{x_3}$$

$$\frac{-y_2}{f} = \frac{x_2}{x_3} \vee y_2 = -f \frac{x_2}{x_3}$$

$$(y_1, y_2) = \frac{-f}{x_3} (x_1, x_2)$$

These equations may seem elementary , and this is only logical since they have been a part of the human knowledge domain around for at least 2500 years ( they can be directly derived from the Pythagorean theorem circa 500BC ) but they are the first fundamental “laws” that govern the scene geometry when the 3D space is dissected to 2D planes.

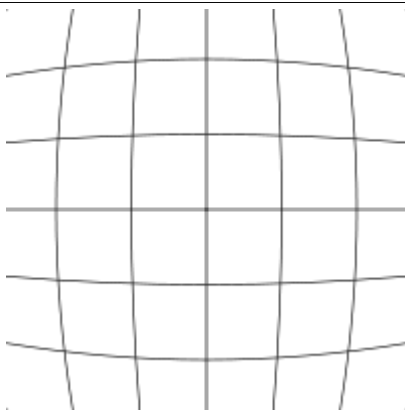
# Mathematical Framework

## 1.2 Camera Calibration

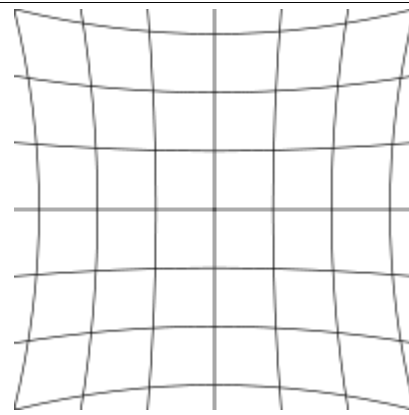
Having explained the underlying geometry behind the ideal pinhole camera model we need to adapt it to real cameras and their physical limits. In mathematics, it is possible to define a lens set that will introduce no distortions in the image captured. In practice, however, and due to manufacturing process inefficiencies two types of distortion occur . Radial distortion , caused by the shape of lens not being parabolic , and tangential distortion due to the assembly process of the camera in the factory.

Radial distortion causes a characteristic bending of straight lines as they get closer to the edges of the image and on systems that are heavily based on those images it can have a very detrimental effect on calculations that gets worse as the errors gradually accumulate in time. While disparity mapping algorithms can partly withstand this kind of distortion due to using a relatively large neighborhood of pixels that overall remains the same , point tracking and optical flow algorithms that estimate and track the camera position are very vulnerable to this kind of distortion. The reason this happens is because the relative positions of pixels change as they move to the edges and give wrong constraints for the system of equations to be solved later on.

These optical defects have been very notable during the analogue period of photography , when the photographs could not be dynamically altered to correct this distortion. Modern commercial camera makers often use low quality lens and mitigate the problem on the on-board image processing chip that reduces the problem. Trying to take into account the automatic approximate rectification inside the black box of the camera chipset makes the final “external absolute” rectification process even more difficult. Luckily due to the low cost of computer webcams the image captured by them is unaltered so we can study the distortion and then remove it.

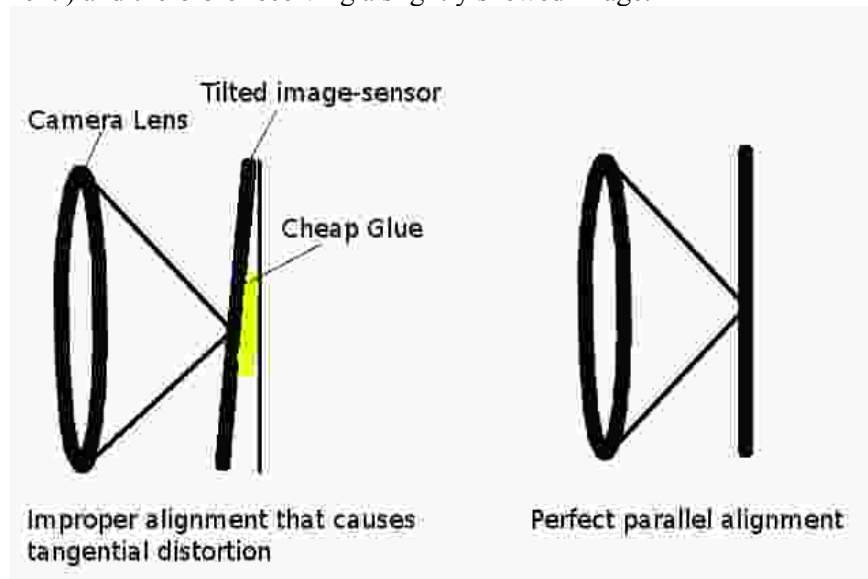


Barrel distortion

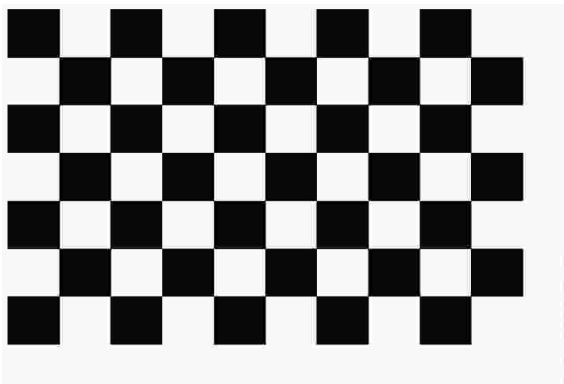


Pincushion distortion

Tangential Distortion on the other hand is a matter of misplacing the imaging sensor relatively to the lens ( not a fully parallel placement ) and therefore receiving a slightly skewed image.



Figuring out the way with which a camera distorts the projection of the world on to its image sensor is called camera calibration. There are numerous methods and considerations to be taken into account to achieve calibration , even methods that gradually “auto calibrate” the raw input images without special patterns and objects or prior training of the algorithm. [ citations needed ] .



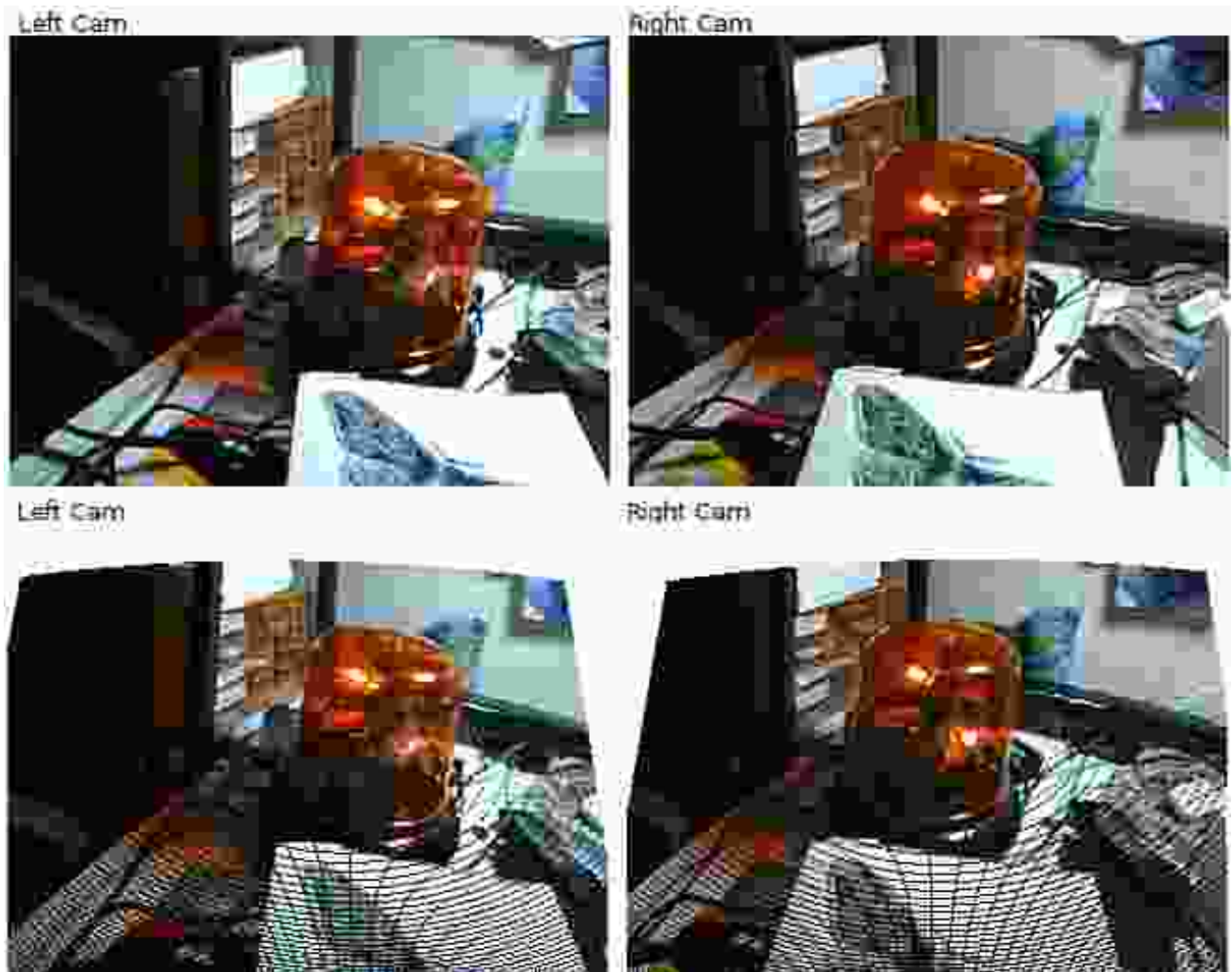
*Illustration 7: OpenCV Chessboard 10x7 calibration pattern*



*Illustration 8: Typical Detection Image Generated by OpenCV*

The OpenCV implementation receives the corners between the chessboard blocks as inputs , which are extracted using a corner detector. First, it computes the initial intrinsic parameters and sets the distortion coefficients to zero. Afterwards using the Levenberg-Marquardt optimization algorithm [citation needed] the reprojection error is minimized until a stable parameter set is found. The method was conceived by Zhang

[citation needed] and Sturm [citation needed] and a thing that is worth to be mentioned is that the cameras used by GuarddoG are graded by the manufacturer to have a less than 5% distortion and the algorithms work sufficiently well even when input is uncalibrated.



*Illustration 9: Raw images received from the cameras and their calibrated equivalent ( the distortion parameters are exaggerated to better show the way calibration alters the input images )*

# Mathematical Framework

## 1.3 Image Rectification

Each camera has intrinsic and extrinsic parameters. Intrinsic parameters model the camera as a device and they are constituted by the skew coefficient (  $\gamma$  ) that is a coefficient that is usually zero , the principle point or image center (  $C_x$  ,  $C_y$  ) and  $F_x$  ,  $F_y$  which is the focal point multiplied by a number that scales from pixels to distance ( and is defined by the size of a pixel in the image sensor ) .

Extrinsic parameters give information about the position of the camera in the world , and are basically a translation and a rotation matrix , usually combined in a 3x4 matrix.

The extended equations from the pinhole model for a perfect undistorted lens with with intrinsic and extrinsic parameters are modeled by the following equations

$$s \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$u = f_x x' + c_x$$

$$v = f_y y' + c_y$$

$x'$  and  $y'$  are used as an intermediate step to better show the added computations when performing resectioning in the page that follows

Radial and tangential distortion correction gets included to the model using  $k_1, k_2, k_3$  coefficients for radial distortion and  $p_1, p_2$  for tangential. They basically work by warping the image with a center of  $c_x, c_y$  and the higher the distance from the center the more it is pushed away.

$$s \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_1 \\ r_{31} & r_{32} & r_{33} & t_1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + t$$

$$x' = x/z$$

$$y' = y/z$$

$$r^2 = x'^2 + y'^2$$

$$x'' = x' (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2 p_1 x' y' + p_2 (r^2 + 2 x'^2)$$

$$y'' = y' (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1 (r^2 + 2 y'^2) + 2 p_2 x' y'$$

$$u = f_x x'' + c_x$$

$$v = f_y y'' + c_y$$

Executing these calculations gives us the rectified position of a point captured by the camera.

Since resectioning the image must be done for every frame received from the usb cameras ( which serve images @ 120 Hz ) and since most camera chips don't offer a hardware interface for passing the distortion parameters to the local integrated circuit so it can perform this kind of image processing with out involving the main processor , a fast technique must be applied to avoid calculating all these displacements on every new frame.

Since the camera doesn't change its focus settings , the distortion parameters are always the same. We can use this knowledge to our advantage generating a precalculation frame that has pointers to the calibrated positions as its elements after acquiring the distortion parameters.

That way , the expensive task of computing the formulas mentioned above happens only once and the least possible overhead is added to the pipeline process ( around 500 microseconds per frame on the main development computer , hardware details are on the second part of this text ). This tactic is followed both by the OpenCV implementation , as well as the GuarddoG RoboVision stack.



# Mathematical Framework

## 2.1 Image Processing

Digital cameras are devices that capture the light that the universe reflects on their sensor. The general problem most vision algorithms try to solve is guessing what kind of a world reflects the light in that way. The algorithms presented here are building blocks that gradually transform the raw RGB input into more computationally meaningful representations .

Convolution is a mathematical operation applied to sets of values that “redistributes” them according to coefficients from a second set of values. The result is a new combined set that has similarities with both previous sets. Convolution is originally defined in mathematical functional analysis and takes a slightly different form in image processing where it is typically performed on a 2D array of brightness values. The carrier of the weights is called a convolution matrix and its elements act as coefficients changing the neighboring elements of each pixel. The larger the convolution matrix size , the smoother the redistribution , but due to the computational cost the most common sizes for kernels are 3x3 or 5x5 with usually the middle pixel used as a point of reference or an anchor point.

The values transformed by the convolution matrix are the red , green and blue light intensities of the pixels retrieved from the image sensor. In the following example we assume a 3x3 kernel and a monochrome image sensor that captured 9x6 pixels. The kernel is passed left to right and up to down until all of the elements are changed. GuarddoG uses Blur , First and Second Derivative Convolution kernels that follow with example images.

1	1	1
1	1	1
1	1	1

**3X3 Convolution Kernel**  
**Divisor 9**

As the anchor of the kernel passes from each element of the image array the value ( marked blue ) gets replaced by the addition of the neighboring elements multiplied with the corresponding kernel coefficients.

$$H(x, y) = \sum_{i=0}^{M_i-1} \sum_{j=0}^{M_j-1} I(x+i-a_i, y+j-a_j) G(i, j)$$

The anchor element on the light intensities array will become

$$(1 \times 90 + 1 \times 80 + 1 \times 70 + 1 \times 90 + 1 \times 80 + 1 \times 70 + 1 \times 90 + 1 \times 80 + 1 \times 70) / 9 \text{ which is } 80$$

**9 x 6 Original Light Intensities Captured**

90	80	70	90	80	70	90	80	70
90	80	70	90	80	70	90	80	70
90	80	70	90	80	70	90	80	70
90	80	70	90	80	70	90	80	70
90	80	70	90	80	70	90	80	70
90	80	70	90	80	70	90	80	70

An important thing to be noted Is that values on the edges of the array ( marked orange ) can not be correctly calculated as not all neighboring elements exist , common solutions for this is zero padding , using a different divisor to compensate for the missing elements or skipping the elements that can not be calculated correctly .

**BLUR FILTER**  
(Gaussian Approximation)

1	2	1
2	4	2
1	2	1

Divisor 16



**FIRST-ORDER DERIVATIVE**  
(Horizontal Sobel )

1	2	1
0	0	0
-1	-2	-1

Divisor 1



**FIRST-ORDER DERIVATIVE**  
(Vertical Sobel )

-1	0	1
-2	0	2
-1	0	1

Divisor 1



**SECOND-ORDER DERIVATIVE**

-1	0	1
0	0	0
1	0	-1

Divisor 3



<chk>

As someone can easily observe by thinking a little about the convolution process , it is a waste of resources to perform multiplications with the null elements of a convolution matrix, and as an example for the second-order derivative that has 5 null elements a little more than half the original number of multiplications can be skipped. An additional optimization that can be performed is combining two convolution matrices in to one to reduce memory access related latencies from two subsequent passes on the image.

Horizontal			Vertical			Combined on		
1	2	1	-1	0	1	p1	p2	p3
0	<b>0</b>	0	-2	<b>0</b>	2	p4	<b>p5</b>	p6
-1	-2	-1	-1	0	1	p7	p8	p9
Divisor 1			Divisor 1					

The values p1 ... p9 are the pixel values on the image array in which the convolution takes place..  
In order to completely avoid multiplications ( at least on the matrix part ) we add and subtract the values and so for the pixel 2 ( p2 ) since the coefficient is 2 we do  $p2 + p2$  .

horizontal\_sum = p1 + p2 + p2 + p3 - p7 - p8 - p8 - p9

vertical\_sum = p1 + p4 + p4 + p7 - p3 - p6 - p6 - p9

final\_sum = square\_root( ( horizontal\_sum \* horizontal\_sum ) + ( vertical\_sum\*vertical\_sum ) )

The final speed up is replacing the square root operation with a log base 2 approximation using shift operations based on the IEEE 754 floating point arithmetic standards and the algorithm described below.

```
inline float square_root (const float x)
{
    union
    {
        int i;
        float x;
    } u;
    u.x = x;
    u.i = (1<<29) + (u.i >> 1) - (1<<22);
    return u.x;
}
```

Of course using an SIMD ( Single instruction, multiple data ) instruction set capable CPU with properly aligned data and loop unrolling can speed up the operations even more but even without these steps , the code form on this level is simple enough for gcc to do a good job optimizing it automatically.

Blur filters even out the colors on an input image using the median color value of the surrounding area . Blurring is a common operation by vision software mainly used due to the fact that image sensors retrieve pixels that suffer from noise , these noise spikes are reduced therefore leading to more stable edge and corner detection.

The First-order derivative operator acts as a differentiation operator , resulting in an output that only responds to “change” of colors and ignores similar colored areas. Thus it is very useful as it reduces the image to its more unique parts , its edges .

The second-order derivative operator also acts as a differentiation operator , resulting in an output that only responds to “change” of “change” of colors ( second order ) and ignores similar colored areas while also having a better reaction to sudden spikes on the color frequency. Its output also reveals the image edges but is much more stable than the first-order operator.

Palette reduction reduces the total number of possible tones that one pixel can take from 16581375 on an 24bit color depth (  $255 * 255 * 255$  ) to an other given number. Reducing the total possible colors causes similarly colored pixels to fall into the same color bin. This can be leveraged to make the datasets more resistant to noise. Conversion from a full color palette to a monochrome image , is a very common operation on computer vision algorithms.

Thresholding can be used as a filter extension to apply a high ( or low ) pass bound on an incoming signal and discard pixels that do not match the criteria. This is generally done after edge detection operations to reduce false output caused by noise.

The RGB Movement operation is a direct absolute subtraction of each of the pixels ( on each of the color channels ). This is passed through a low threshold and results on an output image with a large value where there is a large color difference (movement) and 0 value when the pixel remains unaltered This “delta” version of two images is useful in many occasions. First in determining if the stream of images is static , ( so we can skip redundant calculations and improve the performance and power consumption of the CPU ) , it is important when the robot is not moving and views a supposedly still environment as a really fast alarm function and it helps with disparity mapping , since unoc

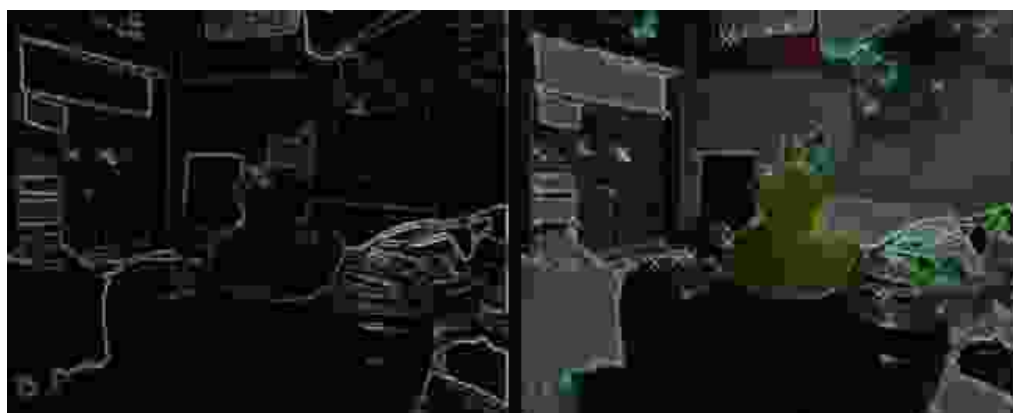
Histograms are produced by counting the total instances of the different colors on an area of an input. They can provide a good general idea about an image , such as its brightness, color distribution and are used in guarddog as a fast discarding mechanism for regions of the image when performing disparity mapping .

The miscellaneous image processing operations used by GuarddoG are mentioned in the table that follows  
</chk>

NAME	OPERATIONS	DESCRIPTION
Gaussian Blur	$9 * \text{Height} * \text{Width}$	Blurring input image to reduce noise
Sobel edge det.	$2 * 9 * \text{Height} * \text{Width}$	Edge Detection
Second-order de.	$4 * \text{Height} * \text{Width}$	Edge Detection
Palette Reduce	$\text{Height} * \text{Width}$	Group color frequencies together to reduce them
Threshold Image	$\text{Height} * \text{Width}$	Discard information that may be subject to noise
RGB Movement	$3 * \text{Height} * \text{Width}$	Subtract row RGB values from two consecutive images
Histogram	$\text{Height} * \text{Width}$	Calculate number of pixels that have the same color

## Mathematical Framework

### 2.2 Corner and Feature Detection

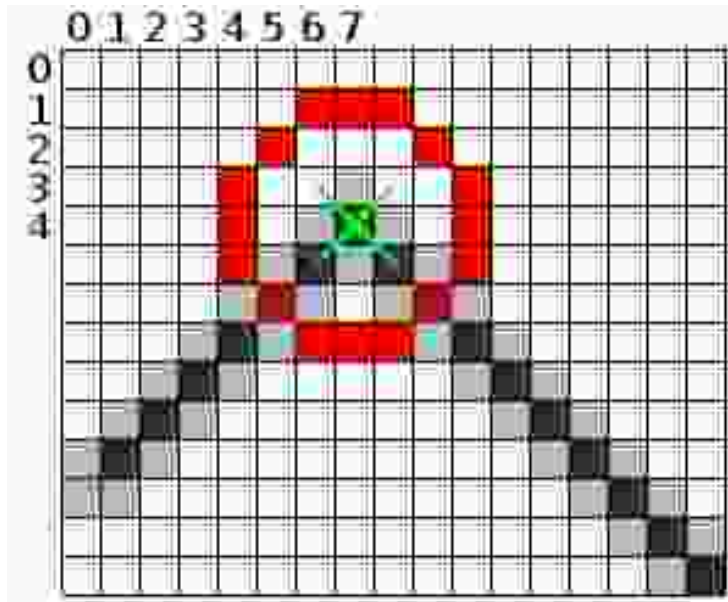


*Illustration 10: Left : An incoming image after passing through First-order Derivative Edge detection , Right : The corners detected , highlighted with green X marks*

After performing the various image processing steps mentioned above , to start moving away from the image as a raw array of color frequencies and into a better representation , we must focus on specific points on it that stand out and have unique characteristics . These points are called features or salient points and can be picked using a multitude of methods. The features used by GuarddoG are corners and offer a good performance and quality trade-off. They are both relatively inexpensive computationally to extract and also exhibit persistence between frames produced from small movement of the camera , in normal indoor lighting conditions.

<chk>

Some feature detectors such as SURF [citation needed] pick points that not necessarily lie in a corner , but nevertheless have a large eigen value and are scale and rotation resistant. The feature detector used by GuarddoG is built with high performance in mind ( and thus lower average quality of feature points ) and is called FAST [citation needed]. It classifies a point as a possible corner by casting a bresenham circle of radius 3 around it. Thus from the 16 points casted if the intensities o f at least 12 contiguous pixels are all above or all below the intensity of the central point by some threshold it returns a match. </chk>



*Illustration 11: An instance of the algorithm detecting a feature (corner) by sampling the 16 points of the circle casted around the point 7,4 using the FAST algorithm. The detector finds two similar colored points and succeeds in detecting the corner.*

<chk>A second feature detector that is used as a lower performance higher quality alternative and performs adequately is the OpenCV cvGoodFeaturesToTrack method by Shi and Tomasi [citation needed]. which utilizes a second-derivative filtered image. It then calls cvCornerMinEigenVal and cvCornerEigenValsAndVecs to pick the minimum eigen values under a threshold and again provides a list of good features on a reasonable computational cost. The inner workings of the algorithm are based on texturedness criteria that are reflected by the eigenvalues. Two small eigen values mean a roughly constant intensity profile within a window, a large and a small eigen value , a unidirectional texture pattern and two large eigen values patterns that can be tracked reliably such as corners. All these are extensively discussed in the original paper.

$$M = \begin{pmatrix} \sum (dI/dx)^2 & \sum (dI/dx * dI/dy) \\ \sum (dI/dx * dI/dy) & \sum (dI/dy)^2 \end{pmatrix}$$

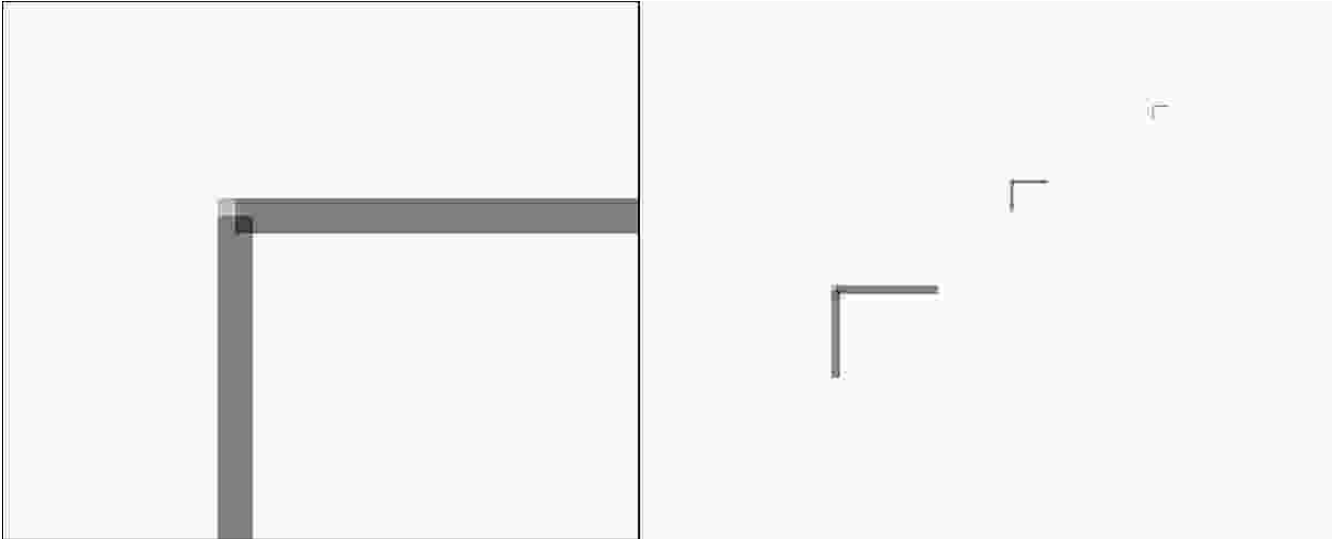
*The minimal eigenvalue is then picked  
since:*

*$x_1, y_1$  corresponds with  $\lambda_1$*

*$x_2, y_2$  corresponds with  $\lambda_2$*

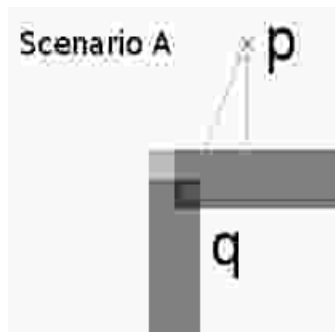
*and compared to a threshold*

A typical image retrieved from the camera consists of a finite number of pixels. The corners returned by the algorithms mentioned above are integers but in reality it is very improbable for a corner to lie exactly on the center of a pixel. This inaccuracy is enough to effectively derail the pose tracking algorithm that takes the corners as its only input and has a tendency to accumulate errors and thus we need more detail about where the corners truly lie. A detected pixel with coordinates (123,69) given as a result from the algorithms above may be fine tuned to a real number such as (123.349 , 69.512 ) for example.

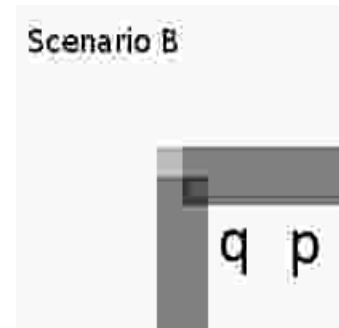


*Illustration 12: Left : In which pixel exactly does the corner lie ? Right : As the same corner image is viewed from increased distance ( or in a increased resolution ) the inaccuracy gets smaller compared to the total area covered*

To start approximating the new corner we have to build up a system of equations that when solved will give us a sub pixel approximation. The OpenCV method for this work is called `cvFindCornerSubPix` and it uses simple vector algebra to achieve it. It is based on the fact that the dot product of orthogonal vectors is zero and if one of the two vectors does not exist ( is zero ) it is again zero. This forms several equations that are all equal to zero which when solved provide a better set of coordinates for the corner.



*Illustration 13: A hypothetical point p and the two vectors that lead to it from point original corner point q*



*Illustration 14: A hypothetical point p on the same line with q*

$$\langle \nabla I(p), q - p \rangle = 0$$

The dot product of the Gradient of pixel p with  $q - p$  is in both cases zero

With a system of enough p points the point q is re positioned with better precision but the process can be repeated with as many iterations needed until an accepted accuracy is achieved. For example to achieve a tenth of a pixel accuracy , the process must be repeated until two subsequent q approximations differ less than 0.10 pixels .

\*</chk>

## Mathematical Framework

### 2.3 Template Matching and Integral Images

After image processing is finished producing “versions” of the data that reveal different aspects of the input images , the next technique performed by guarddog is called Template Matching.

There are numerous criteria that can be used to compare two image parts and decide if they match.

Guarddog uses a combination of pyramid segmentation , feature and template based matching across different templates to achieve high performance without sacrificing result quality. To this end the use of integral images speeds up and greatly improves the algorithm ( performance-wise ) .

The most simple and computationally efficient method for comparing two blocks of pixels is named SAD ( Sum of Absolute Differences ) and is basically the following equation.

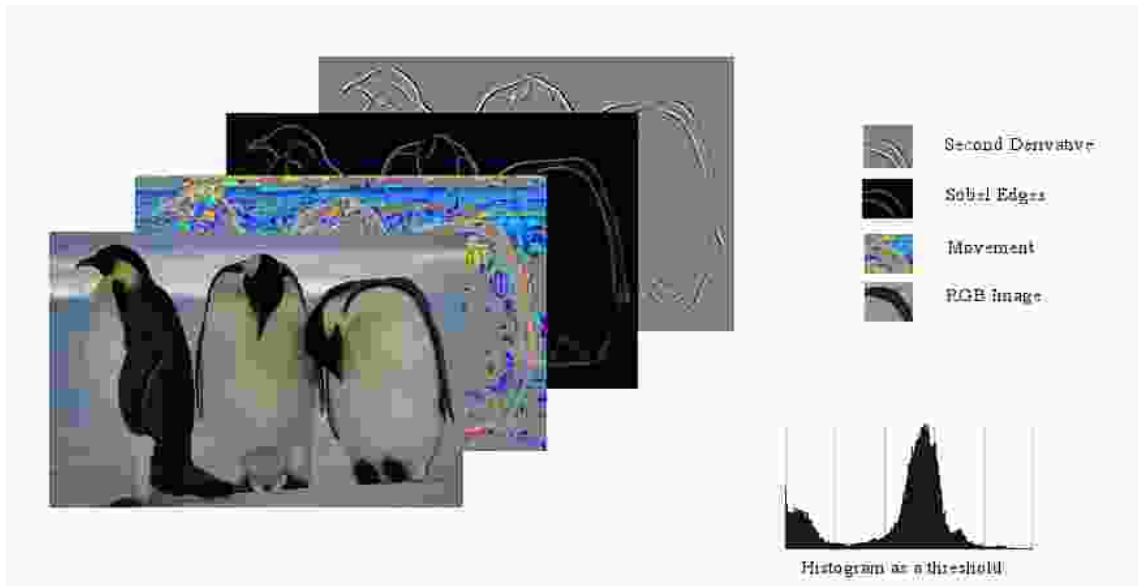
$$SAD = \sum_{x=0}^{width} \sum_{y=0}^{height} |(image1[x][y] - image2[x][y])|$$

This operation can be hardware accelerated on MMX and SSE2 instruction capable CPUs and thus is very lite weight. Although there are other metrics to find out if two image blocks match ( and how similar they are ) such as MSE ( Mean Squared Error ) , SATD ( Sum of absolute transformed differences ) , Normalized Cross Correlation ( NCC ) and other even more complex methods.

To make up for quality loss , while keeping the increased performance that SAD offers guarddog compares different “versions” of the patches that resulted mainly from convolution operations on the original data. That way the computational cost is moved from the block matching operation that can be performed millions of times ( especially in large images ) and does not take a guaranteed time to converting the image itself which has a fixed sized.

The different SAD results are then combined into a single value according to weights to compensate for the different range of values in each of the sub images. In order to further skip unneeded calculations a local histogram is used as a threshold that can completely avoid calculations if the 2 image blocks bear no resemblance at all ( i.e. one is completely white and the other completely black )





*Illustration 15: The things taken into account when comparing patches*

Before going into more detail about the template matching function , another useful representation for massive calculations on images is called integral images , or summed area tables.

$$I(x', y') = \sum_{x=0}^{x'} \sum_{y=0}^{y'} (image[x][y])$$

Once the table that every x,y has as an element the I(x,y) is created . We can skip a huge number of adding operations for an arbitrary area of the image ( limited only by the maximum value of an integer on the machine ). Any block addition operation is thus reduced to 4 operations.

$$\sum_{x=x1}^{x2} \sum_{y=y1}^{y2} image[x][y] = I(x1, y1) + I(x2, y2) - I(x2, y1) - I(x1, y2)$$

The resulting operation is not SAD because the subtraction does not produce an absolute difference on each pixel , the resulting operation is a plain Sum of Differences which is an even worse metric than SAD but it has such a big performance impact , that when used in conjunction with the sub images mentioned before it can make dense disparity mapping feasible , and when used in small enough areas provides good overall results.

Instead of :

$$|image1[x1][y1] - image2[x1][y1]| + |image1[x2][y1] - image2[x2][y1]| + \dots + |image1[xN][yN] - image2[xN][yN]|$$

we have

$$image1[x1][y1] + image1[x2][y1] + \dots + image1[xN][yN] - ( image2[x1][y1] + image2[x2][y1] + \dots + image2[xN][yN] )$$

which is the same with

$$image1[x1][y1] - image2[x1][y1] + image1[x2][y1] - image2[x2][y1] + \dots + image1[xN][yN] - image2[xN][yN]$$

\*

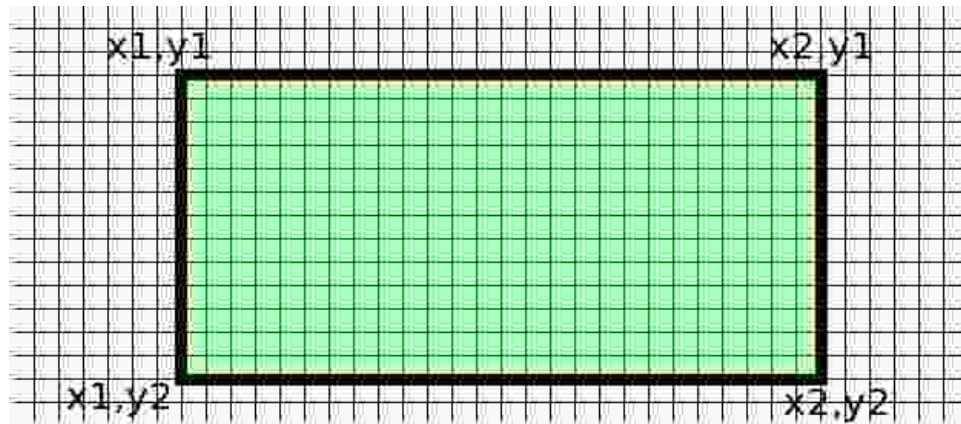


Illustration 16: Typically , we find the sum of the green area by adding all the pixels in it performing  $(x2-x1)*(y2-y1)$  operations

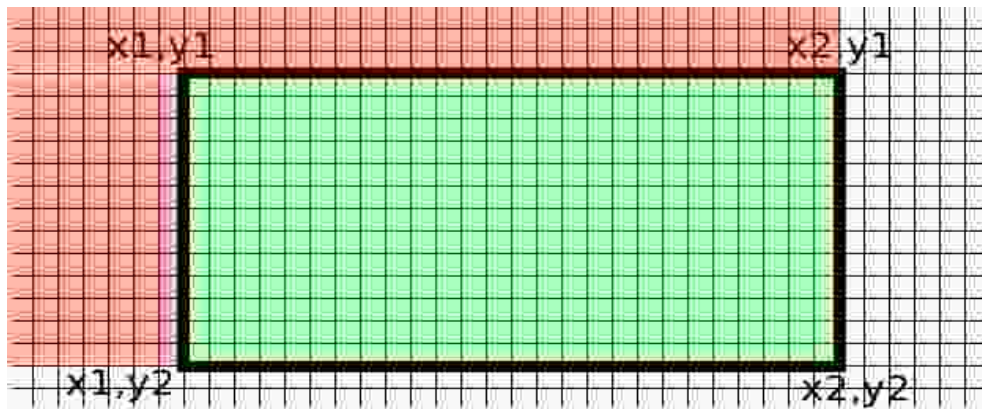


Illustration 17: We can find the sum of the green area by performing 4 operations ,  $I(x1,y1)+I(x2,y2)-I(x1,y2)-I(x2,y1)$  provided we have first calculated the integral array  $I$

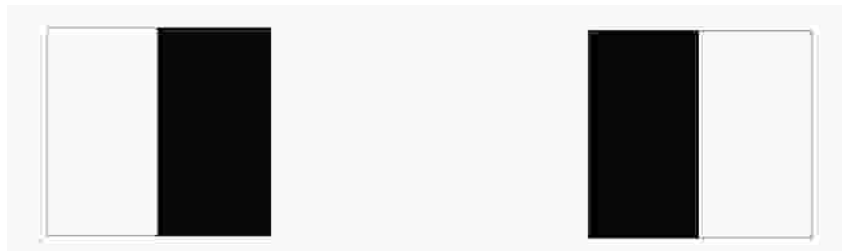


Illustration 18: A SAD metric returns total mismatch of these two blocks. An addition of differences metric ( not absolute ) such as the integral imaging technique described before returns a total match of the two image blocks

## Mathematical Framework

### 2.4 HAAR Wavelet based Face Detection

Haar-like features are digital image features used in object recognition. Their similarity with Haar wavelets is what gave them their name and they were used in the first real-time face detector. GuarddoG uses the OpenCV implementation of a haar cascade detector with an appropriate training file, while the implementation is largely based on the Viola Jones Face detection algorithm (Robust Real-time Object Detection) [citation needed].

There are many approaches to face detection and as a refinement recognition, including eigen faces (M. Turk and A. Pentland (1991). "Face recognition using eigenfaces"), image pyramids (Neural Network-Based Face Detection, 1998), and mixed methods (W. Kienzle, G. Bakir, M. Franz and B. Scholkopf: Face Detection - Efficient and Rank Deficient. In: Advances in Neural Information Processing Systems 17, pg. 673-680, 2005.), each of which have their own pros and cons.

The reason for choosing a Haar feature based face detection is that it is again accelerated by integral images and thus it can fit in nicely in the pipeline of the vision processor algorithm while performing incredibly well for upright faces that are the only kinds of faces that a small indoor robot should normally respond to.

A Haar Wavelet is a small region that consists of two areas, one black (low value) and one white (high value). As a pattern it can have a lot of iterations, and the ones displayed below are the most common ones. To decide if a feature is present, a simple sum operation is performed on each of the two areas and then the intensity difference is calculated between the white and black areas.

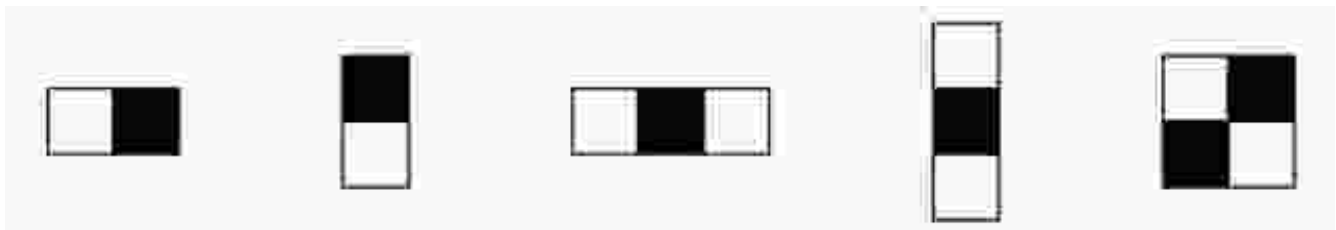
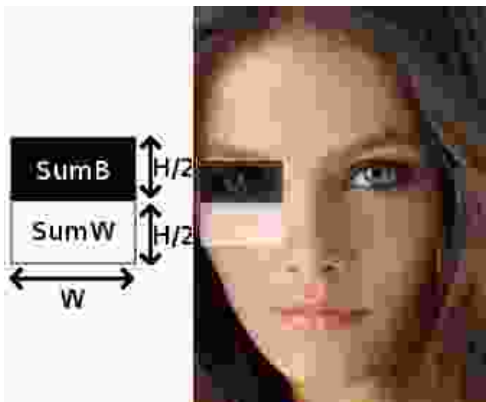


Illustration 19: Common Haar Wavelets



SumB = The Sum of color intensities in black area  
SumW = The Sum of color intensities in white area

$$\text{FeatureValue} = \text{SumW} - \text{SumB}$$

If ( FeatureValue > Threshold ) { FeatureValue=1 }  
else { FeatureValue=-1 }

Haar feature detection is a multi scale function basis and frequency is generally determined by its scale, not the direction. As many image bases, it forms a laplacian pyramid where its subscale is the subsampled low-resolution version (pre-filtered) of the signal plus a number of basis-projected versions of the signal for the high frequency components of that level. For instance the HWT of an image at a given (frequency) level produces a low freq image (LL, smoothed and subsampled) + a LH, a HL and a HH component corresponding to the 1st, 2nd and 5th pattern describe in the illustration 19. That way the image is transformed to an array of response numbers to these simple patterns and using a correct cascade of haar wavelets appropriate to the size, and orientation of detection this can be used as a tool for generic object detection ( Papageorgiou, Oren and Poggio, "A general framework for object detection" ) [citation needed]

The Viola and Jones detector basically works using this framework to discard portions of the image as “non-faces”. To construct an optimal haar cascade the classifier is trained with two image sets, one with faces ( for face detection usage ) and one with non-faces and an adaptive boosting machine learning algorithm, popularly coined as AdaBoost [citation needed] picks the best features that will drive the face detector. A sample detection image is the following, using the OpenCV cvHaarDetectObjects implementation and the haarcascade\_frontalface\_alt.xml cascade. The good response rate of this method was also confirmed by realtime usage on the International Fair of Thessaloniki 2011 where GuarddoG collected over 4500 faces on a course of a week with a very low false detection rate.



*Illustration 20: Left : Sample face detected ( marked by purple circle ), features detected by the corner detector explained at topic 1.2.1 ( marked with yellow dots ). Right : a possible HAAR cascade manually created for dramatization of the way HAAR Cascades digitize images and thus serve well for two dimensional face and object detection.*



*Illustration 21: Random faces out of a 4500+ faces collection gathered during IFT 2011*

\*after

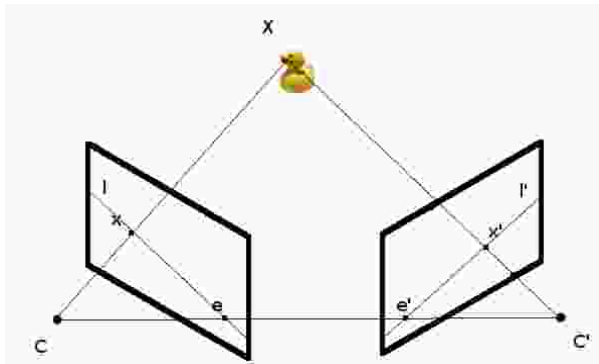
## Mathematical Framework

### 3.1 Epipolar Geometry

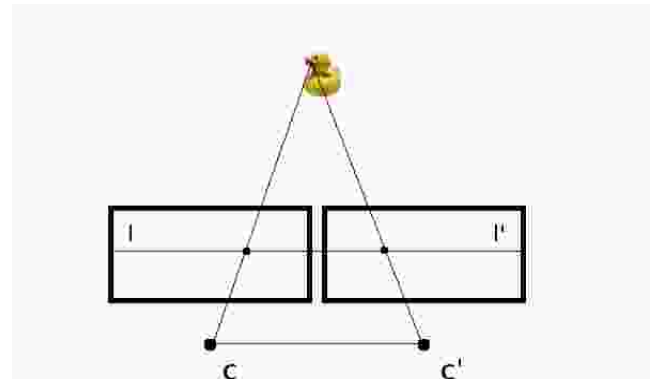
Assuming a rectified input of two pinhole cameras with a known alignment, viewing a 3D scene, there are some geometric relations about the projections of 3D points among them.

Both cameras see the world from a different viewpoint, and while the projected image is different there are some geometric constraints that can be leveraged to extract depth data using disparity mapping, a process which will be analyzed later.

For reasons of efficiency this project uses cameras positioned in parallel so the epipolar plane forms a parallel line from frame to frame. This configuration is used to reduce errors caused by incorrect calibration and reduce the overall complexity of the algorithms that are based on matching parts from one image to the other.



*Illustration 23: A non parallel alignment where we can see highlighted the camera centers  $C$  and  $C'$ , the baseline that goes through both of them, the epipoles  $e$  and  $e'$  which are the intersections of the image planes with the base line, the projection of the point  $X$  at  $x$  and  $x'$  when connected to the epipoles gives us the epipolar lines  $l$  and  $l'$*

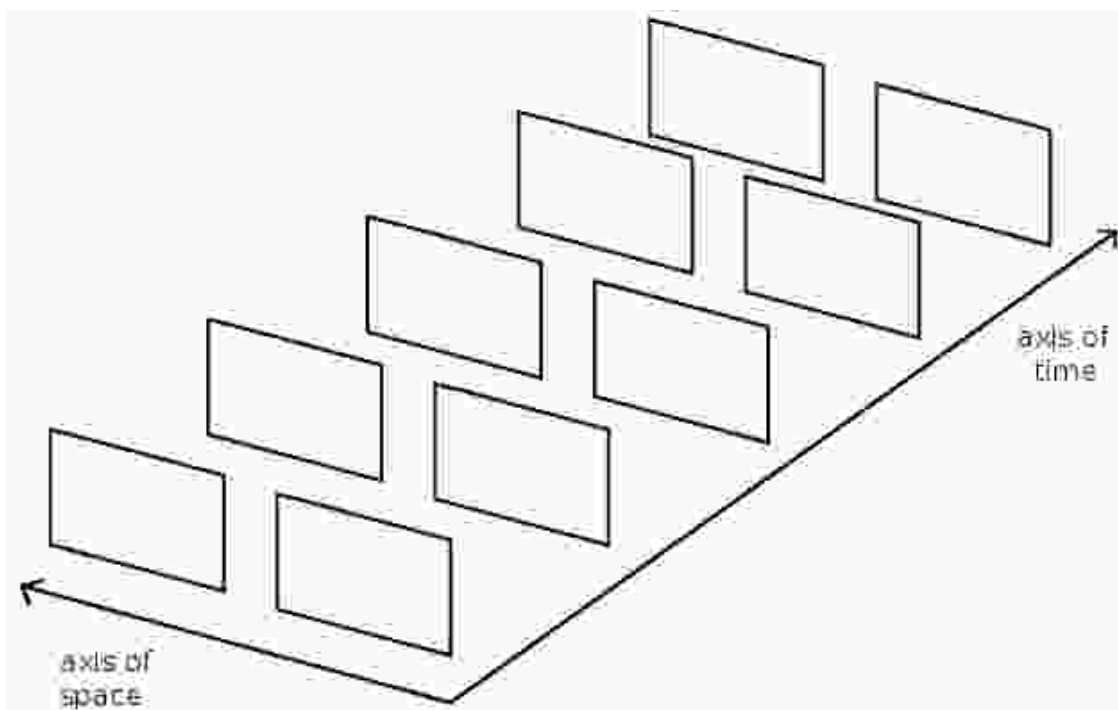


*Illustration 22: The parallel alignment used by GuarddoG, all the epipolar lines are parallel. The base line between  $C$  and  $C'$  does not intersect with the image planes.*

With the parallel setup the two projection images are essentially being produced by a translation of the camera center parallel to the image plane. This results in the points  $e$  and  $e'$  being in infinity, and the baseline never touching the image plane (since it is parallel to it)

Since the projection of all the points on the line from  $C$  to  $X$  and  $C'$  to  $X$  lay on the  $l$  and  $l'$  epipolar lines, to find out the projection of the ducks head on the right image we can reduce our search area in the same height coordinates from image to image and that makes disparity mapping practical for computation.

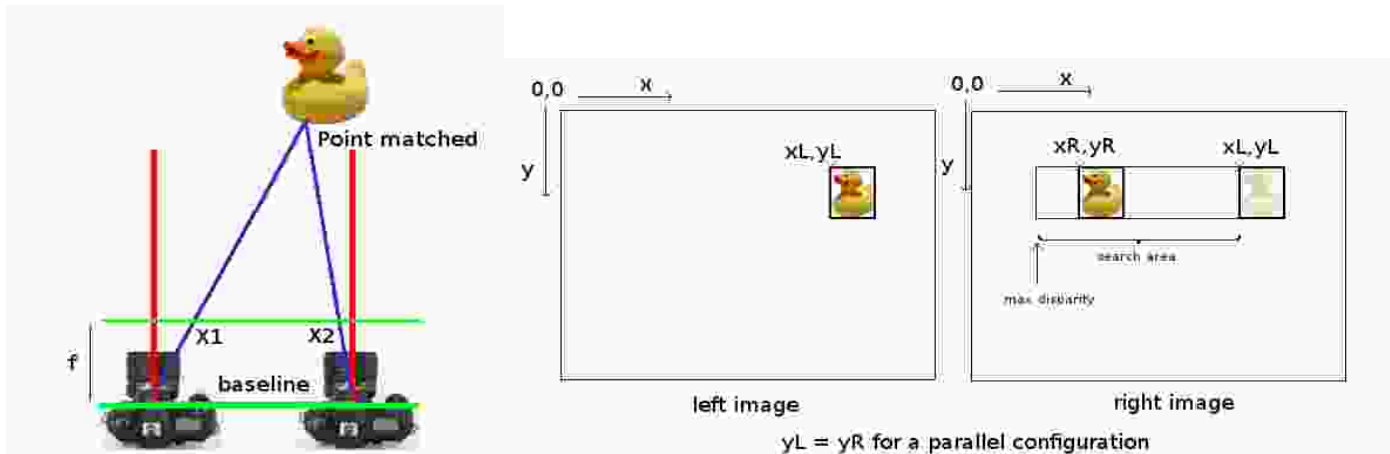
From a stream of frames (in the axis of time and not space) observed as the robot moves, since we have lots of different types of movements and combinations of rotations and translations, epipolar geometry is once again a useful concept for the calculation of the fundamental matrix between two frames. The reason for this is because it provides the fundamental matrix equation constraints. GuarddoG though uses homographies and not the fundamental matrix for camera pose estimation since the 3d points have a much larger overhead since they have to be extracted through disparity mapping and typically have a higher error rate and lower coverage than the corners that are used for the homographies.



*Illustration 24: The stream of incoming images spans both in time and space axis. Epipolar geometry can help us move in both directions provided that the scene we view as we move remains static.*

## Mathematical Framework

### 3.2 Binocular Disparity Depth Mapping on a parallel camera setup



*Illustration 25: Disparity mapping , geometry overview*

Binocular disparity depth mapping is a procedure that uses two image sources as input and produces an output containing depth information about the scene viewed. This is achieved by matching small parts from the left image to the right one and vice versa and calculating the difference of the image region projections. Due to the complex ill-posed nature of 3D scenes , occlusions , specular lighting highlights and frequent low texture areas , it is a difficult task especially when computing a dense depth map , since there is a very large area to search for every 3D voxel of output.

Since the disparity mapping algorithm developed as a part of the project differentiates from traditional block matching algorithms since it uses summed area tables , this topic will diverge from the others since it contains detailed comparisons between other algorithms. GuarddoG uses a parallel binocular camera setup on rectified images which simplifies the procedure since , as discussed in the previous topic , epipolar lines are collinear and parallel. This reduces the vagueness of the search domain and also reduces the total number of worst-case operations , which as seen in the overview are in the worst case 24,576,000 comparison operations ( using a 10x10 window this means 2,457,600,000 pixel operations ) for two 320x240 images. For performance reasons the resolution of the two images is also 320x240 due to the target low-end CPU

Typical disparity mapping algorithms use a metric such as SAD , SSD , MSE and others as mentioned in the Template Matching topic of this document. There are many algorithms for disparity mapping which use different approaches and ideas on the subject. A list of related disparity mapping algorithm can be found in the website of the Middlebury benchmark for stereo vision ( [vision.middlebury.edu/stereo/eval](http://vision.middlebury.edu/stereo/eval) ) , which is a list of algorithms compared on the same rectified image datasets. A very informative taxonomy of dense disparity mapping algorithms , also published by the Middlebury College [citation needed ] is an invaluable source that compares both the methods and quality metrics for each of the methods. The GuarddoG algorithm fares relatively well when taking into account the simple principles of its operation , especially for low quality settings which use a large quantizer reducing the depthmap resolution .

GuarddoG does not rely on very detailed depth maps since pose tracking happens using 2D points on the image projections , and depth maps are mainly used for collision avoidance tasks.

The classic approaches on dense disparity mapping procedures use the model on illustration 25[citation needed ] and can be grouped in 3 steps.

- 1 – Preprocessing the image to make it suitable for the nature of operations on step 2
- 2 – Performing the comparison operations from one image to the other and storing the results on a depth buffer
- 3 – Refining the output depth buffer using some smoothness constraints

Comparisons ( step 2 ) are typically distinguished by their matching method ( SAD , SSD , absolute difference etc.) and optimization function ( graph-cut , dynamic programming , winner takes it all , simulated annealing , phase matching etc . ) . The Middlebury College taxonomy paper [citation needed ] again provides a good and contemporary resource for sorting out the different algorithms.

In GuarddoG the approach followed , described in general terms is to focus on preparing many representations of the data on the preprocessing step , and then use raw subtraction on them ( Sum of Differences with the help of summed area tables ) with a window aggregation on a pyramid of different levels and a winner takes all optimization function.

The algorithm is compared with the libELAS and Hirschmuller disparity mapping algorithms which are briefly explained in the following paragraphs.

The first step , preprocessing , is typically the fastest part of the procedure , since it does not involve iterations on the image. Converting an image to its sobel derivative for example requires  $320 \times 240 \times 6 = 460,800$  operations ( much less than the 2,457,600,000 operations worst case for step 2 , with an even larger impact on real CPU time , due to less data locality overheads ) .

Guarddog uses the following image representations :

→ Second derivative → Summed Area Table Representation

RGB Image → Gaussian Blur → Sobel Edge → Summed Area Table Representation

RGB (Movement ) Difference With last RGB Frame → Summed Area Table Representation

The RGB Movement difference metric is also one of the areas of the GuarddoG algorithm that makes it better suited for disparity mapping on a stream of successive moving images since the moving edges act as a coefficient that helps matching quality , and thus still disparity maps ( such as the ones on the Middlebury benchmark ) provide a worst result than real operation moving imagery. This is also the reason for choosing the specific camera controllers ( analyzed extensively in the hardware camera sensors topic ) since their 120 fps input and fast shutter enables “clear” edges that stand out on movement ,even in moderate movement scenarios.

The second step involves performing a very large number of comparisons between areas on the left image and areas on the right one to find a pair that is the closes match and gives the true disparity value for each of the pixels. Methods such as libELAS [citation needed] use robust support points which are used as a basis for neighboring points and interpolation is performed on triangular areas for pixels between them thus reducing the time needed , instead of an exhaustive search through the whole image. This has a more dramatic performance impact on large resolution images , where there is also more information available for increased disparity resolution and thus the low resolution benchmark that follows doesn't do justice to the algorithm , but it is a good indicator of its performance.

The next method compared with the GuarddoG disparity mapping algorithm is the work by Hirschmuller [citation needed] implemented in the StereoSGBM method of OpenCV , ( Semi Global Matching ). Its results are impressive both for their accuracy and their speed and as an algorithm it solves the disparity mapping algorithm by trying to minimize an energy function using mutual information [citation needed].



GuarddoG uses a traditional disparity approach which calculates all the possible window matches and compares their score keeping the best ( winner takes it all ).

The novelty of the algorithm is that it uses integral images and comparing a combination of histogram , sum of differences on sobel , sum of difference in movement , sum of difference in second derivative and sum of difference on rgb values metric , each of which is performed with 4 operations instead of a NxN for a window of size N. Although this idea and work done on this disparity mapping algorithm originates by own experiments in 2007 it still remains useful today even compared to state of the art disparity mapping algorithms targeted for real time operation. Integral images and sums of raw differences could also be used on many of the other algorithms that use a different approach for a cumulative improvement of performance in addition to their own speed ups.

The third step , post processing typically re scans the output and normalizes it removing outliers and smoothing it with a gaussian or other function. Empty areas can be filled with neighboring depth values and iterative algorithms can pass the output to the second step again until convergence to a stable result or a timeout occurs. GuarddoG has a simple gap filling algorithm as a post processing filter but it is typically not activated since without outlier filtering it can help propagate noise and degrade the precision of the depth map.

#### **GuarddoG ( traditional ) disparity mapping algorithm pseudocode**

```
xL_Limit = height
yL_Limit = width
x_step = matching_window_width / detail
y_step = matching_window_height / detail

while ( yL < yL_Limit )
{
    xL = 48; // Starting point , typically 15% of the image size therefore 48
    for a 320x240 image
        while ( xL < xL_Limit )
        {
            best_match = Infinity;
            if ( //Filtering low texture areas to reduce errors
                EdgesOnInputWindow(
                    xL,yL,
                    matching_window_width,
                    matching_window_height
                ) > edges_required_to_process_threshold)
            {
                MatchWithHorizontalScanline (
                    xL,yL
                    matching_window_size_x,matching_window_size_y
                    &best_match,
                    &xR,&yR
                )

                if ( best_match != Infinity )
                {
                    /* WE FOUND A MATCH */
                    RegisterDisparity(xL,yL,xR,yR,window_width,window_height)
                } else
                { /* AREA IS EMPTY :P */ }
            }
            xL+=x_step
        }
        yL+=y_step
    }
```

```

MatchWithHorizontalScanline
(
    xL,yL
    matching_window_size_x,matching_window_size_y
    &best_match,
    &xR,&yR
)
{
    xR_Limit=xL
    yR_Limit=yL // this can be an offset used for bad calibration situations
    best_score=Infinity
    while ( yR <= yR_Limit )
    {
        if (xR_Limit>MaxDisparity ) { xR = xR_Limit-MaxDisparity; } else
        { xR = 0; }
        while ( xR < xR_Limit )
        {
            score = ComparePatches
                (xL,yL
                xR,yR
                window_width,
                window_height
                ); // This function uses integral images to extract a score
                // and this is the speed up of the guarddog algorithm
            if ( best_score < score )
            { //New best result
                best_match=abs(xL-xR)
            }
            ++xR
        }
        ++yR
    }
}

```



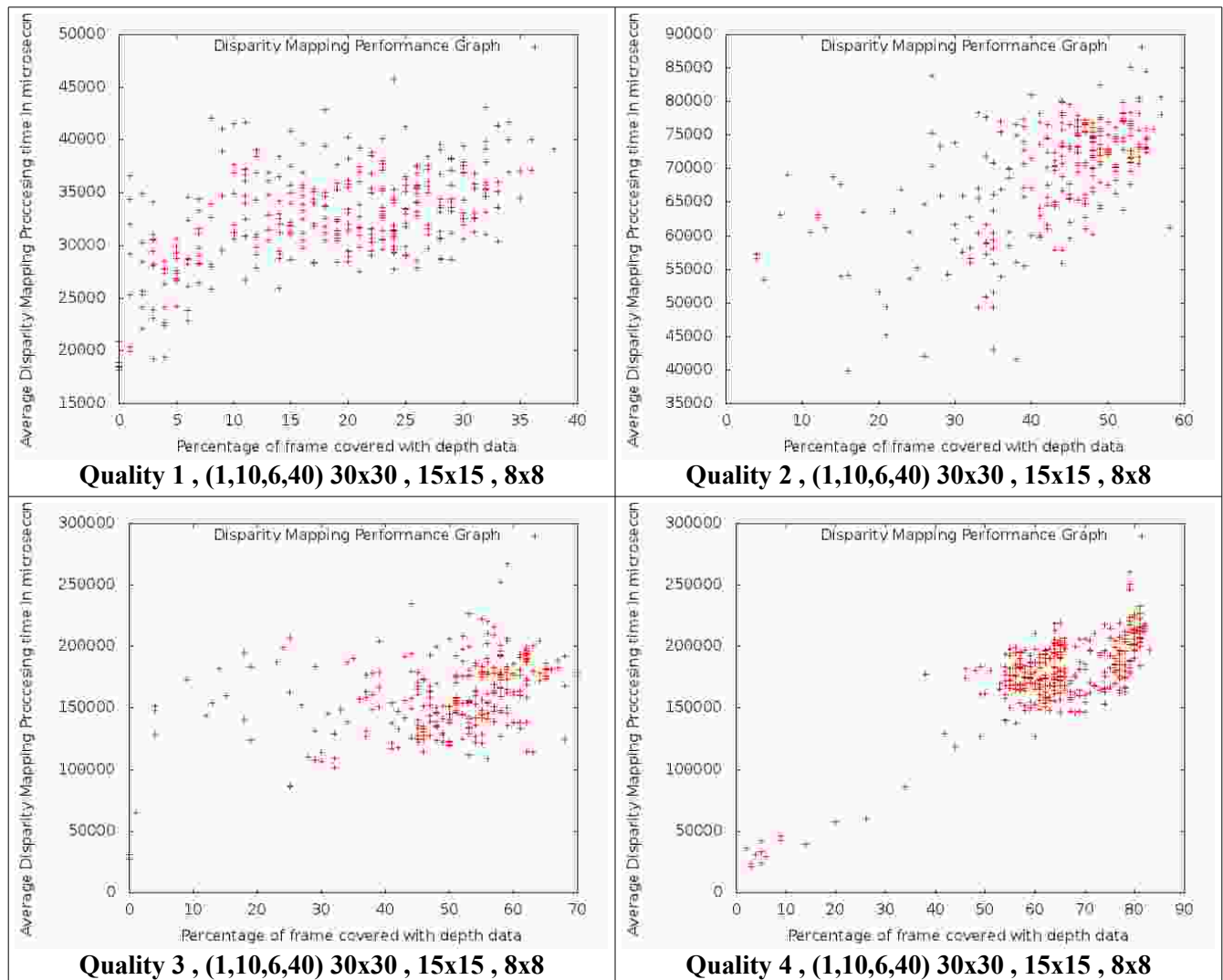
*Illustration 26: Disparity Mapping on the GUI of GuarddoG*

\*data sets after here

The following is a graph of covered area with depth information ( percent ) vs processing time

for quality 1 we have values between 15,000 – 50,000 microseconds for coverage 0-35%  
for quality 2 we have values between 35,000 – 90,000 microseconds for coverage 10-60%  
for quality 3 we have values between 90,000 – 300,000 microseconds for coverage 10-70%  
for quality 4 we have values between 40,000 – 300,000 microseconds for coverage 10-80%

The maximum coverage possible is 85% due to the initial value of xL , ( xL = 48; as seen on the pseudocode )



In the extensive comparisons that follow show the results for the different quality quantizers of the GuarddoG disparity mapping on the Tsukuba stereo set., and after that a comparison between the ground truth , guarddog , libElas and Hirschmuller algorithms follows for quality setting 4 ( since lower settings have worse output ) and 320x240 size input images

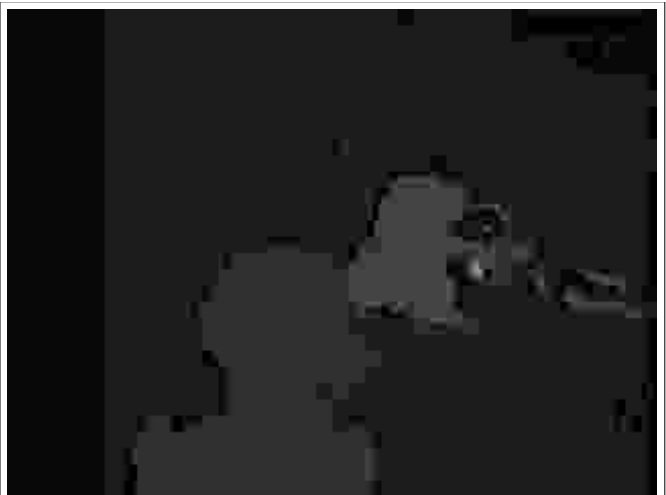
In all the GuarddoG examples mentioned here there are 3 passes with 30x30 , 15x15 , 8x8 windows and the coefficients for each of the blocks in the comparison function is 1xRGB difference , 10xMotion difference , 6xSobel difference , 40x Second-derivative difference

\* data sets after here

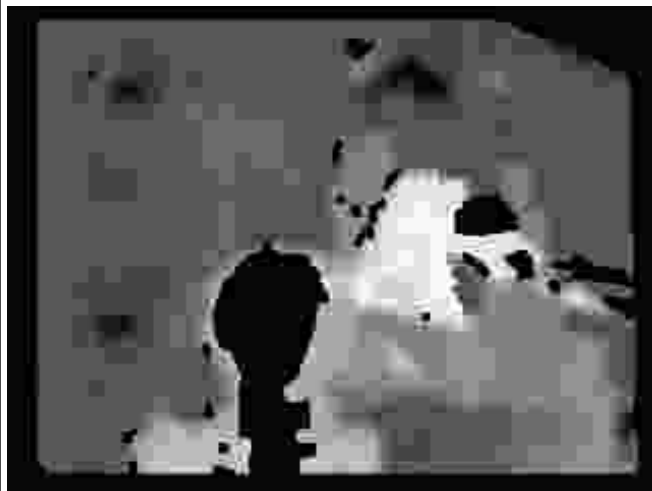
## Tsukuba Test Image Extensive Comparison



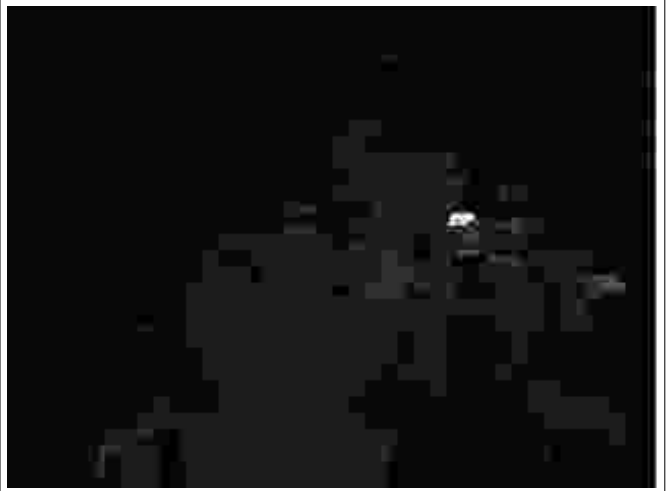
**GROUND TRUTH**



**OpenCV StereoSGBM Hirschmuller 34 ms**



**libELAS 52 ms**



**GuarddoG ( quality setting 2 ) 66 ms**



**GuarddoG ( quality setting 3 ) 118 ms**



**GuarddoG ( quality setting 4 ) 290 ms**

## Tsukuba Test Image Extensive Comparison



**LineSeg 1300+ ms**



**Segmentation Based 2000 ms**



**Variable Windows 26000 ms**






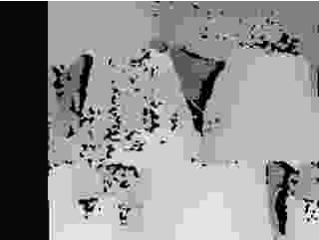

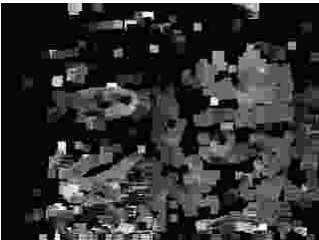

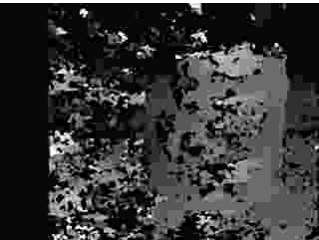



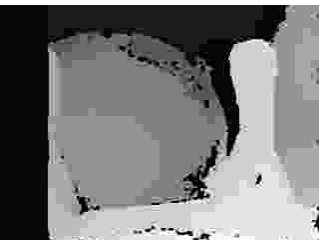


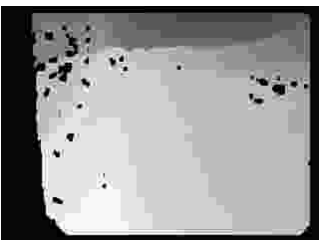
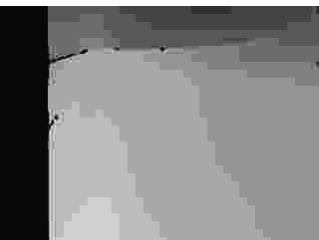







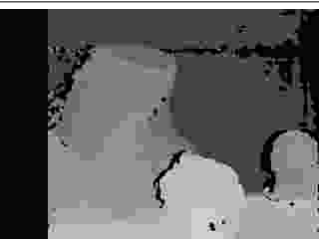
**Fast Bilateral 32000ms**




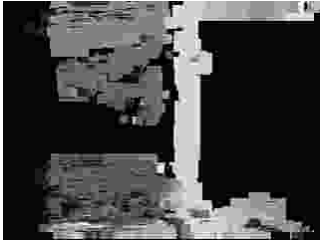
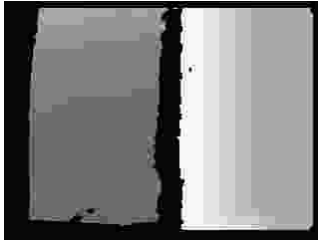


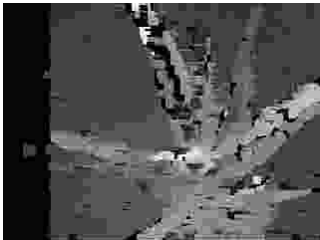

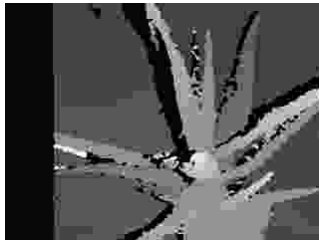


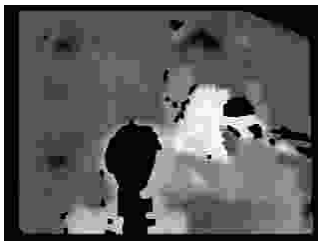
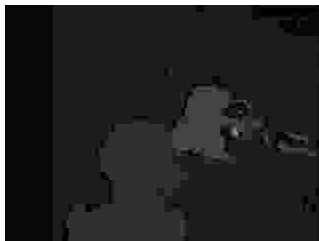


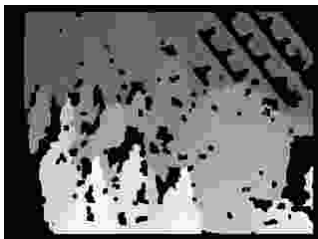



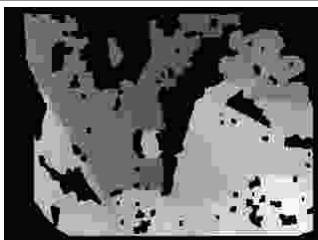
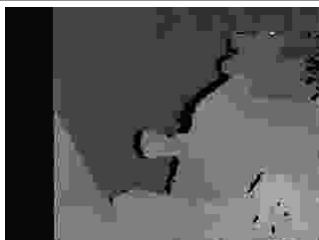
**Adaptive Weights 1221000 ms**



**Segment Support 2358000**

Original Image	GuarddoG	libELAS	OpenCV StereoSGBM
			
flowerpots	142 ms	51 ms	38 ms
			
gddg ( custom )	249 ms	28 ms	36 ms
			
bowling	173 ms	48 ms	40 ms
			
cloth	433 ms	51 ms	31 ms
			
lampshade	147 ms	39 ms	36 ms
			
middleburry	171 ms	27 ms	37 ms

\*

Original Image	GuarddoG	libELAS	OpenCV StereoSGBM
 wood	 181 ms	 40 ms	 37 ms
 aloe	 346 ms	 52 ms	 38 ms
 tsukuba	 205 ms	 41 ms	 35 ms
 cones	 251 ms	 52 ms	 32 ms
 teddy	 200 ms	 40 ms	 33 ms

# Mathematical Framework

## 4.1 Homography estimation

Given two sets of two dimensional points and the correspondence between them , a problem that arises is calculating the transformation that took place to lead from the first set of points to the other. This is called a homography and being able to find a close approximation of it is a tool that can be used to allow the camera position to be tracked , utilizing purely visual means.

Supposing we have the points :  $p_1 ( x_1 , y_1 , 1 ) , p_2 ( x_2 , y_2 , 1 ) \dots p_n ( x_n , y_n , 1 )$  which correspond to the points  $p'_1 ( x'_1 , y'_1 , 1 ) , p'_2 ( x'_2 , y'_2 , 1 ) \dots p'_n ( x'_n , y'_n , 1 )$

We want to find a 3x3 matrix H so that  $p'_i = H p_i$  for every i from 1 to n

$$\begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} = H \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

$$\begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

*performing the multiplication*

$$\begin{bmatrix} x'_i \\ y'_i \\ z'_i \end{bmatrix} = \begin{bmatrix} h_{11}x_i + h_{12}y_i + h_{13}z_i \\ h_{21}x_i + h_{22}y_i + h_{23}z_i \\ h_{31}x_i + h_{32}y_i + h_{33}z_i \end{bmatrix}$$

*for inhomogenous coordinates*

$$\begin{bmatrix} x'_i/z'_i \\ y'_i/z'_i \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{(h_{11}x_i + h_{12}y_i + h_{13}z_i)}{(h_{31}x_i + h_{32}y_i + h_{33}z_i)} \\ \frac{(h_{21}x_i + h_{22}y_i + h_{23}z_i)}{(h_{31}x_i + h_{32}y_i + h_{33}z_i)} \\ 1 \end{bmatrix}$$



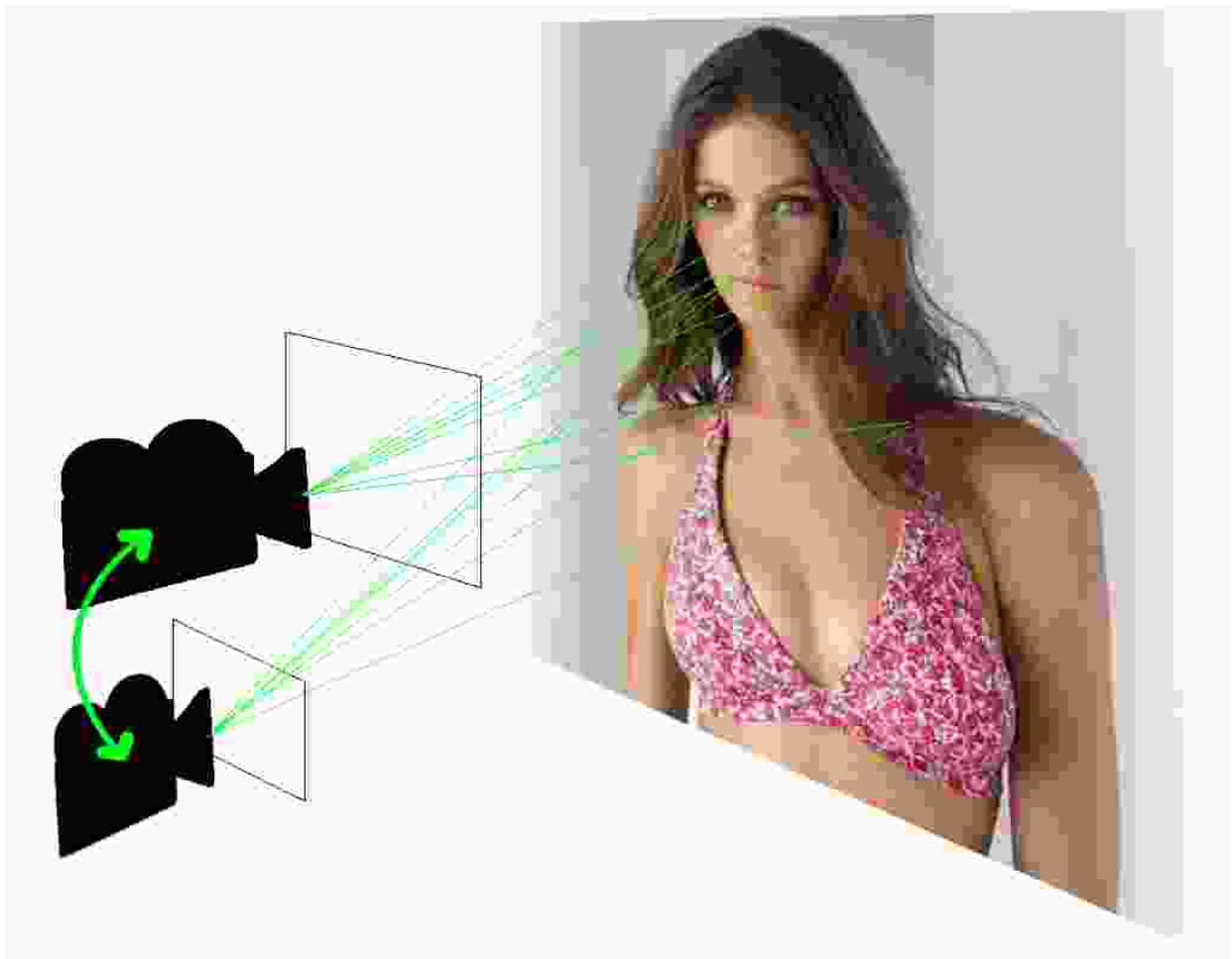
Provided we have enough ( correct ) point correspondences we can form enough equations to find the values of  $h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}, h_{33}$  .but due to errors , not only caused by feature detection , but also by the matching procedure even when using subpixel accuracy points that have a high percentage of correct matches the usual case is that the equations cannot be solved as they are incompatible and there is no possible H matrix that can satisfy them.

The solution to the problem is to start picking pairs and then compare their squared differences

$$\sum \left( x'_i - \frac{(h_{11}x_i + h_{12}y_i + h_{13}z_i)}{(h_{31}x_i + h_{32}y_i + h_{33}z_i)} \right)^2 + \left( y'_i - \frac{(h_{21}x_i + h_{22}y_i + h_{23}z_i)}{(h_{31}x_i + h_{32}y_i + h_{33}z_i)} \right)^2$$

Gradually using a point picking algorithm such as RANSAC ( the next theory issue examined ) that due to its design can be resistant to outlier matches , an adequatel approximation can be achieved .

The OpenCV methods for finding a homography , provided we have first extracted two sets of points and matched them is called `cvFindHomography` and it can use the RANSAC , a least median or a raw method using all of the available points. Due to the importance of pose tracking for the camera of the robot , and despite of the stochastic nature of the algorithm the RANSAC option is chosen by guarddog to compensate for the medium quality of features points and their matches.

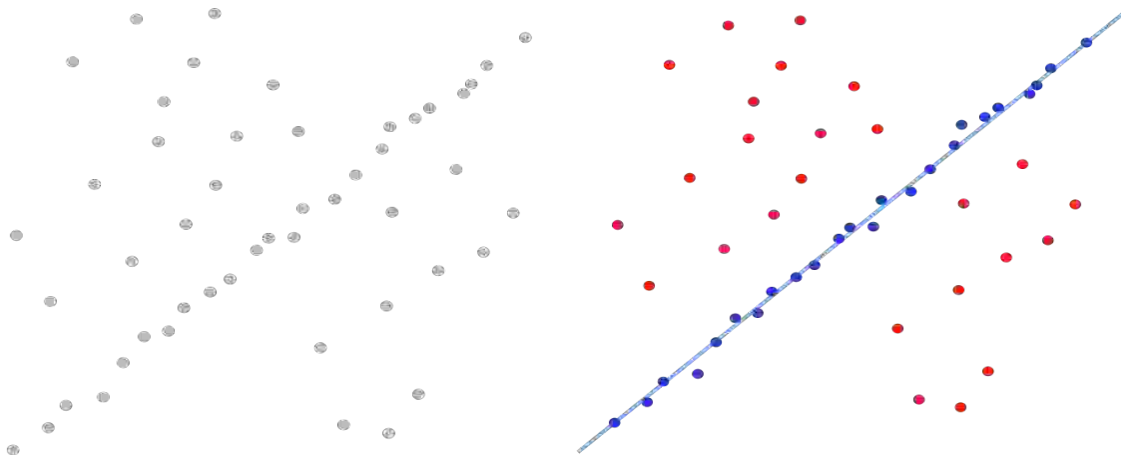


*Illustration 27: In a picture and a few words , a homography finds out the transformation that took place between two views of a scene from two matched sets of 2D points*

# Mathematical Framework

## 4.2 RANSAC

RANSAC or RANdom SAmple Consensus is an algorithm that is designed to pick elements from a dataset in a way that maximizes a desired metric. It was first published in 1981 [ citation needed ] and differs from other algorithms that perform similar tasks because it filters out outliers as part of its process and for a high enough probability of a dataset element being an inlier and a matching configuration it returns a result unaffected and undistorted by the outliers.



*Illustration 28: Left : A collection of points that form a line with a high number of incorrect measurements , Right : RANSAC given criteria to match points along a line can successfully reject outliers and recover the line , Images from Wikipedia , public domain*

The algorithm has a model that grades the points using a heuristic and iteratively picks small subsets of the data and keeping track of the error rate of a particular subset. Each time a large enough subset fits the model better than all previous ones this is recorded and kept as the new top standard which all feature subsets try to improve. The obvious downside of this algorithm is that it has a very high complexity upper bound for the procedure since it is stochastic ( non-deterministic ). To improve its performance it can be fitted with timeout counters that will return after a given time with the best result calculated at the time or it can return the best value when it is satisfactory compared to the maximum acceptable error threshold.

A high-level algorithm is given in the next page which gives a clear view of the inner workings of RANSAC.

### RANSAC Algorithm pseudocode

input:

- data - a set of observations
- model - a model that can be fitted to data
- n - the minimum number of data required to fit the model
- k - the number of iterations performed by the algorithm
- t - a threshold value for determining when a datum fits a model
- d - the number of close data values required to assert that a model fits well

to data

output:

- best\_model - model parameters which best fit the data (or nil if no good model is found)
- best\_consensus\_set - data points from which this model has been estimated
- best\_error - the error of this model relative to the data

iterations = 0

best\_model = 0

best\_consensus\_set = 0

best\_error = Infinity

while ( iterations < k )

{

    maybe\_inliers = n randomly selected values from data

    maybe\_model = model parameters fitted to maybe\_inliers

    consensus\_set = maybe\_inliers

    for every point in data not in maybe\_inliers

        if ( point fits maybe\_model with an error smaller than t )  
            { add point to consensus\_set }

    if ( the number of elements in consensus\_set is > d )

        /\*this implies that we may have found a good model,  
        now test how good it is\*/

        this\_model = model parameters fitted to all points in consensus\_set

        this\_error = a measure of how well this\_model fits these points

        if (this\_error < best\_error )

        {

            /\*we have found a model which is better than any of the previous

ones,

            keep it until a better one is found\*/

            best\_model = this\_model

            best\_consensus\_set = consensus\_set

            best\_error = this\_error

        }

    ++iterations;

}

return best\_model, best\_consensus\_set, best\_error

# Mathematical Framework

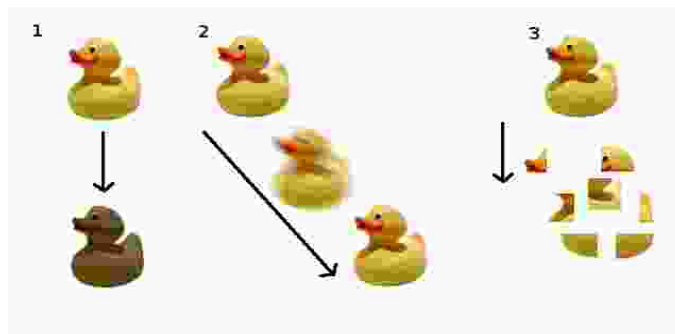
## 4.3 Optical Flow

Optical flow is a term describing the process of registering movement on a moving scene. The goal of optical flow algorithms is to robustly track the points on an image as they move and overcome various ambiguities that rise from the incoherent nature of 3d scenes. There are two kinds of optical-flow algorithms , dense and sparse and they differ in the total number of points they are designed to work on. Dense algorithms are generally a lot more computationally expensive and are typically used in monocular setups to perform both tracking and depth estimation.

There are many modeling approaches on building such an algorithm with the most famous being the Lukas Kanade pyramid[citation needed] method , which will be extensively described , the Horn-Schnuck method[citation needed] , work by Black and Anadan [citation needed]. All the algorithms make some basic assumptions about the world they view and regardless of the way the data is processed ( using pyramids , velocity fields or other constructs ) .

The assumptions for the Lukas Kanade algorithm are the following :

Assumptions	Details	Weaknesses
Brightness Constancy	Tracked surfaces retain the same color between frames	shadow changes , illumination changes , blinking lights , camera exposure changes , image noise
Temporal Persistence	The rate of movement is sufficiently small between frames.	fast motion , rapid movement , large computation times between frames lead to slower frame rate and thus larger movement between frames
Spatial Coherenece	“Large” enough surfaces move in groups	small particles moving in different directions



*Illustration 29: Some bad instances on the optical flow problem , 1 Brightness constancy violation , 2 fast movement out of the detection window , 3 spatial coherence violated*

The assumptions mentioned are translated to mathematical constraints which are checked for being in effect in the neighboring regions of a feature point .

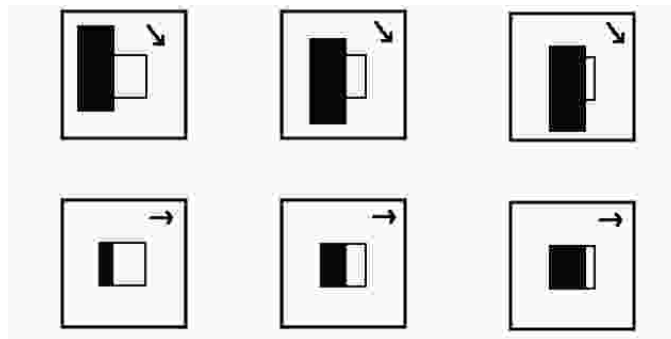
The first one(brightness constancy) is a very straightforward constraint and it basically means that as the time (t) passes , a specific point ( f(x) ) does not change its light intensity , so the the partial derivative of the change of the pixel value divided by the difference of time between the two frames must be zero.

$$\text{Brightness constancy} \quad \frac{\partial f(x)}{\partial t} = 0$$

The second is the rule of temporal persistence and building on the first rule basically means that for every point  $I(x, y, t)$  in a 2D image with coordinates  $(x, y)$  and at a specific time  $(t)$  has the same intensity response on an area “sufficiently close” in space and time  $I(x + \Delta x, y + \Delta y, t + \Delta t)$  , substituting the function  $I$  with the partial derivatives it describes and dividing by  $\Delta t$  we get the final equation which has two unknowns , the velocity on the axis  $x$  and  $y$  , and thus cant be solved , this is were the third constraint comes in.

$$\begin{aligned} \text{Temporal persistence} \quad I(x, y, t) &= I(x + \Delta x, y + \Delta y, t + \Delta t) \\ I(x + \Delta x, y + \Delta y, t + \Delta t) &= I(x, y, t) + \frac{\partial I}{\partial x} \Delta x + \frac{\partial I}{\partial y} \Delta y + \frac{\partial I}{\partial t} \Delta t = 0 \\ \text{dividing with } \Delta t \text{ gives us} \\ (I(x + \Delta x, y + \Delta y, t + \Delta t) - I(x, y, t)) \frac{1}{\Delta t} &= \frac{\partial I}{\partial x} \frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y} \frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t} = 0 \\ \dots &= \frac{\partial I}{\partial x} V_x + \frac{\partial I}{\partial y} V_y + \frac{\partial I}{\partial t} = 0 \\ \frac{\partial I}{\partial x} V_x + \frac{\partial I}{\partial y} V_y &= -\frac{\partial I}{\partial t} \end{aligned}$$

Spatial Coherence , provides us with the last tool required. Having a “large enough” image patch moving together allows us to take into consideration all the neighboring points and build more equations to solve for  $V_x$  and  $V_y$  . The neighborhood can be as large as we want it but a very large window will be easier to violate the coherence constraint , a very small window on the other hand provides less data to work with and suffers from the aperture problem shown in the image.



*Illustration 30: The aperture problem.*

*First row : We have a black rectangle moving diagonally over a small detection window*

*Second row : Inside the detection window movement appears to be horizontal*

For a 5x5 window we have the following over constrained system of equations

$$\begin{bmatrix} I_x(P_1) & I_y(P_1) \\ I_x(P_2) & I_y(P_2) \\ \dots & \dots \\ I_x(P_{24}) & I_y(P_{24}) \\ I_x(P_{25}) & I_y(P_{25}) \end{bmatrix} \begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} I_t(P_1) \\ I_t(P_2) \\ \dots \\ I_t(P_{24}) \\ I_t(P_{25}) \end{bmatrix}$$

$$A \ v = b$$

This is then solved using a least squares minimization

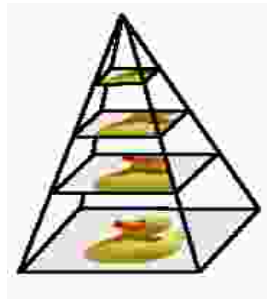
$$\begin{aligned} A \ v &= b \\ A^T A \ v &= A^T b \\ v &= \frac{A^T b}{(A^T A)} \end{aligned}$$

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(p_i)^2 & \sum_i I_x(p_i)I_y(p_i) \\ \sum_i I_x(p_i)I_y(p_i) & \sum_i I_y(p_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(p_i)I_t(p_i) \\ -\sum_i I_y(p_i)I_t(p_i) \end{bmatrix}$$

$A^T A$  is called a structure tensor and the equations can be solved when  $A^T A$  is invertible.

$A^T A$  is invertible when it has two large eigenvectors and this will happen in areas where texture moves in at least two directions. That's the reason corners are good tracking features ( See corner and feature detection ) since they have large two large eigen values.

Though GuarddoG cameras capture frames with a rate of 120 fps on 320x240 and this in theory is a fast enough rate to enforce the temporal persistence , this along with the aperture problem can be mitigated using a gaussian image pyramid , iterating with a variable window.



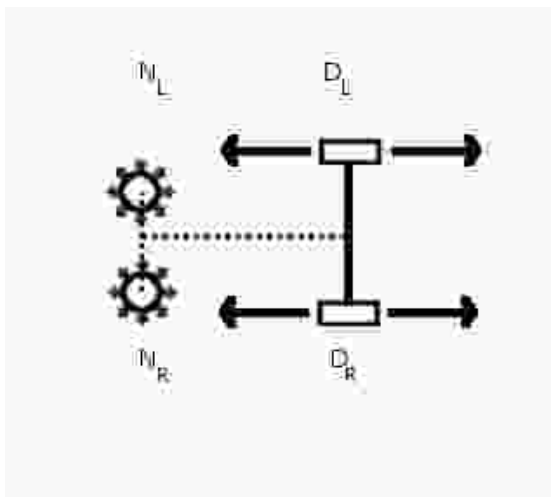
*Illustration 31: A gaussian pyramid window*

# Mathematical Framework

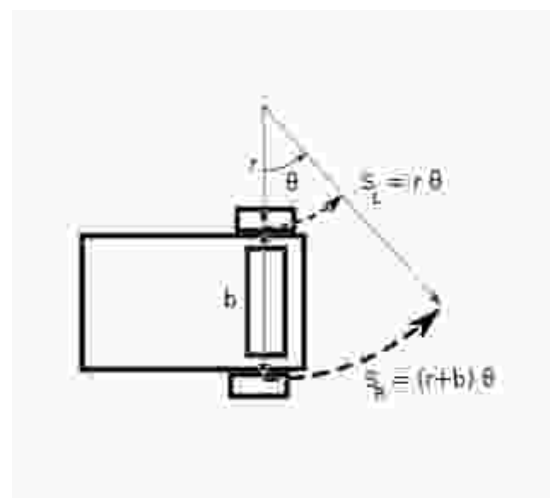
## 5.1 Dead Reckoning

Having reached this point and with the framework described in the previous pages we have a good depth point cloud for the scene viewed by the cameras , an accurate tracking of the camera pose using purely visual means , a list of possible faces detected and this is enough data to start reconstructing the environment the robot will move on. The simplest method for doing this is called dead reckoning. This approach uses a starting point which is considered known and marked as zero and calculates all the subsequent movement data from the pose tracker , the motor encoders and the accelerometer to estimate the next movement point , after the next position point is reached , it is considered known and the calculation produces a new point. The process is repeated for every movement and the result is a tree of movements originating from point zero. This is a computationally cheap and easy to implement method but this ease comes with a cost. The problem using this method is that it has no mechanism for error correction and even minuscule errors in every movement are gradually accumulated and distort the world map generated by the robot. Before discussing a more advanced obstacle/self positioning system that accounts for errors it is important to understand the principles of motion used by a dead reckoning algorithm.

Although this is still the mathematical analysis of the procedure we need to know a little more about the way that the robot is steered in order to make a precise modeling of the procedure. Most robots use differential wheels which is a configuration consisting of three or four wheels from which only two drive and the rest rotate freely in any direction as shown on illustration 33.  $N_L$  and  $N_R$  are wheels can rotate freely in any direction without resisting movement.  $D_L$  and  $D_R$  are two independent electric motors of known diameter and they can rotate in both directions providing motion to the whole robot body. Planning ahead using the kinematics of the platform is a fairly easy task using the geometry of the drive system as shown in Illustration 34

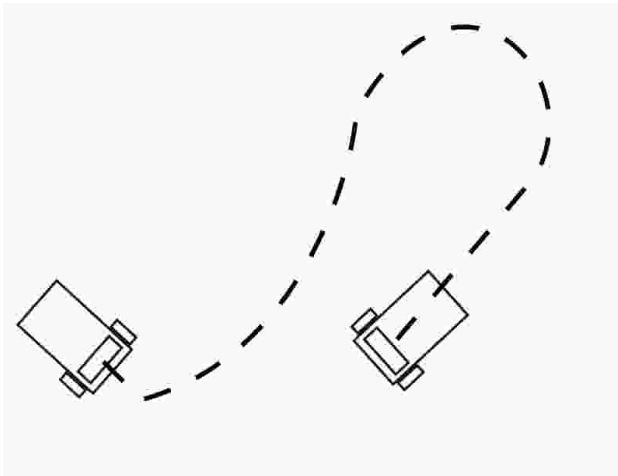


*Illustration 32: The drive system for a four wheeled front differential robot base configuration*

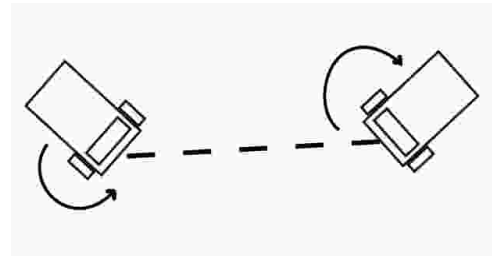


*Illustration 33: The underlying geometry when driving a differential drive system*

Building on the simple mathematical equations derived using the geometry of the differential system ( Illustration 34 ) , we can begin to combine route segments to plan ahead for larger maneuvers. Once we have an array of checkpoints that we have to pass through to reach the required goal ( using the A\* algorithm which is analyzed in detail in topic XXX ) a second algorithm that has the task of maintaining the correct heading on each of the checkpoints by regulating power output to the two motors can output the curved result seen in illustration 35 . Another simpler way to move between waypoints is to first make the required turn and then travel in a straight line. With this type of movement there is no need for constant voltage regulation and motor encoder sampling since either both of the wheels have the exact same power output ( when going straight ) either they have the exact opposite ( when rotating ) In cases were it is not possible ( due to hardware constrains ) to maintain the power fluctuations needed to steer the differential drive with precision the second approach is better and also easier to implement.

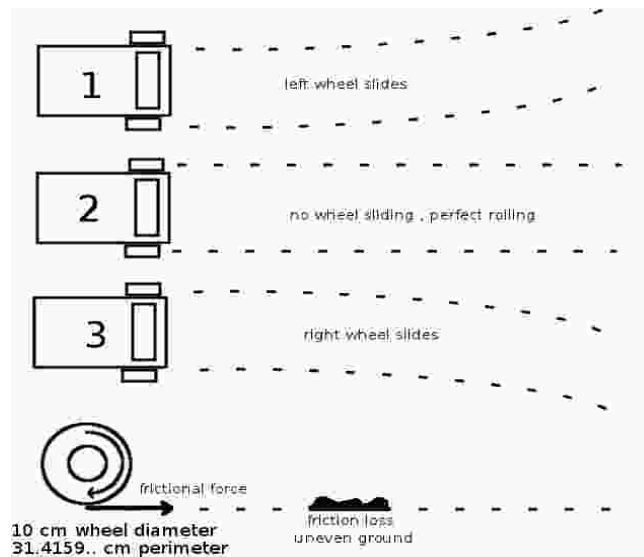


*Illustration 34: A path generated ( to go from left position to the right one ) using gradual turning driven by the differential supplying different power output to the wheels to control the curvature*



*Illustration 35: The same path running both motors at the same power output or at the exact opposite to perform a complete turn.*

As stated many times in this document , this simplistic mathematical model once again does not fully mirror reality. Many additional complications to the problem include uneven terrain , imperfect gears and tires , finite resolution encoders on the motors and latencies during sampling. Needless to say dead reckoning as a generic method does not compensate for any of those and so the real path that the robot travels on starts to deviate gradually until it becomes completely erroneous.





# Mathematical Framework

## 5.2 Simultaneous localization and mapping

Moving forward from Dead Reckoning we must use a more intricate algorithm that will improve precision on the generated map , both in terms of temporal location precision as well as obstacle validity. The basis of SLAM is the same as dead reckoning , the difference is that instead of using a single mathematical point at each of the algorithm iterations the robot is thought to be lying on a two dimensional plane with a given probability for any point on the area. The sensory inputs along with the data coming from the motor encoders produce a new probability distribution and this gradually refines the distribution up to a point when we can be certain about the robot's whereabouts.

A method that overcomes these problems is called Monte Carlo Localization [citation needed] [citation needed] as described in Monte Carlo Localization for Mobile Robots [citation needed] which is implemented as a part of the Mobile Robot Programming Toolkit created by the University of Malaga [citation needed]. An other useful resource for SLAM methods is the OpenSLAM website [citation needed] which features a number of modified versions of similar algorithms.

Monte Carlo Localization is a global localization method , meaning that the algorithm begins with no a-priori knowledge of the position of the robot whatsoever.

The algorithm is based on samples which are possible locations on the world the robot moves on. In each iteration of the algorithm an array of sensor readings is used. In GuarddoG these consist of the encoder values on each of the two wheels , the accelerometer reading , the 2 ultrasonic values and the point cloud with the tracked camera position.

The algorithm works with a three dimensional state vector  $X [ x , y , \theta ]$  that gives the position of the robot and its heading and uses 2 steps , the prediction step and the update step. Prediction step uses the previous , or starting particles and applies the motion model of the robot on each of them by sampling. This approximates a new sample which does not yet incorporate sensor measurements. The update step consists of weighting the sensor readings from the sample we took on the prediction phase against the measurements from sensors and computing the likelihood of having a sample given the specific sensor input. By resampling from the weighted sample set we acquire a new sample set picked using high likelihood samples and the process repeats producing a constant list of the most probable areas that fit both the existing model and the sensory input.

The method is an estimation of the Bayesian filtering problem where we try to approximate the probability of a point  $X$  given  $n$  sensor readings , or  $P(x_n | Z^n)$  . Prediction phase ( using only motion data ) uses

$P(x_n | Z^{n-1})$  and  $P(x_n | x_{n-1}, U_{n-1})$  obtained by integration

$$P(x_n | Z^n) = \int P(x_n | x_{n-1}, U_{n-1}) P(x_n | Z^{n-1}) dx_{n-1} .$$

\*

The Update phase utilizes the sensory input  $Z$  to produce  $P(x_n | Z^n) = \frac{P(z_n | x_n)P(x_n | Z^{n-1})}{P(z_n | Z^{n-1})}$

Further resources about the theoretical justification of the algorithm are provided in [citation needed] [citation needed] as mentioned in the original publication from the team of Dieter Fox [citation needed].

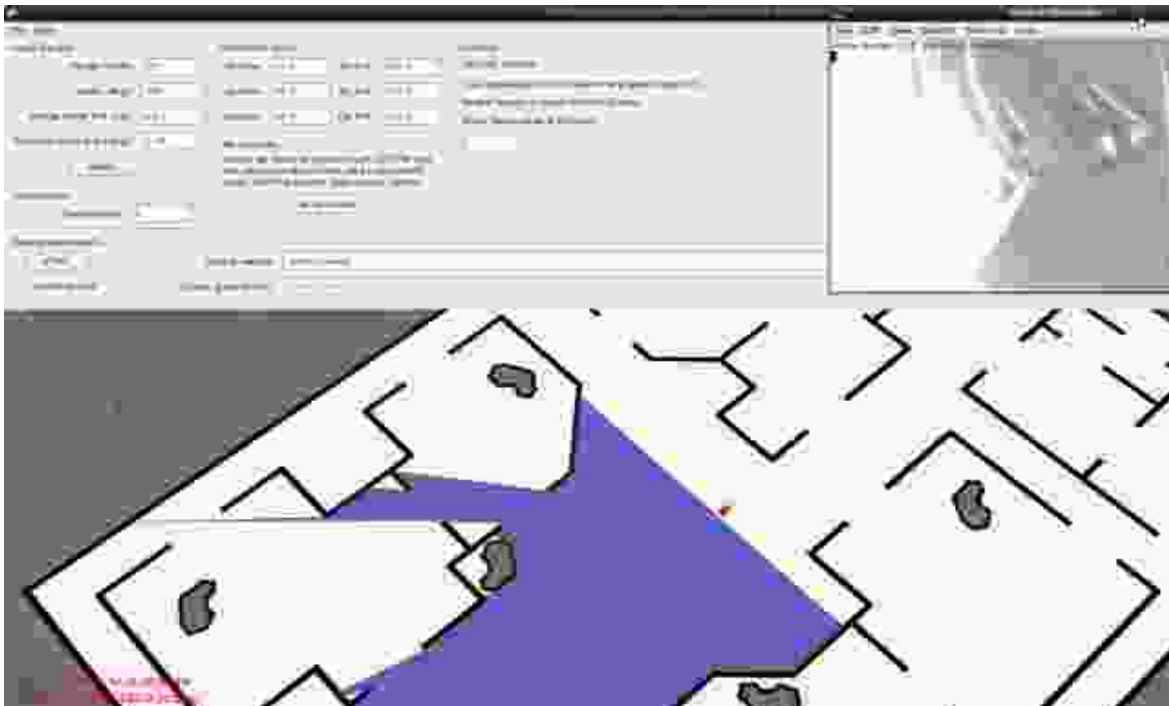
### Monte Carlo Localization Algorithm

```

input:
  Distance  $U_t$ 
  Sensor reading  $Z_t$ 
  Sample set  $S_t = \{(X_t(i), W_t(i)) | i=1, \dots, n\}$ 

//PREDICTION PHASE
for (i=1; i<n; ++i) // Update the current set of samples
{
   $X_t = \text{updateDist}(X_t, U_t)$  // Compute new location using motion model
   $W_t(i) = \text{prob}(Z_t | X_t(i))$  // Compute new weighted probability
}
//UPDATE PHASE
 $S_{t+1} = \text{null}$ 
for (i=1; i<n; ++i) // Resample to get the next generation of samples
{
  Sample an index  $j$  from the distribution given by the weights in  $S_t$ 
  Add ( $X_t(j), W_t(j)$ ) to  $S_{t+1}$  // Add sample  $j$  to the set of new samples
}
return  $S_{t+1}$ 

```



*Illustration 36: An instance of the Monte Carlo Localization Algorithm in the MRPT simulation application*

## Mathematical Framework

### 5.3 A\* Path Finding

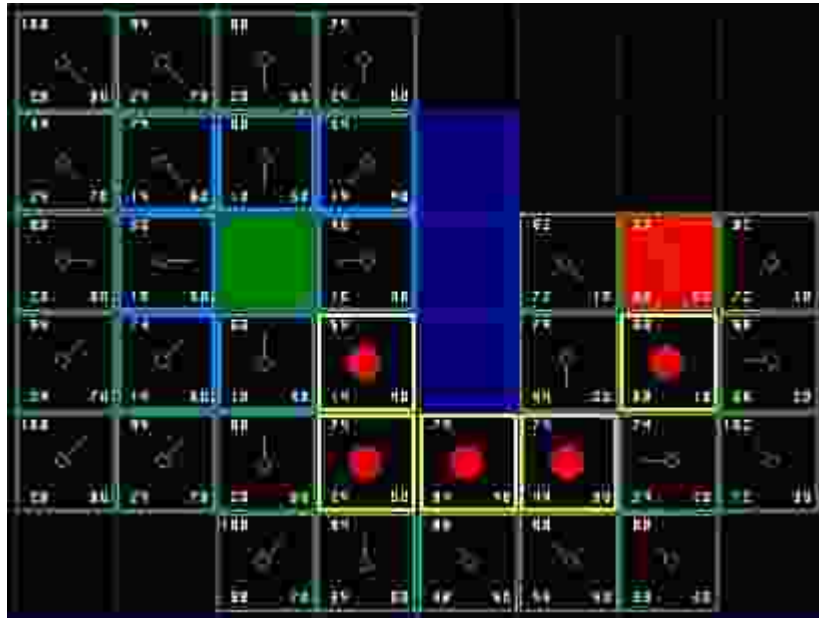
Assuming a two dimensional map acquired by the operations above, and a stable track of the position of the robot, there is need for an algorithm to perform path finding, in order for the robot to be able to reach a target position and dynamically change its course when new obstacles are detected. The algorithm used by this project for this kind of functionality is A\*, an extension of Dijkstra's graph search algorithm. Successful path finding is very critical because it means less battery drain due to unnecessary movements and better performance as a guard.

A\* uses a heuristic that has to never over-estimate the route cost, and such a heuristic is the Manhattan distance that is commonly used by many implementations.

The complexity of the algorithm is  $|h(x) - h^*(x)| = O(\log h^*(x))$  where  $h$  is the heuristic used.

The cost of the algorithm for each new node is calculated using  $f(n) = g(n) + h(n)$  where  $g$  is the cost of the transition to the new node and  $h$  the heuristic for the transition to the goal node.

A\* is thus admissible since adding  $g$  which is an exact estimation of the distance from the source node to the optimistic heuristic since will always make the algorithm seek the solution with the lowest possible cost.



*Illustration 37: A\* algorithm run instance, every block has the manhattan distance on the lower right corner, the previous step distance on the lower left corner, and the sum on the upper left corner.*

### **A\* Algorithm**

OPEN SET = START NODE

CLOSED SET = EMPTY

**while** the node with the lowest cost in OPEN SET is not the GOAL NODE:

current = **remove** lowest rank item **from** OPEN SET

**add** current **to** CLOSED SET

**for** neighbors **of** current:

cost =  $g(\text{current}) + \text{movementcost}(\text{current}, \text{neighbor})$

**if** neighbor **in** OPEN **and** cost **less than**  $g(\text{neighbor})$ :

**remove** neighbor **from** OPEN, /\*new path is better\*/

**if** neighbor **in** CLOSED SET **and** cost **less than**  $g(\text{neighbor})$ :

**remove** neighbor **from** CLOSED SET

**if** neighbor **not in** OPEN SET **and** neighbor **not in** CLOSED SET:

**set**  $g(\text{neighbor})$  **to** cost

**add** neighbor **to** OPEN SET

**set** priority queue rank **to**  $g(\text{neighbor}) + h(\text{neighbor})$

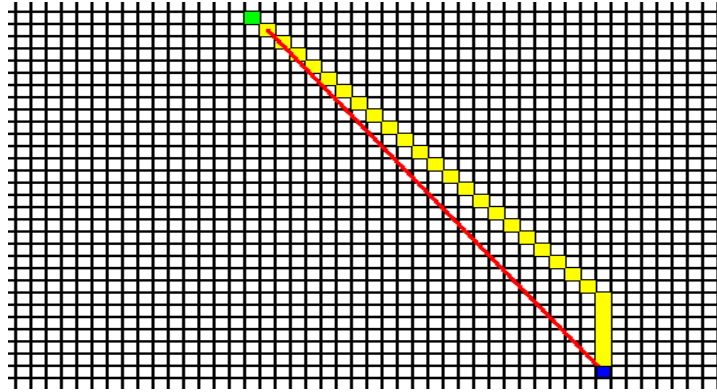
**set** neighbor's parent **to** current

Reconstruct path following parent pointers from goal to start

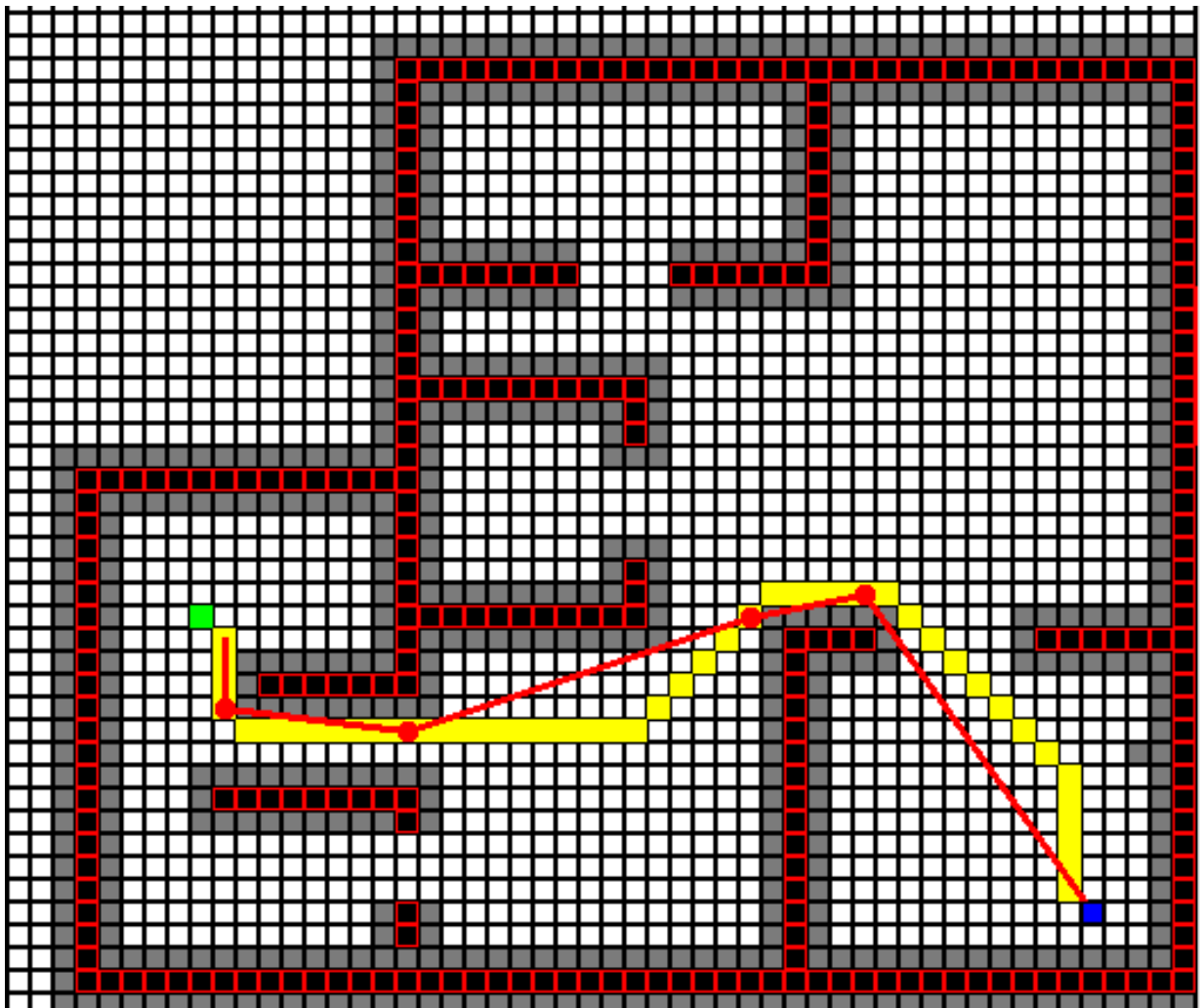
One of the shortcomings of a raw implementation of an uncustomized A\* algorithm is that in the real world diagonal movement is a little further away than horizontal ( pythagorean theorem ). The result is that returned paths can be “non optimal” for a real world moving robot. Added to this problem comes the fact that in physical movement one tends to hold a course turning as little as it is possible. A\* can provide an optimal solution that has many turns , but this will take more time for the robot to be traversed. The solution to this problem is keeping the heading of the robot as an information vector on every opened node and adding an extra weight when turns are made , while also adding an extra weight when performing diagonal movement to balance them.

The final element needed is a way to represent uncertainty about the mapped obstacles since there may be errors in the input , not only caused by “mis-detection of obstacles” but also by the the lack of detail of the map since an area of 200 m<sup>2</sup> quantized at a scale of 10 cm<sup>2</sup> per block results in an array sized 2000x1000 that cannot reflect the full complexity of the scene.

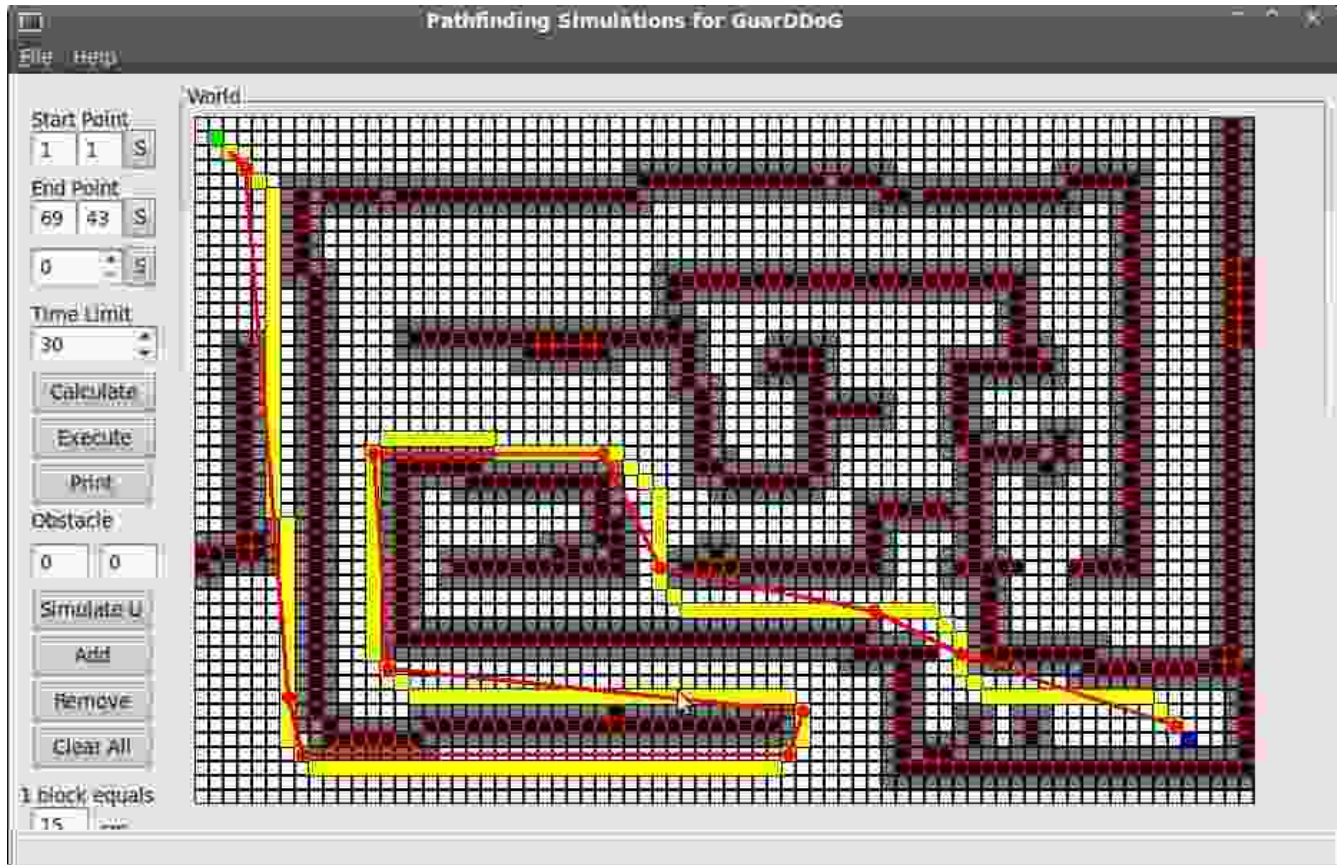
Using these modifications , the output becomes better but there is a further improvement that can be achieved by using the largest possible straight paths to connect sub regions of the A\* paths. Doing that the turning maneuvers of the robot are reduced to the fewest possible. To achieve that , after a path has been extracted ,instead of reconstructing the path following the parnet pointers we use a second pass algorithm runs which casts a line ( using Bresenham's line algorithm ) from the last step of the path to all the previous ones until an obstacle is detected. The previous point before the obstacle is then marked as connected to the first one and the algorithm continues until the source node is connected. This improves the operation of the robot . This could also be improved in the future to use odometer based curves instead of point to point turning , something that would also make the movement of GuarddoG seem more life like.\*



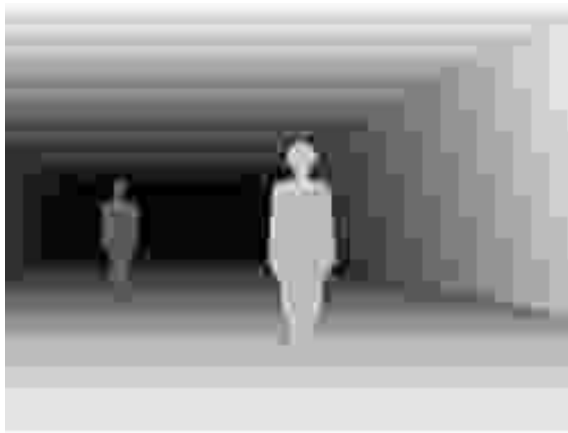
*Illustration 38: The problems that may occur using an uncustomized A\* Algorithm , and how they are corrected*



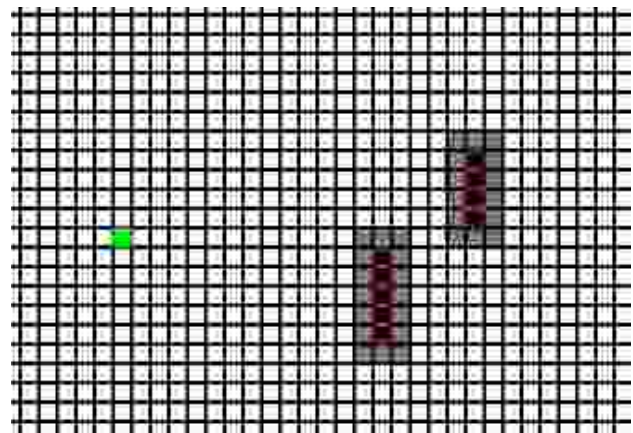
*Illustration 39: The green block is the source , the blue the target , red/black blocks are obstacles and gray areas , areas of uncertainty. The yellow path is the one that A\* returns and the red line the compressed path for as little turning as possible.*



*Illustration 40: A small maze like instance for the A\* algorithm on the GuarddoG world mapping GUI and the output path*



*Illustration 41: The 3D appearance of obstacles*



*Illustration 42: The 2D appearance of obstacles, which can be detected by ray casting on the depth map of illustration 37*

# Mathematical Framework

## 6.1 First-order logic and a Wumpus like world

GuarddoG lives in a Wumpus like world , or a Shakey one also taken from Russel & Norvig's AI a modern approach. Its mission is to find intruders in a random home layout . It is only natural for an agent operating in such a kind of world to use first-order logic and forward chains of inference to decide its actions and interact with his human owners. GuarddoG uses a string passing interface for executing jobs. It features some direct and immutable commands such as forward , backward , left and right , labels such as kitchen , living room , toilet and operators to combine them. Although inference rules have been removed from the design ( at the time of writing ) to reduce the surface of the project they are presented here for reference and they will be reinstated in future versions of the robot.

This kind of functionality on one hand unifies the command interfaces of the robot and on the other hand makes it more intelligent and human-like. Whether the robot is controlled via a voice to text module , a handheld mobile device , a computer or a web interface the input is always strings of english sentences or buttons that can be aliased to strings and this makes development much more practical and the robot much more easy to control since it responds in the same way , whatever the medium of communication.

The syntax of the commands is simple and it looks like this

```
FORWARD(100cm)
NEW_PLACE(KITCHEN)
GOTO(TOILET)
SIGNAL ALARM
AUTONOMOUS MODE(1)
```

Inference can be used by creating an object model such as the one used in the OpenMind [citation needed] project which is based on a real world knowledge base. In the future a system possessing such a database coupled with a vision based object recognition algorithm could make correspondances between visual cues and their string descriptions that would easily be integrated to a the unified string interface described above. Such recognition engines for point clouds already exist with most notable the RoboEarth project [citation needed] which strives to be a world wide web for robots and where every object recognized by one robot can then transmit its knowledge and share it with all the other robots. A list of the possible commands that GuarddoG can execute can be found in the software unified string interface topic.

There is no point in further analyzing the mathematics behind first-order logic calculus in this document since it is a mature and well documented subject . The book Artificial Intelligence of Stuart Russel and Peter Norvig is an invaluable resource for the theory behind AI systems. [citation needed] Robots with human-like artificial intelligence have a long long way to go and this is apparent for any one that has a good understanding of the problem's vast nature. GuarddoG focuses on seeing and space perception and tries to acknowledge simple commands and scripts.

\*

# Implementation of Mathematical Framework

## 6.2 The big picture

Having explained all the key theory concepts its time to move away from the purely mathematical/algorithmic domain to the real implementation onboard GuarddoG :

<b>Camera Model</b>	A pure pinhole camera model abstraction is assumed
<b>Camera Calibration</b>	Camera calibration is performed a priori using a 10x7 chessboard pattern and the OpenCV implementation to be able to be comparable to existing projects. The calibration parameters extracted are stored in a file and used through the project.
<b>Image Rectification</b>	The calibration parameters supplied by the calibration step are read from the storage file at each program startup and are used by the pipelining to transform the raw images coming from the cameras to calibrated equivalents where the mathematical model of the pinhole camera is in effect.
<b>Image Processing</b>	The calibrated pair of images are first compared to the last pair received and if they have a “noteable” difference , acquired by direct subtraction of each of the pixels , they are transformed using convolution matrices to their gaussian blur , sobel , second-derivative and integral image representations. If the scene is static they are not processed at all saving cpu time.
<b>Feature Corner Detection</b>	The sobel representation passes through the algorithm explained in the corresponding theory topic and a list of corners is returned and paired to the image.
<b>RANSAC / Homography Estimation Optical-flow estimation</b>	The corner list extracted is juxtaposed to the last corner list and using RANSAC and the OpenCV implementation a precise approximation of the transformation that took place is returned.
<b>Haar Wavelet Detection</b>	Using the input images and cvHaarDetectObjects with a training set for face detection , a list of faces is returned.

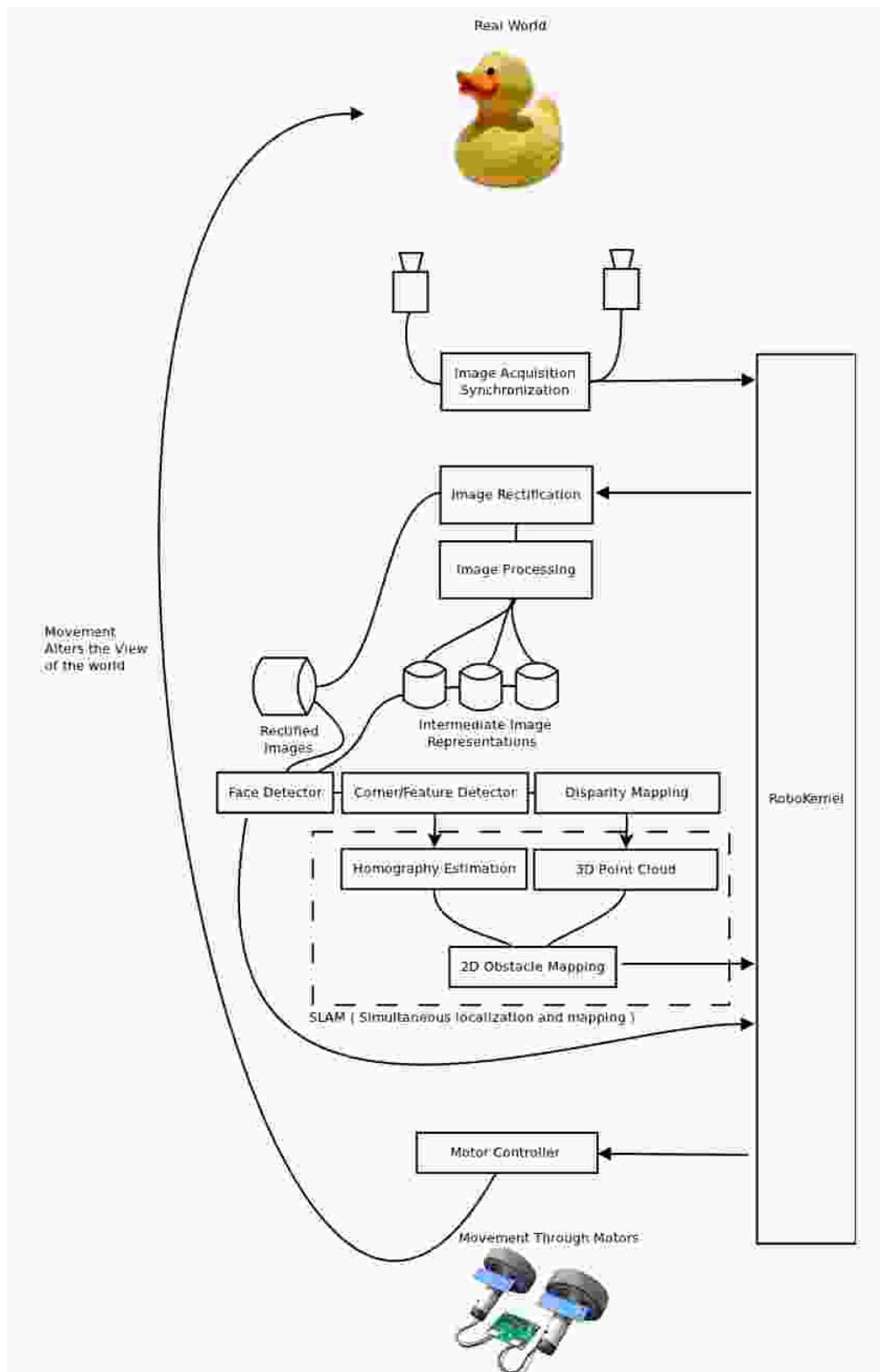


<b>Disparity Mapping</b>	GuarddoG can use any one of the algorithms described in the disparity mapping topic to extract depth information from a pair of images.
<b>A* Path Planning / Dead Reckoning</b>	Using sensory input from the ultrasonic sensors and the motor encoders as well as the depth map , the camera transformation extracted with the homography , the face list and accelerometer data the position of the robot along with close obstacles are added to a 2D map where A* can be executed to provide a path towards the target position of the robot.
<b>First-order scripting logic</b>	State keeping and driving the robot towards a useful purpose , leveraging the different abilities the robot and performing the desirable work is assigned to the scripting logic that synchronizes all of the individual parts of the project and checks for errors in their execution.

All of the above have been implemented as part of the project , but the part that is currently an implementation deficiency is the scripting logic “Robot Hypervisor” that manages the many sub libraries used by the project. SLAM is also not implemented but used on tests using an external completely separate project called MRPT. Adding to the difficulty to implement it are hardware constraints , such as the bad quality of the physical handmade assembly of the robot , the fact that it is mainly powered using the electrical grid which means dragging heavy power cables which get tangled often. All these make position estimation very difficult and are discussed in the next chapter of this document which explains hardware related choices and limitations.

It is also a note worthwhile , that most of the complex functionality resides in the “visual cortex” and scripting logic. Sensor communication and low level details although they too adhere to mathematical principles are very common place in bibliography and as implementations even on hobby level robotics. For those reasons and reasons of concision they have been exempted from this theoretical analysis.

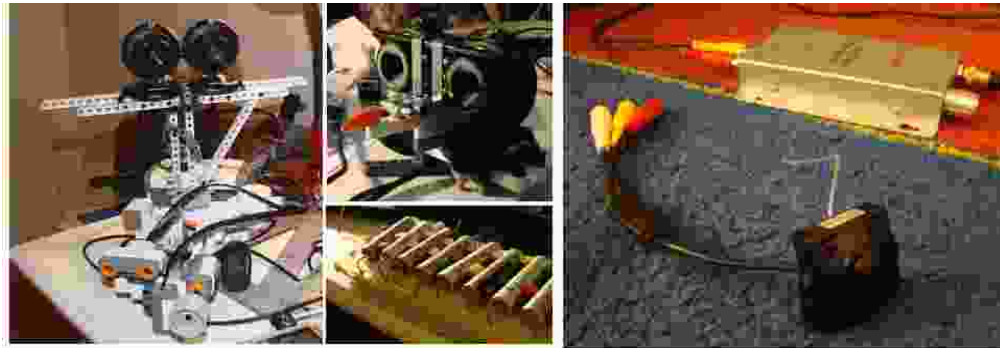
The next illustration outlines the way the individual methods form a common data pipeline implemented in VisualCortex , MasterPathPlanning and the MotorHAL libraries of the project . The “intelligent” part of the project resides in the right part of the RoboKernel , not shown in this graph . The implementation follows the design that arises naturally from the nature of the operations performed on the data and all notation inside the program's source code matches the notation used scientifically and in this paper to make the code more maintainable and easier for a potential new programmer that may want to use it.



*Illustration 43: A schematic of the pipeline of data as they go through the system. This image is the connection diagram for all the methods presented here*

# Hardware

## 2.1.0 Overview



*Illustration 44: Early experimentations while trying to create the initial GuarddoG platform*

Building the physical platform of GuarddoG from scratch was a daunting task , partly because it meant delving into uncharted waters for a computer scientist and partly due to the numerous options available that should be tried and dismissed after a trial and error procedure. This has little if none scientific value whatsoever but is a good warning about the kind of problems one will face when implementing these algorithms in real world application , and not just in a computer simulation.

The project started with an implementation based on the Lego mindstorm kit with wireless transmission of video and commands to the robot via bluetooth. This proved to be a wrong approach for a number of reasons which became evident as time passed , due to the small range of bluetooth , the issues of privacy when broadcasting unencrypted video , cost , small size and bad viewpoint , stability and many other design problems. The second step was moving away from this approach and performing all computations on board the robot , which meant larger power consumption , larger batteries more weight and a chassis that should support it.



*Illustration 45: Moving on a local processing solution while still using the lego mindstorm kit*



*Illustration 46: The cotton filled wheels and the attempts to reduce weight by changing the PSU and other parts*

This design proved to be better suited for the task but the mindstorm kit reached its limits , mainly due to the weight of the whole contraption which could not any more be supported by the small motors and wheels. Many other solutions were tried ( such as filling the plastic on the wheels with cotton ) reducing the size of the PSU and others but the idea of using the mindstorm kit was finally abandoned.



*Illustration 47: Moving on to GuarddoG mk4*

After a lot of iterations a plastic body with a rigid base was chosen which is the ideal fitting size for the project but issues of power consumption still remained. Instead of moving the whole 4+ kg base every time a look towards a new direction was needed , it was much more efficient to turn just the “head”. Other problems included the difficulty of calibrating the two cameras since their relative alignment changed as the robot moved because they where loosely hold together by a clamp like wooden board. To improve this a 2 degrees of freedom head was made from 2 tuppens ( which is actually the most cost effective way to make one ) along with a laser cut plexi glass clamp that fitted exactly the camera dimensions thus improving , but not solving , some of the alignment and calibration problems. To improve camera tracking in low texture , low brightness areas two headlights were added to the design that could occasionally flash for illumination , and interaction with humans to provide visual cues for the state of the robot. Along with them an arduino which is open hardware with excelent documentation along with ultrasonic sensors , an accelerometer and other peripherals was included in the design.

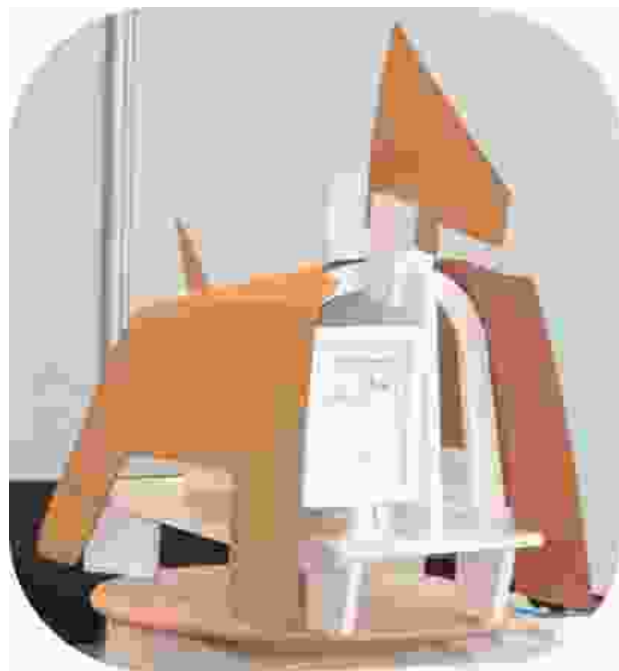


*Illustration 48: A more recent ( at the time of writing ) state of the GuarddoG physical implementation*

Though this design was the most fitting for the job it was still problematic mainly due to the poor workmanship on my part and the fact that the different ideas have been literally patched the one on to the other as they got added to the design. Thus knowing what the final requirements where after a long procedure and trial error a new GuarddoG was designed using CAD in order to be able to be produced at fixed parts and made easier to assemble and disassemble instead of relying on random parts. One of the most important final changes was the use of a new camera pair which will be detailed on the following topic since it was a major improvement to the old camera set of the robot.



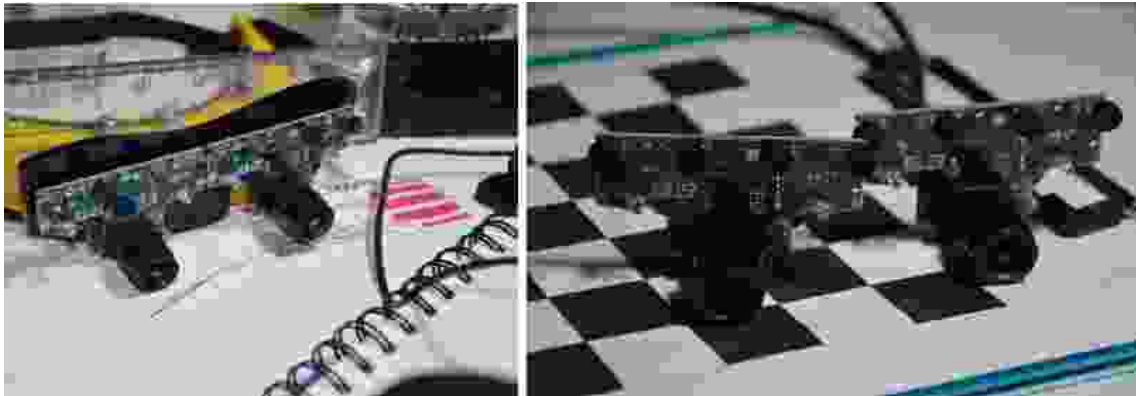
*Illustration 49: GddG mk4*



*Illustration 50: GddG mk5 a.k.a. Jack mockup , the final version of GuarddoG which is yet to be constructed*

## Hardware

### 2.1.0 Camera Sensors and Synchronization issues



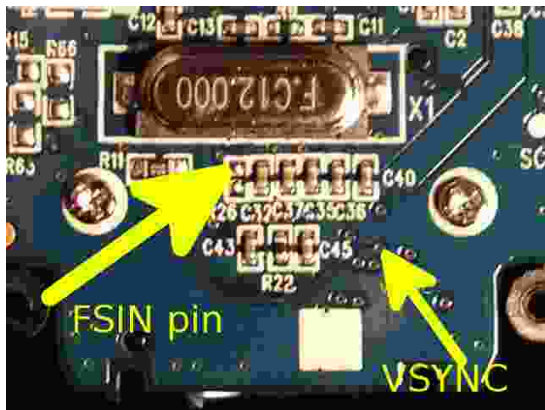
*Illustration 51: Left : The CAD designed and laser cut plexiglass rig that keeps the cameras aligned correctly , Right : The PS3 cameras without the rig.*

The cameras used by GuarddoG are based on the OV7720/OV7221 CMOS VGA (640x480) Sensor , and are cheap and easy to find as they are the camera system used by the Playstation 3 Gaming Console

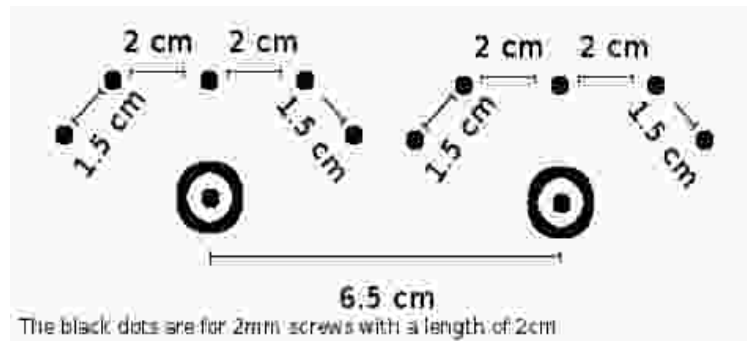
#### Camera Sensor Key Specifications

Array Size	640 x 480
Power Supply Digital Core Voltage	1.8VDC + 10%
Power Supply Analog Voltage	3.0V to 3.3V
Power Supply I/O Voltage	1.7V to 3.3V
Power Requirements - Active	120 mW typical (60 fps VGA, YUV)
Power Requirements - Standby	< 20 $\mu$ A
Temperature Range	-20°C to +70°C
Output Format (8-bit)	<ul style="list-style-type: none"><li>• YUV/YCbCr 4:2:2</li><li>• RGB565/555/444</li><li>• GRB 4:2:2</li><li>• Raw RGB Data</li></ul>
Lens Size	1/4"
Max Image Transfer Rate	60 fps for VGA
Scan Mode	Progressive
Electronic Exposure	Up to 510:1 (for selected fps)
Pixel Size	6.0 $\mu$ m x 6.0 $\mu$ m
Fixed Pattern Noise	< 0.03% of VPEAK-TO-PEAK
Image Area	3984 $\mu$ m x 2952 $\mu$ m
Package Dimensions	5345 $\mu$ m x 5265 $\mu$ m

Stereo vision on a mobile robot traditionally requires expensive hardware-synchronized cameras. Because standard stereo reconstruction algorithms assume that the images from the left and right cameras are captured from a common scene at the same time , any motion that occurs between the left and right cameras snapshots is equivalent to an error in the model. This error, causes the quality of the depth mapping to decrease and the distance notion of the robot to be distorted , something that in turn impacts all of its functionality as errors tend to accumulate .



*Illustration 52: The position of FSIN and VSYNC on the camera board*



*Illustration 53: Screws on the PS3 cameras and a schematic of the alignment used*

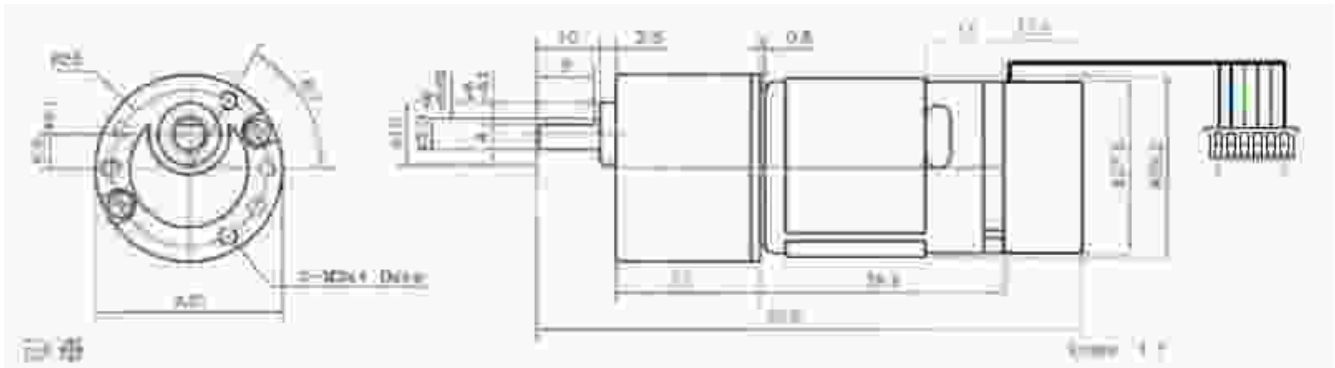
Hardware synchronization , the process of forcing two or more cameras to share a common hardware clock , has been traditionally available only in professional stereo vision systems. Thankfully , the inexpensive PS3 Eye camera is built on the same high-end OmniVision OV7720 chip set that is comparable to those found in many machine vision cameras. These cameras can be hardware-synchronized using the exposed frame clock input (FSIN) and output (VSYNC ) pins . By shorting one camera's VSYNC pin to the others cameras FSIN pins the cameras are forced to share a common clock . To reduce the risk of a difference in ground potentials damaging the OV7720 delicate circuitry , each camera has to be also modified to share a common ground .

This hardware synchronization guarantees that both cameras capture images simultaneously , but does not guarantee that the frames will travel retaining their synchronization when sampled using the Universal Serial Bus (USB) . Each camera has its own hardware clock and that means that in addition to the small distortion in space ( due to optics ) we have a small distortion in the fourth dimension , the axis of time. To tackle this problem GuarddoG uses cameras that have a very fast refresh rate of 120fps @ 320x240 or 75fps @ 640x480 pixels with a rewired shutter (FSIN , VSYNC pins ) in order for synchronization on the hardware side of the camera snapshots. A secondary problem is that there is non uniform latency over the USB cable and the USB host controller . This problem is combated using direct frame grabbing via V4L2 , zero-copy passing by pointer to the beginning of the image pipelining and static linkage of the libraries consisting of the project to reduce delays and overheads.



## Hardware

### 1.1.0 Motor System and Peripherals



GuarddoG uses two EGM30 motors made by Devantech which feature an encoder a 30:1 gearbox and work at 12V. They are rated for usage in medium size robotics applications ( weights up to 5kg ) and perform very well.

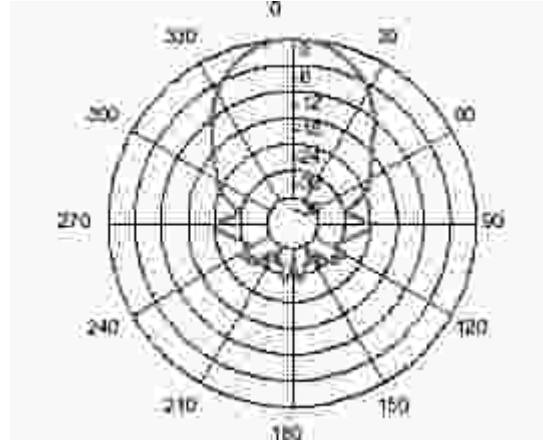
The manufacturers technical specifications follow for reference .

Rated voltage	12v
Rated torque	1.5kg/cm
Rated speed	170rpm
Rated current	530mA
No load speed	216
No load current	150mA
Stall Current	2.5A
Rated output	4.22W
Encoder counts per output shaft turn	360
Minimum Speed	1.5rpm
Maximum Speed	200rpm





*Illustration 54: SRF-05  
Ultrasonic Sensor*



GuarddoG uses 2 ultrasonic sensors positioned in the left and right front of the robot. Although the robot has a fairly robust vision system and depth estimation it has its limits. Just as a human that stands right in front of a wall has no way of determining the obstacle since there are literally no visual cues except by the haptic feedback one experiences by touching it so does a robot. Due to the absence of arms and synthetic skin ultrasonic sensors come in as a replacement and enable the robot to have a sense of obstacles on very dark and very close situations.

Frequency	40kHz
Max Range	4 meters
Min Range	3 centimeters



*Illustration 55:  
Memsic 2125  
Dual Axis  
accelerometer*

To help with the task of registering movement correctly a low cost dual axis accelerometer is affixed to the chassis of GuarddoG. It can register accelerations and decelerations that might be caused by bumping on a wall somebody pushing the robot or in case of sudden change in motions. Although an IMU provides much more precise data when implemented along a vision system [citation needed] , due to the small budget of the project this just had to do. A larger three axis accelerometer could in the future be affixed to the head of the robot to help the vision system conserve its computational powers just like the ear labyrinth does on a human.

Measures $\pm 3$ g on each axis
Low current operation , less than 4 mA at 5 VDC
Movement/Lack of movement detection

# Hardware

## 1.1.0 The Energy-Heat-Weight-Cost Problem

Maybe the most mission critical and expensive element for a robust robot right now is its power supply and battery autonomy. Although the computational speed of modern computer systems has continued to rise in exponential rates while also slowly decreasing its power consumption, the same scale of improvements has not been made in batteries and consumer grade available power technology.

The power consumption for a mobile robot is very high and relative to the mass, movement speed and total mileage targeted for a platform and reversely related to the maximum service time of the robot between two charging sessions. GuarddoG is supposed to guard homes and offices of 100m<sup>2</sup> area something which takes a great toll on the total autonomy time.

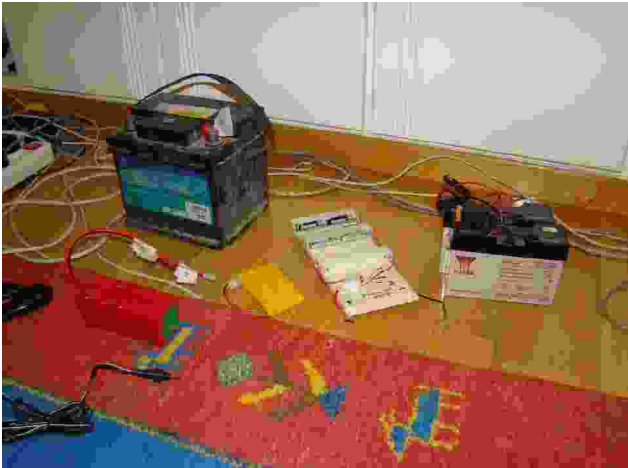
By increasing the capacity of the battery one increases the cost and (or) weight of the robot. By increasing the weight of the robot this increases the energy required to carry the battery around and thus the waste “heat” produced.

The problem can be mitigated by including one or more charging stations in the facility that the robot operates in. This way they can act as rest points in between patrols. According to the battery chemistry ( since most contemporary battery technologies don't have memory issues anymore ) this can also be used to keep the charge level sufficiently high and thus delay full charges until the battery level is so drained that it will not last for another patrol number multiplied by a safety factor.

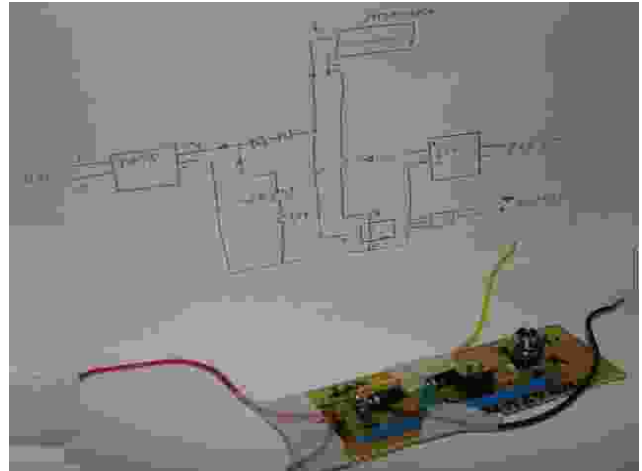
Battery Technology	Cell Voltage	En. Density Mj/Kg	Cost
NiCd	1.2V	0.14	€
NiMH	1.2V	0.36	€
Lead/Acid	2.1V	0.14	€ €
Lithium/ion	3.6V	0.46	€ € € €
Fuel Cell	-	1.5+ (?)	€ € € €
Gasoline/Diesel	-	70+	€ € €

A more long-term problem is that all the battery cells mentioned above have a finite ( and relatively small ) recharge cycle lifetime. Lithium/Ion for example lose 10% of their capacity each year in the best case, without taking into account the damage done to the battery by overheating, high current demand spikes and other real usage scenarios.

This is a very large problem since robots should ideally have the same kind of operational uptime as web-servers and other computer infrastructure but an empty battery can be a catastrophic failure which will render them useless. The same goes for their life cycle as products and the lifetime of the other parts of the robot that should be the maximum possible in order to reduce waste and improve cost effectiveness for a possible buyer.\*



*Illustration 57: Battery types tested, none fitted the cost/weight-energy requirements of GuarddoG but NiMH was the chosen chemistry*



*Illustration 56: The custom power "switch" between charging AC operation and DC operation, with help from Nikolas Zervos (Telcom Greece)*

Other novel ideas on power supply solutions is wireless transmission of power ( for small scale devices ) , which has a long way to mature and will probably never be sufficient to move a 5+ kg device.

As a part of the GuarddoG design work , a large number of options was tested and the final battery pack chosen was a custom 12V NiMH 10 cell battery that could only power the robot for approximately 1h ( on an average power consumption scenario ). Other alternatives were tested but due to the lack of available funds the power supply is currently a large deficiency both in GuarddoG as well in most other commercial available robots.

The most promising technology for robots seems to be Fuel Cell technology which uses water and the electrical grid to charge fuel cells that are like small engines and in turn generate electricity on board the robot in the fashion of gasoline that recharges the battery in a car. An other source of relatively cheap power is combining battery packs from laptops that since to their large scale economies can be relatively cheap.

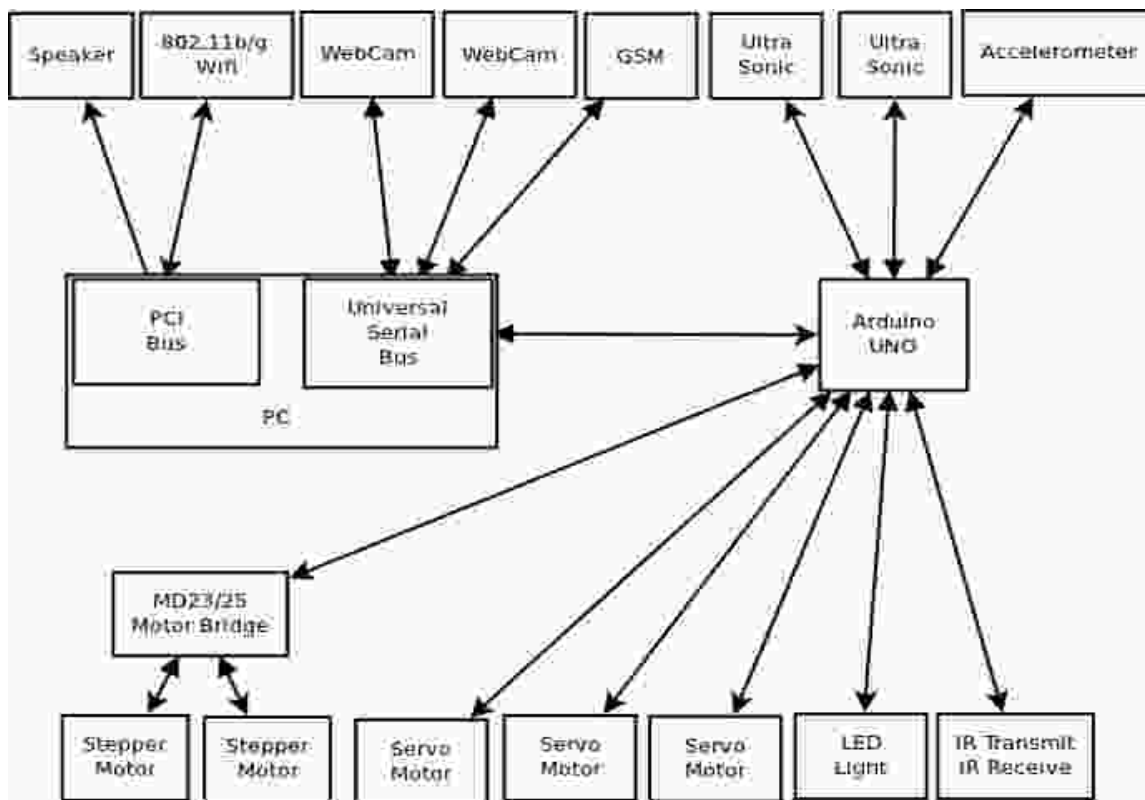
Not having made a final decision about the battery type , meant not being able to make a charging station for it , so a small switching module was designed and implemented that basically switches between 12V DC from the onboard transformer from 220V AC , the charger of the NiMH battery and the NiMH battery itself.

GuarddoG mk4 currently uses an Intel Celeron 1.2Ghz Mini-ITX Motherboard. Instead of a hard-disk a flash memory USB stick is used as a hard-drive reducing 5W of consumption and making the robot more shock resistant. To further cut down battery drain the GuarddoG platform is planned to move to an ARM architecture so that the logic computers could be directly powered using 5V 1A supply , something that will make the robot be able to drop the need for an onboard transformer and make it even lighter. Eventually using composite materials the weight budget will be 70+% distributed to the battery and motor system and the rest 25% to the chassis , leaving 5% of the weight for electronics. A project that has not yet been released at the time of writing but will probably be the host computer for future versions of GuarddoG will be the Raspberry Pi featuring an ARM 700MHz CPU 256MB RAM and USB support. Due to its low specs there might be a cluster of two or more involved in the design , one dealing only with the visual routines and the other one with controlling the peripherals , logic and Speech to Text. Functionality. The peripherals of GuarddoG are controlled by an Arduino which has a very small energy footprint compared to the motors and main computer. Networking is based on a 802.11 b/g WiFi adaptor that operates as a WiFi access point and through a GSM module that can operate without WiFi infrastructure utilizing SMS and possibly GPRS/3G connectivity.

The GSM module can also build upon the existing infrastructure for consumer cellphones , since each robot will have a unique IMEI ( International Mobile Equipment Identity ) enabling some degree of theft protection. In Greece since 2010 it is mandatory to associate a personal identification number with each mobile phone number , so this also could serve as a framework for managing whose property a robot is.

Of course a consumer robot application should use a much more powerful CPU than the one that GuarddoG relies for computing tasks. A custom VLSI could also provide a substantial speed up with low power consumption requirements , and the same would also be true about the arduino part of the project which could be more robust using a custom integrated circuit more closely coupled with the main system. Alas these things were out of scope for this project so they are ideas not yet realized.

Having explained the reasoning and the problems we are trying to resolve , the following system diagram gives a clear look on the internal communication scheme. The Arduino UNO uses I2C communication or analogue sampling to communicate with its peripherals and the onboard firmware is responsible for exposing a high level digital interface for all of them.



*Illustration 58: A connection diagram for the different hardware modules*

## **Hardware**

### **1.1.0 Parts List**

#### **Casing Costs**

Various Plastics , screws , etc = 45 euro

#### **Embedded Electronics**

1x Arduino = 25 euro ( Uno )

3x Infrared Led = 3 euro

1x RD-01 ( or RD-02 Devantech motor kits ) = 130 euro

2x Buttons ( power -on ) = 2 euro

2x Switches ( power supply ) = 2 euro

2x LED HeadLights = 10 euro

2x Ultrasonic Devantech SRF-05 with mounting = 40 euro

1x Dual Axis Accelerometer ( memsic 2125 ) = 30 euro

Subtotal : 252 euro

#### **Computer Hardware**

( 1x Fan = 5 €

1x AC-DC 12 V Converter = 30 €

1x PicoPSU 90W = 45 €

1x Mini-Itx Motherboard = 65-75 € ( Currently on guarddog Intel D201GLY2 )

1x 512-2048MB RAM DIMM ( on guarddog 512MB DDR2 ) = 30 € )

or

( 1x Car (12V) USB power supply = 5 €

2x Raspberry Pi's = 52 €

2x USB cables = 3 € )

plus

1x WIFI PCI card ( WG311T ) or USB = 30 €

2x Webcams ( On guarddog MS VX-6000 ) = 92 € , 2xPS3 Eyes = 60 €

1x USB Flash Drive 8GB + = 20 €

Subtotal : 327 € mk4 (old hardware) or  
170 € mk5 (new hardware)

**Total 624 € mk4 or 467 € mk5**

( batteries not included )

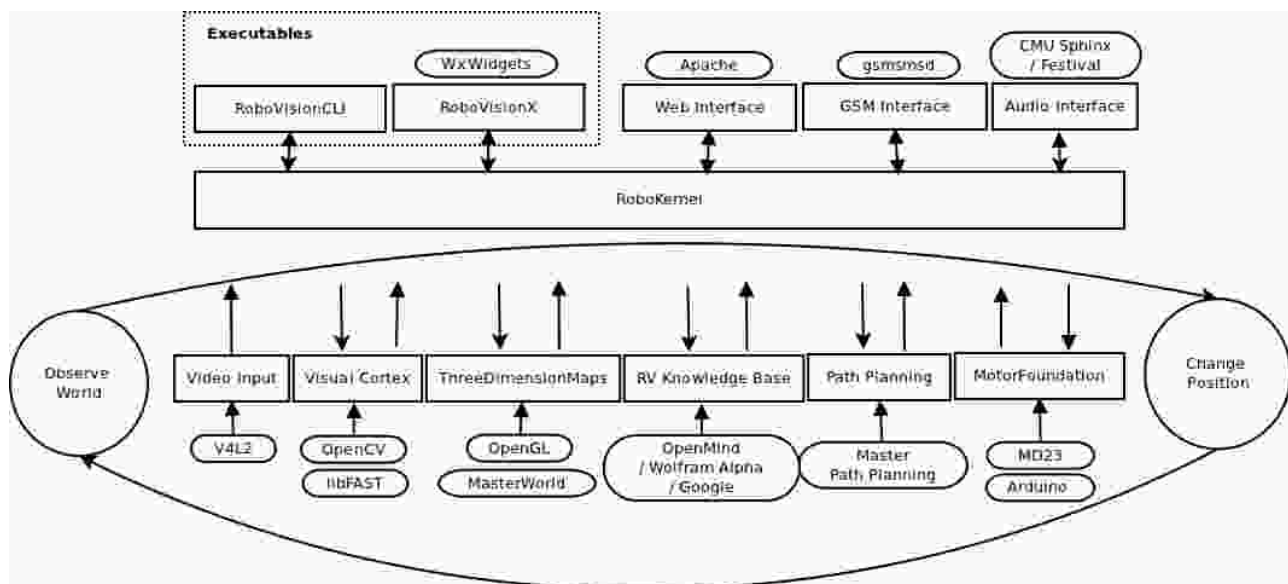
\*

# Software

## 1.1.0 Overview

The software for GuarddoG has gone through 3 major rewrites from scratch. During its development literary all aspects of the source code changed , from the programming language and design pattern of implementation to the Operating System and libraries used. An effort was made firstly to have the best possible performance as far as the architecture could impact it , low complexity and good separation between modules.

In the beginning the whole project was a single executable and all the functionality was embedded in a graphical user interface on a pc running Windows XP. As the software grew larger and at the time of the first transition to the embedded computer onboard GuarddoG the limitations of this approach became apparent. Windows XP was not made for this kind of deployment and using it running on a compact flash memory stick it thrashed it destroying it after approximately 5h of usage ( with all virtual memory settings , file indexing , etc. off ) . The operating system was replaced with Windows XP Embedded and although the issues with the flash memory stick were solved using the File Based Writing Filter which prevented direct reading and writing to the actual device and instead simulated the files in the RAM of the computer , permitting permanent writing at the end of the session by passing the changes to the actual file system on the Compact Flash device and thus greatly conserving read/write cycles. On the disadvantages of the WinXP Embedded OS there were many stability issues ( blue screens ) and driver problems which were difficult to be traced due to the closed source nature of the OS and eventually the decision was made to completely rewrite the daemon for a Linux/\*nix based OS. The graphical user interface was split into 7 major static libraries and on top of them two different executables could be linked , one for “CLI-daemon mode” on the headless environment of the GuarddoG embedded computer , and one with a full GUI based on wxWidgets for the development computer that made debugging easier providing visual tools.



*Illustration 59: A schematic of the software outline , rounded rectangles are external dependencies plain rectangles are libraries implemented as a part of the project*

\*AFTER

The statically linked libraries share a common memory stack and heap and that allows efficient memory reuse without unnecessary “interface” bureaucracy which is a common performance overhead. Some of the libraries have standalone functionality and others with the prime example being libRoboKernel.a are delegation libraries that coordinate the overall task accomplished.

During the Operating System's initialization phase and after run level 5 is reached the Guarddog initialization bash script ( guarddog\_init\_service ) is called and starts to prepare the runtime environment for the main application. A tmpfs , a partition existing entirely in the RAM memory of the computer is created under RoboVisionRuntime/ and the contents of RoboVisionCLI are copied there in order to minimize flash memory wear. Persistent storage is provided by accessing ../DataSets which is the designated directory for storing non volatile data , such as usage statistics , faces , state keeping etc. Permissions are also taken care of since tmpfs typically start of requiring root privileges.

Secondary daemons such as the gsmmsd are kickstarted and after testing if they all work execution is passed to the RoboVisionCLI using the noinput flag since no direct user input will be possible ( the console runs in the background ) . When the RoboVisionCLI executable stops the script resumes killing all possible processes that may remain running and deallocates the tmpfs.

RoboVisionCLI process in turn being an instance without user input basically calls the initialize function of RoboKernel which starts all the sub modules and sleeps until it receives information from the RoboKernel that signals termination.

RoboVisionX is an alternate executable that can be called adhoc without all the tmpfs initializations running from the RoboVisionX directory and it spawns a wxWidgets GUI which provides controls and visual versions of the data handled by the RoboKernel and this is the interface used during the development of the project. The directories for datasets continue to be ../DataSets so this is too a very easy and non-obtrusive design since there is no need for additional lines of code to manage between development directory structure and actual on board usage decreasing the surface area for bugs and easily avoided problems.

A web interface which is provided using PHP and Apache can be used to access the platform using any device with an internet browser. This service is currently protected using a simple password scheme but this is also a very serious security topic since a compromised robot can pose a serious privacy and physical threat for its owners many times worse than a compromised desktop computer. This kind of functionality can be easily deactivated but it is ultimately very useful given a proper security identification system for remote operation using cheap internet enabled mobile devices that are becoming common nowadays.

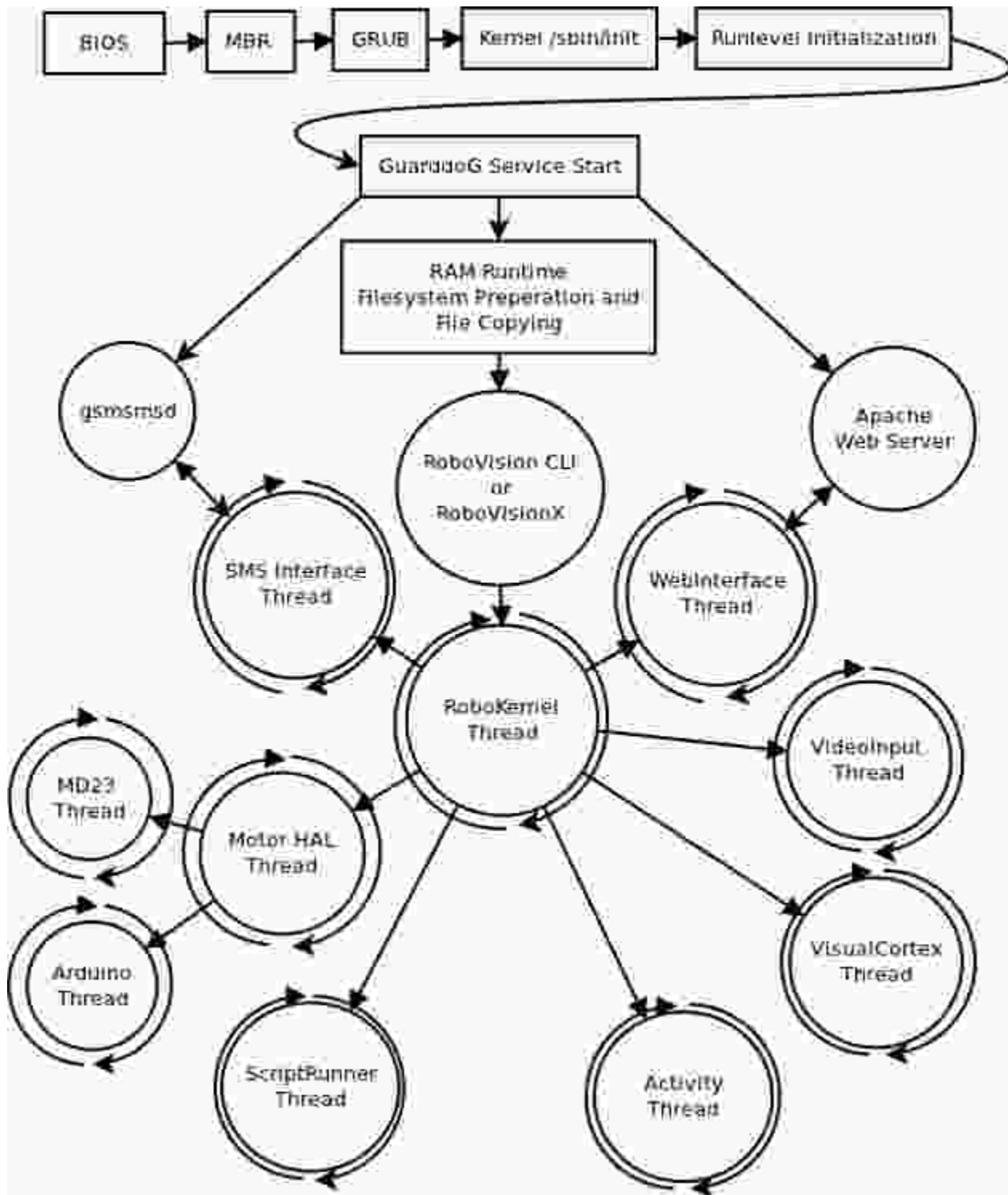
All of the various sub modules have a dedicated thread that “runs” and serves them using the pthreads framework. This threaded design is one of the great architectural strengths of Guarddog since internally there is no IPC used , ( which would imply redundant memory copying ) . I/O operations overlap with CPU work , memory allocation is minimized since the same memory blocks get shared by all the libraries and synchronization overheads are also minimal. Of course in order to keep the design resistant to race conditions on memory access a simple “thread-safety” protocol is maintained since only RoboKernel , the delegation library , can decide when and where output from a library can be inputted in others , so these kinds of problems are centralized in a single library where it is easiest to face them by following the natural pipeline of data from the WebCameras to the motor controllers.

Communication between External Interfaces ( GUI , CLI , HTTP Interface , Speech Recognition , GSM SMS etc ) is performed using a Unified string interface with high level commands such as Move Forward and others explained in more detail in the Unified String Interface section of the Software topic. This also has proved to be a very “human” and economical in its implementation input system

Next comes a process and thread creation diagram that can give a quick look on how the different libraries are organized internally.

\*

## The Process and Thread creation map





\*AFTER

The project has been using git as a version tracking management system since late 2010 , so 2 of the first crucial years of its development are missing from the history-tree. Using the tools at Ohloh.net on the git repository an interesting break down of the project can be extracted.

Language	Lines Of Code	Comments	Comment R.	Blank lines	Total Lines
C	32,588	2,692	7.6%	5,091	40,371
C++	14,067	4,316	23.5%	2,964	21,347
HTML	1,888	19	1.0%	81	1,988
Shell Script	1,432	203	12.4%	464	2,099

Ohloh.net also came up with figures about the development cost of such a platform using the Basic COCOMO model with coefficients  $a=2.4$  and  $b=105$  which is approximately \$656,665 for the resulting 49,134 lines codebase. The projected estimation was 12 person-years which is 3 times larger than the time it took. All these data of course take into account the many years of development , the cumulative number of lines changed , etc. The figures are a little exaggerated but taken into account that half of the work done (before 2010) building this code was not committed it is an accurate estimation.

The code development tree uses git as a version management system. The repository is hosted on github.com which offers on-line web visualizations of the code and data while also featuring bug/issue tracking and a good platform for keeping programmers up to date with e-mails etc.

The project repository is <https://github.com/AmmarkoV/RoboVision>

It can be cloned for read only usage from a remote computer issuing :

```
git clone https://AmmarkoV@github.com/AmmarkoV/RoboVision.git
```

Project dependencies can be automatically downloaded on a debian ( apt-get ) based linux distribution by running `./apt-get-dependencies` and the project can be compiled running the custom bash script `./make` or by opening the workspace file using Code::Blocks IDE.

The choice of an Open Source operating system was one of the most influential for the course of the project , since it made development easier and removed many programming overheads. A good example of this is Text to Speech functionality. In a Windows operating system there are limitations according to the version of the system ( and for no other apparent reason ) Windows XP can use the Microsoft Speech API but only up to version 5.1 , Windows Vista can use it up to SAPI 5.3 and Windows 7 to SAPI 5.4. In a Linux OS ( ubuntu for example ) to make a program “speech-enabled” one can just issue “`sudo apt-get install festival`” to install the most up to date version of the libraries required for text to speech based on work from the University of Edinburgh and then just issue “`system(“echo “Hello World” | festival -tts”);`” .

To make direct usage of the library in Windows one must install Microsoft Visual Studio , download 4GB+ of libraries and the Windows SDK just to get started. Linux is once again a lot easier. These kind of differences are usually quietly ignored , but since it is one of the important lessons learned through the GuarddoG development experience it must be stressed that development for scientific purposes comes naturally in conflict with the usage of closed source profit oriented tools , especially when the international programming community can offer such great alternatives such as Free and Open Source Software .

\*

Working with images means processing very large volumes of data. The limited resources of the computer also mean that the data must be efficiently stored and transported from one library to the other to reduce the overheads of shifting large memory chunks from one place of the memory to another. To combat that and since input sizes don't change size the GuarddoG software tries to be “zero copy” by passing pointers to the data since they are stored on the program's heap and are available to all the different code modules. A good example of this is the camera input which comes at pairs of 320x240 RGB values at a frame rate of 120 frames per second or 2x225KB . V4L2 ( Video 4 Linux 2 ) and the Linux Kernel maps the memory where the frames are received into the executables address space , and from there on the image gets processed in all the ways mentioned in the first part of this document without redundant memory copying trying to conserve CPU cycles.

Parts of the code that have to do with repetition of operations on to the incoming data are hand-optimized by the OpenCV implementation using SIMD ( Single Instruction Multiple Data ) calls that are supported by most of the Intel CPUs as well as recent ARM based CPUs such as the ARM11. Performance statistics are kept using timers between all major calls which can give a very good glimpse of the overall performance of the robot and helped with identifying problematic parts of the code during the development .

The project uses the following third party libraries. The code of GuarddoG ( RoboVision ) has a GPLv3 license. Since the layout of the code uses its own model most of them can be fairly easily removed by different implementations of possibly better performing libraries in the future.

Library	Purpose	License	Dependency
OpenCV	Feature/Face Detection	BSD License	+++
CMU-Sphinx	Speech Recognition	BSD License	+
Festival	Speech To Text	MIT License	+
wxWidgets	GUI / Image Loading	LGPL	+
libPNG	Image Manipulation	libPNG	+
Gnu Scientific Library	Mathematics	GPL	+
Portable Threads	Threading system	LGPL	+++++
libUSB	USB communication	LGPL	+++++
GSM-Utills	GSM communication	-	+
Apache Web Server	Web Interface	Apache	+
OpenGL+FreeGLUT	3D Visualization	-	+++
GnuPlots	Performance Visualization	GPL	+

# Software

## 1.1.0 Libraries Outline

### Video Input ( libVideoInput.a )

#### Purpose :

Image Acquisition from USB webcams using V4L2. Take care of synchronization issues on the two cameras .. Recording/Playback of streams of input frames in order to create test suites that can be re-run without the actual hardware to emulate it for review / benchmarking without additional coding effort on the rest of the project. Future functionality may also include remote video streaming to include new deployment possibilities.

#### Key calls :

**int InitVideoInputs(int X);** Initialize memory for accomodation of X inputs ( x is 2 in guarddog )  
**int InitVideoFeed(int inpt, char \* viddev, int width, int height, int bitdepth, char snapshots\_on, struct VideoFeedSettings videosettings);** Initialize video feed , this is called two times , one for /dev/video0 and one for /dev/video1  
**unsigned int NewFrameAvailable(int webcam\_id);** If this is set it means that a new frame is ready  
**unsigned char \* GetFrame(int webcam\_id);** returns a thread-safe pointer to input for camera X , cameras are typically 0 left and 1 right  
**unsigned int SignalFrameProcessed(int webcam\_id);** This must be called to signal that we are done with the GetFrame call and a new frame can take its place

#### External Dependencies :

V4L2 , pthread , libPNG

#### Output :

2 raw RGB frames that can be sampled directly from the webcams or from an older recording. The output is passed to Visual Cortex for processing

## Visual Cortex ( libVisualCortex.a )

### Purpose :

Visual Cortex's role in the project is self explanatory. It is the library responsible for deriving meaning from the stream of input images. Memory is organized in video registers which are structures that make programming more intuitive. These video registers are allocated in three sizes unsigned char , unsigned short and unsigned int according to the depth of information needed for each of the operations. All of the operations in the library have to do ( more or less ) with transforming a video register and storing the result in the same or a different video register. Precise timers monitor performance on operations , and the way this library works can make it very modular. The library originally contained its own HAAR Classifiers , Corner Detection and Homography Estimation code segments but due to their inferior performance these were easily changed with the OpenCV calls to enhance the result. Disparity Mapping is also an example of this flexibility since both the OpenCV and the Hirschmuller algorithm co-exist and can be dynamically changed as the default Disparity Mapping technique.

Performance statistics are automatically recorded and converted to live updated graphs using gnuplot , and a large array of settings can be modified to tune the different aspects of each algorithm ( comparison window sizes , thresholds , heuristics etc ). Processing happens in three logical steps. The first involves the RoboKernel or another library or executable calling **unsigned int VisCortX\_NewFrame(unsigned int input\_img\_regnum,unsigned int size\_x,unsigned int size\_y,unsigned int depth,unsigned char \* rgbdata);** to point the library to a new raw frame. If image resectioning is activated this frame gets converted to a resectioned equivalent and if not it is retained as it is and checked against the last frame of the webcam. If they are both “the same” or very similar the frame is marked as non-changing and subsequent operations are skipped since they will output the same results and are redundant. When both of the frames are collected , the library is ready to receive requests to extract depth maps , faces , and other useful information from the image pair. By default in every frame processed corner detection and homography estimation is performed to keep accurate position tracking without requiring external repeated commands to do so. Face Detection happens typically in one of the two frames ( depending on movement or “flow” detected on each of the images ) in order to cut by almost half its processing time and depth mapping is performed only when the robot needs depth estimations since it is typically a large amount of CPU work.

There is a lot of side functionality implemented as part of the library and performance has been taken seriously into account , utilizing summed area tables , pointer arithmetic , loop unrolling and other techniques to maximize performance. The only optimizations not implemented were SIMD ( Single Instruction Multiple Data ) operations on the Intel chips since a planned migration to an ARM architecture will take place and OpenCL/CUDA accelerated filters since the current graphics board of the robot does not support them. Speed could be dramatically increased ( 4x – 32x ) in some cases by using these technologies. Despite of the difficulties when implementing them the source code layout should provide a good framework to unobtrusively implement them with minimal cost to the rest of the design.

## Key calls :

**unsigned int VisCortx\_Start(unsigned int res\_x,unsigned int res\_y);** This is the initialization call for the library which sets up registers , precalculations and allocates state variables according to the image resolution.

**unsigned int VisCortx\_Stop();** This is the closing call for the library

**unsigned int VisCortx\_SetCamerasGeometry(float distance\_between\_cameras,float diagonal\_field\_of\_view,float horizontal\_field\_of\_view,float vertical\_field\_of\_view);** This call sets some camera information based on the manufacturer specifications.

**void VisCortx\_CameraParameters(int right\_cam,double fx,double fy,double cx,double cy,double k1,double k2,double p1,double p2,double k3);** This call sets camera information based on calibration for the specific cameras onboard the robot ( Since every camera has a unique assembly with slight differences ).

**unsigned int VisCortX\_NewFrame(unsigned int input\_img\_regnum,unsigned int size\_x,unsigned int size\_y,unsigned int depth,unsigned char \* rgbdata);** This call passes the frames received by VideoInput into the library.

**void ExecutePipeline();** Signals that both images have been passed and that processing may commence.

**void VisCortx\_FullDepthMap(unsigned int max\_milliseconds);** This call starts depth mapping with a timeout value for blocking the thread that called the depth mapping

**unsigned char \* VisCortx\_ReadFromVideoRegister(unsigned int reg\_num,unsigned int size\_x,unsigned int size\_y,unsigned int depth);** This returns a pointer to a video Register , i.e. the depth map register.

**unsigned int VisCortx\_RecognizeFaces(unsigned int cam);** Extracts a list of faces.

**unsigned int VisCortx\_GetFaces(unsigned int vid\_reg,unsigned int point\_num,unsigned int data\_type);**  
Gets a list of face coordinates

**int UpdateCameraPose(unsigned int reg\_num)** automatically updates position values on MasterWorld ( another library ) so it is not called externally

## External Dependencies :

OpenCV , GSL , libFAST , gnuplot

## Output :

A list of faces detected on each of the input images , a depth map “3d point cloud” using information extracted by means of disparity mapping , a list of feature or salient points , and an array containing the elements of the transformation that took place between the current and the last pair of frames

# World Mapping

## MasterPathPlanning ( libMasterPathPlanning.a )

## MasterWorld ( libMasterWorld.a )

### Purpose :

Mapping services , position tracking , path planning , SLAM , obstacle avoidance and general movement planning. The functionality is split in two libraries one called MasterPathPlanning which offers a two dimensional world abstraction , along with scale information and an agent abstraction in order to be able to represent moving persons ( faces ) detected and receive high level commands such as “plan a route to person X” or location names for commands like “plan a route to kitchen room”. PathPlanning also gets notified of changes on the position of the robot through MasterWorld that may produce a new path plan towards the target .

None of the two libraries actually executes movement , path planning plans a route and master world transfers to and from the planning logical module to the Hardware Abstraction Layer (HAL) presented next. MasterWorld does not yet perform internal SLAM ( only dead reckoning ) but the functionality will have to be appended later.

### Key calls :

```
struct Map * CreateMap(unsigned int world_size_x,unsigned int world_size_y,unsigned int
actors_number);
struct Map * LoadMap(char * filename) ;
int SaveMap(struct Map * themap) ;
int SetMapUnit_In_cm(struct Map * themap,unsigned int cm_per_unit) ;
int ObstacleExists(struct Map * themap,unsigned int x,unsigned int y) ;
int ObstacleRidiousExists(struct Map * themap,unsigned int x,unsigned int y);
int SetObstacle(struct Map * themap,unsigned int x,unsigned int y,unsigned int safety_ridious);
int RemoveObstacle(struct Map * themap,unsigned int x,unsigned int y,unsigned int safety_ridious);
int SetAgentHeading(struct Map * themap,unsigned int agentnum,float heading) ;
int SetAgentLocation(struct Map * themap,unsigned int agentnum,unsigned int x,unsigned int y) ;
int SetAgentTargetLocation(struct Map * themap,unsigned int agentnum,unsigned int x,unsigned int y) ;
int MoveAgentForward(struct Map * themap,unsigned int agent,float leftwheel_cm,float rightwheel_cm) ;
int AddObstacleViewedByAgent(struct Map * themap,unsigned int agent,float horizontal_angle,float
vertical_angle,float distance_in_cm);
int FindPath(struct Map * themap,unsigned int agentnum,unsigned int timeout_ms) ;
int ExtractRouteToLogo(struct Map * themap,struct Path * thepath,char * filename) ;
int ExtractMaptoHTML(struct Map * themap,char * filename,unsigned int map_size);
int GetRoutePoints(struct Map * themap,struct Path * thepath) ;
int GetRouteWaypoint(struct Map * themap,unsigned int agent,unsigned int num,unsigned int
*x,unsigned int *y) ;
int GetStraightRouteWaypoint(struct Map * themap,unsigned int agentnum,unsigned int count,unsigned
int *x,unsigned int *y);
```

### Output :

These 2 libraries provide access to a virtual environment when according to the robot sense of depth acceleration , movement etc a virtual “simulation” can be run and re run that can produce a list of commands that in turn will drive the robot towards its objectives.

\*

# **Motor Foundation**

MotorHAL ( libMotorHAL.a )  
Arduino (libRoboVisionSensorLib.a)  
MD23 (libMD23.a)

## **Purpose :**

An abstraction layer that can maximize code reuse , despite the constant changes in GuarddoG hardware. Historically this consisted only of a Mindstorm Library which was later replaced by an MD23 and recently an additional arduino for ultrasonic sensors , accelerometers and other low level peripherals. These can be changed without impacting the rest of the codebase with substantially less effort and upkeep cost.

## **Key calls :**

```
unsigned int RobotInit(char * md23_device_id,char * arduino_device_id);
unsigned int RobotClose();
void RobotWait(unsigned int msec);
unsigned int RobotMoveJoystick(signed int joy_x,signed int joy_y);
unsigned int RobotRotate(unsigned char power,signed int degrees);
unsigned int RobotStartRotating(unsigned char power,signed int direction);
unsigned int RobotSetHeadNod(char * pose_string);
unsigned int RobotSetHeadPose(unsigned int heading,unsigned int pitch);
unsigned int RobotGetHeadPose(unsigned int * heading,unsigned int * pitch);
unsigned int RobotMove(unsigned char power,signed int distance);
unsigned int RobotStartMoving(unsigned char power,signed int direction);
unsigned int RobotManoeuvresPending();
void RobotStopMovement();

int RobotGetUltrasonic(unsigned int which_one);
int RobotGetAccelerometerX();
int RobotGetAccelerometerY();
int RobotSetLightsState(unsigned int light_num,unsigned int light_state);
int RobotIRTransmit(char * code,unsigned int code_size);
```

## **External Dependencies :**

libUSB , pthreads

## **Output :**

The MotorHAL library and the 2 sub libraries that communicate with the motors and arduino peripherals are the gateway of the software to the real world.

## **RoboKernel** ( libRoboKernel.a )

### **Purpose :**

This is the core program daemon linked to from each of the the executables of GuarddoG. Its job is to start all of the Input interfaces ( IRC , GSM , Joystick , Scripts , Web Interface ) and sub libraries and pass data between modules. Executables use the IssueCommand to operate it and retrieve state information directly from the submodules.

### **Key calls :**

```
int IssueCommand(char * cmd,char * res,unsigned int resmaxsize,char * from);
```

```
unsigned char * GetVideoRegister(unsigned int num);  
unsigned int GetCortexSetting(unsigned int option);  
void SetCortexSetting(unsigned int option,unsigned int value);  
unsigned int GetCortexMetric(unsigned int option);
```

```
int StartRoboKernel();  
int StopRoboKernel();  
int CheckThatRoboKernelStopped();
```

```
int RoboKernelAlive();
```

```
struct Map * GetWorldHandler();
```

### **External Dependencies :**

pthread

### **Output :**

Robot functionality

### **Some of the smaller or incomplete libraries part of the project :**

They are not analyzed for brevity  
Auditory Input , Input Parser , IrcInterface ,RVKnowledge Base

\*



## Software

### 1.1.0 Unified String Interface

In the process of reducing the surface area for the different libraries and their in between communication , a mini scripting high-level language was developed ( for now it only features functional usability , no arithmetic operations , loops or other operators ). This language serves as a layer between the different libraries to make debugging easier and make the code more easily portable as parts of the design changed. For instance joystick input was in the beginning handled using the Win32API , later on the design its code was replaced with a wxWidgets wxJoystick implementation and finally by linux specific code which utilized directly the device file `/dev/input/js0` . During all the transitions joystick events were passed using `JOYSTICK INPUT(x_axis,y_axis)`. What is more , a user can in fact use his cell phone to send an SMS to the robot GSM stick `JOYSTICK INPUT(x_axis,y_axis)` or using an IRC relay , or via the web interface and command the robot just as he would in a local setup. This also makes the Executable program very agile and the computational overhead is minuscule compared to the benefits of this design. Adding a working knowledge base to this interface will make the robot much more “intelligent” since all inputs will share a common language and complex queries using forward chain reasoning and other techniques. On an even more advanced level direct object transformation into strings , without explicit orders from the robot's owner will make it more human-like and intelligent although on a much lower level than human intelligence.

Command	Details	Command	Details
DANGER	Signal Alarm	AUTO CALIBRATE	Dynamic camera calibration
SAFE	Stop Alarm Function	DEPTH MAP	Perform depth map
MOTION ALARM	Signal Alarm on Motion	HEAD POSE	Change the pose of the head
PANORAMIC	Take a panoramic image set	SET LIGHT	Activate / Deactivate lights
SWAP FEEDS	Swap Camera Inputs ( L/R )	GOTO	Move to new position
WEB INTERFACE	Enable/Disable Web Interface	FORWARD	Move Forward
DRAW MOVEMENT	Highlight movement on frame	BACKWARD	Move Backward
DRAW FEATURES	Highlight Corners on frame	LEFT	Rotate Left
DRAW CALIBRATED	Draw Calibrated input frames	RIGHT	Rotate Right
FIND FEATURES	Find Features on input frames	TOGGLE AUTO RECORD SNAPSHOTS	
CLEAR FEATURES	Clear current feature list	TOGGLE AUTO PLAYBACK SNAPSHOTS	
PLAYSOUND	Plays a wav/ogg/etc file	RECORD SNAPSHOT	Record images

Command	Details	Command	Details
RECORDSOUND	Record an wav file	RECORD COMPRESSED	Record images
STOP SOUNDS	Kill all prcesses playing sounds	PLAYBACK SNAPSHOT	Playback images
SAY	Text To Speech echo string	PLAYBACK LIVE	Revert to live stream
DEPTH MAP TO FILE	Save Depth map to file	SENSORS	Retrieve Sensor Value
REFRESH MAP	Refresh map with new points	SCRIPT	Run Script
DEPTH MAP IMPORT TO MAP	Refresh map with new points	STOP SCRIPT	Stop Script
SOBEL DERIVATIVE	Apply Sobel Convolution filter	HYPERVISOR STATISTICS	Retrieve Statistics
CONVOLUTION FILTER	Apply Convolution filter	TELL	Tell string to KB
FACE DETECTION	Detects Faces on input frames	ASK	Ask string from KB
REMEMBER IMAGE	Remember current image part	SEARCH	Search KB
IDENTIFY IMAGE	Identify current image part	JOYSTICK INPUT	Simulate Joystick Movement
DELAY	Delay next command	FUNDAMENTAL MATRIX	Calculate fundamental mat.
AUTONOMOUS MODE	Start Autonomous mode	-	-

<b>Communication Interfaces Available that were Unified</b>
GUI
CLI
External call to guarddog script
Web interface
IRC interface
GSM SMS Interface
Speech Interface via CMU-Sphinx

## **Future Work**

### **1.1.0 The list of future things**

***CAD designed body***

***\*Low Level Assembly ( MMX/SSE3 ) optimizations***

***CUDA / VLSI acceleration***

***\*Network Connectivity – Encryption over RF***

***\*NLP – AI Knowledge Base***

***\*Face / Object / Speech Recognition***

***\*Physics Simulation***

***Completion***

***\*Commercial Personal Robots***

***\*Car sized guarddog or "CardoG"***

**Hansei** (反省<sup>?</sup>, "self-reflection") is a central idea in [Japanese](#) culture. Its meaning is to acknowledge your own mistake and to pledge improvement. This is similar to the German proverb *Selbsterkenntnis ist der erste Schritt zur Besserung* where the closest translation would be "Self-awareness is the first step to improvement".

# Acknowledgements

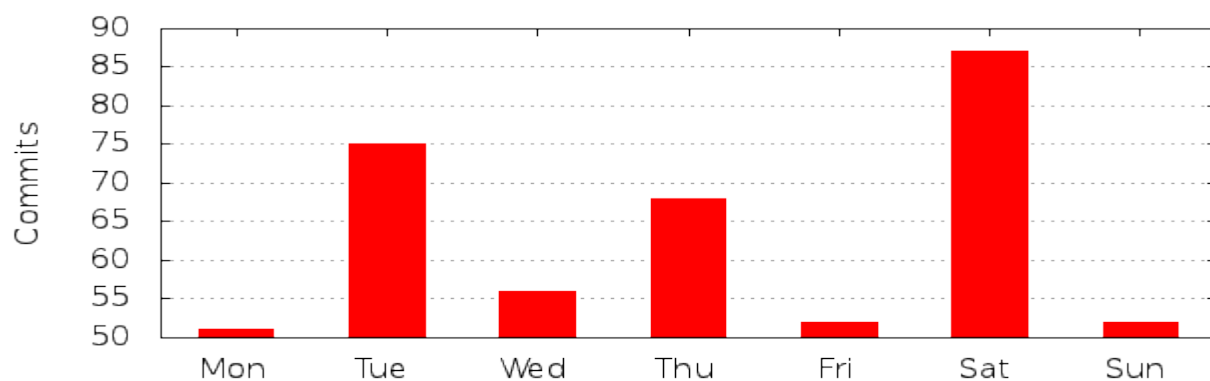
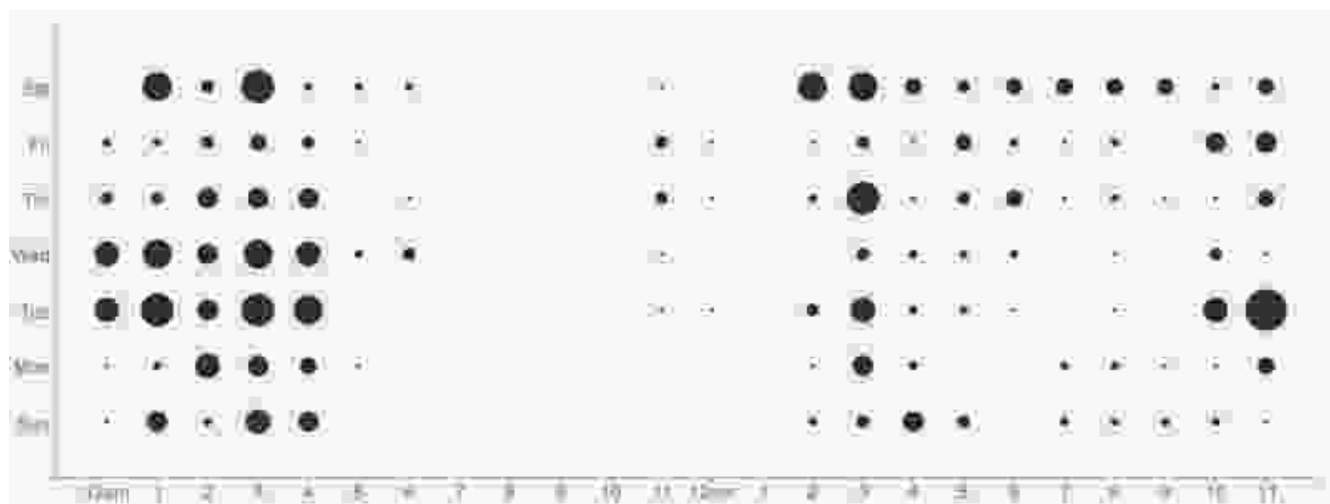
## 1.1.0 Statistics

### ***Acknowledgements***

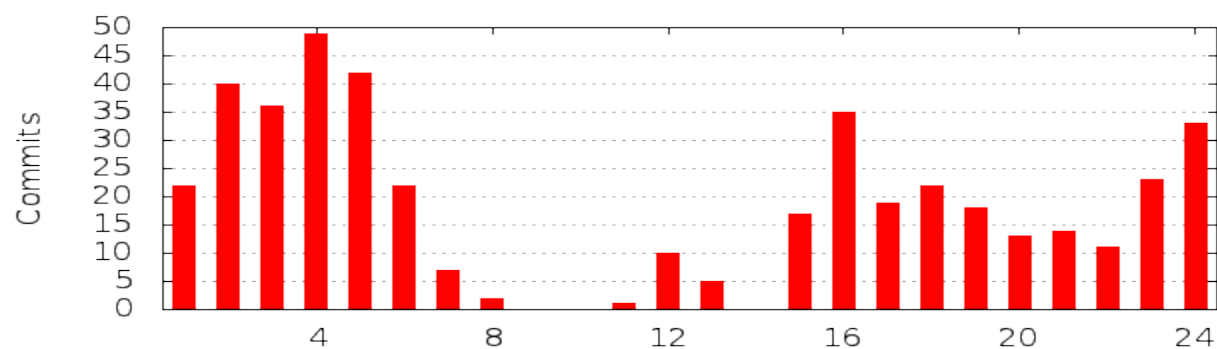
***My parents :) , foss programmers everywhere , mr gepap for lending me the book Multiple Video Geometry for 4 years***

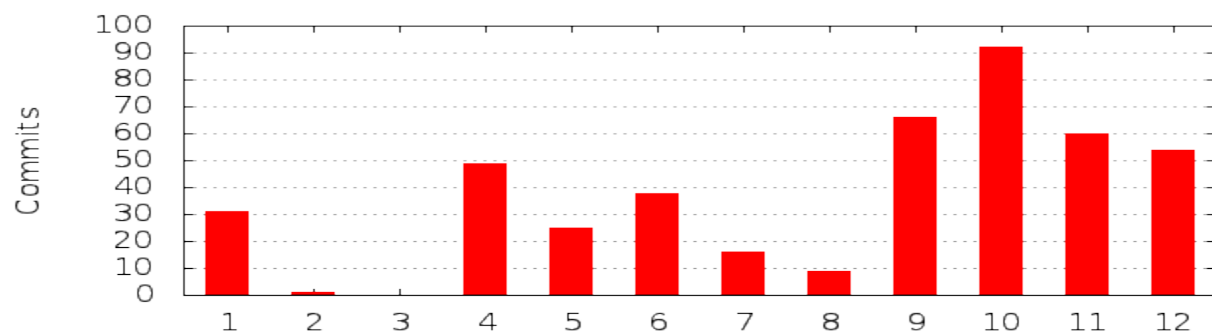
***\*GNU/Linux OpenCV Git / Github***

Commit Statistics



\*





\*

## **Bibliography / References**

[1]

[2]

[3] A Flexible New Technique for Camera Calibration (1998) by Zhengyou Zhang , Zhengyou Zhang

[4] Edward Rosten , Fusing points and lines for high performance tracking , (**YEAR HERE**) Machine learning for high-speed corner detection.

[5] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, (**YEAR HERE**) SURF: Speeded Up Robust Features

[6] Jianbo Shi , Carlo Tomasi , ( YEAR HERE ) Good Features to Track

[7] Yoav Freund, Robert E. Schapire. "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting", 1995