

VISUAL-BASED GUIDANCE FOR AUTONOMOUS VEHICLES

A PROJECT REPORT

submitted by

SESHAN PS (118004159)

towards partial fulfilment of the requirements for the award of the degree

of

**Bachelor of Technology in
Electronics & Communication Engineering**



School of Electrical & Electronics Engineering

SASTRA DEEMED TO BE UNIVERSITY

(A University established under section 3 of the UGC Act, 1956)

Tirumalaisamudram

Thanjavur-613 401

July 2018

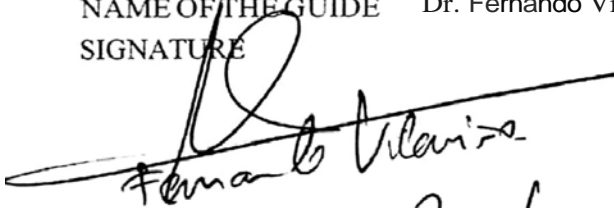
BONAFIDE CERTIFICATE

This is to certify that the project work entitled "V SUAL-BASED GUIDANCE FOR
AUIONOMOUS VEIDCLFS" is a bonafide record of the work carried out by

SESHAN P S (118004159)

student of final year B.Tech., Electronics and Communication Engineering, under my guidance during the period February - June, 2018, in partial fulfillment of the requirements for the award of the degree of B.Tech in Electronics & Communication Engineering of the SASTRA DEEMED TO BE UNIVERSITY, Thimmalaisamudram, Thanjavur - 613401, during the year 2014-2018.

NAME OF THE GUIDE Dr. Fernando Vilarino
SIGNATURE



Project Viva-Voce held on

Barcelona 13th June

Examiner -I

Examiner-II

MENTOR'S CERTIFICATE

This is to certify that the project work entitled "VISUAL-BASED GUIDANCE FOR AUTONOMOUS VEHICLES" has been carried out by

SESHAN P S (118004159)

Student of final year B.Tech., Electronics and Communication Engineering under my guidance during the period February - June, 2018, submitted to SASTRA DEEMED TO BE UNIVERSITY, Thirumalaisamudram, Thanjavur- 613401, in partial fulfillment of the requirements of the award of the degree of B.Tech in Electronics and Communication Engineering during the period 2014/2018.

Seshan has developed a study about autonomous driving. He has worked both the hardware and the software parts. For the hardware parts he has investigated a car model including differential gear, 3-phase motor and controller and both Raspberry Pi and Arduino. The software part has been integrated in a mini-PC running Linux. He has used a stereo-pair camera to get a disparity map, which could be integrated for a depth map. He has programed in OpenCV and a number of libraries for computer vision. His final report includes a detailed explanation of the work.

Barcelona, June 14, 2018.

**Fernando
Vilariño**

Digitally signed by Fernando
Vilariño
DN: cn=Fernando Vilariño, o,
ou=Computer Vision Center UAB,
email=fernando@cvc.uab.es, c=ES
Date: 2018.06.14 12:55:09 +01'00'

Prof. Fernando Vilariño
Associate Director and Associate Professor at CVC - UAB

ACKNOWLEDGEMENTS

First of all, I express my gratitude to **Prof. R. Sethuraman**, Vice Chancellor, SASTRA Deemed to be University who has provided all necessary facilities and encouragement during the course of study. I extend my sincere thanks to **Dr. G. Bhalachandran**, Registrar, SASTRA Deemed to be University for providing me the opportunity to pursue this project.

I also extend my genuine gratitude to **Dr. S. Vaidhyasubramaniam**, Dean, Planning and Development, SASTRA Deemed to be University for his constant moral support to pursue my project at Universitat Autònoma de Barcelona (UAB), Spain, under the Semester Abroad Programme (SAP). I also thank **Dr. M. Sridharan**, Professor and **Mr. M. Raja Subramanian**, Dept. of Training, Placement & International Relations for guiding me throughout the project tenure.

I express my heartfelt indebtedness to **Dr. K. Thenmozhi**, Associate Dean (ECE), and **Dr. John Bosco Balaguru**, Associate Dean (Research) who motivated me during the project work.

I owe my deepest gratitude to my guide **Prof. Fernando Vilariño**, Associate Director and Associate Professor at the Computer Vision Centre (CVC), UAB for all his suggestions and advices, and for providing more support than I could ask for. It has been a tremendous learning experience for me.

I express special gratitude to my final year project coordinators, **Dr. V. Muthubalan**, **Dr. R. Ramesh** and **Dr. B. G. Jeyaprakash**, SASTRA Deemed to be University whose contribution in simulating suggestions and encouragement helped me to coordinate my project. We wish to thank all the teaching and non-teaching staff members of department of Electronics and Communication Engineering of SEEE for their kind support and cooperation.

ABSTRACT

Keywords: Computer Vision, Self-driving car, Stereo camera

Autonomous cars or Self driving cars have already gained a lot of momentum currently that they are no longer a thing of the future. With Tesla already working on commercialising the car, everyone is aware of where the future of transportation is headed. This project was taken to understand the mechanism/technology going under the hood of the cars. The project will largely focus on the computer vision part of the system, and if possible, the vision part will be coupled and tested on the hardware i.e., a small-scale RC car ($1/10^{\text{th}}$ of a real sized car). A DUO3D stereo camera will be used as the main sensor for the Vision system. The frames taken from the stereo camera will be processed using OpenCV, for normal images, disparity maps, and 3D reconstruction of the image. The control signals can then be sent to an Arduino using serial communication which will control the actuators of the car.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
ABSTRACT	vi
LIST OF TABLES	x
LIST OF FIGURES	xi
CHAPTER 1	1
INTRODUCTION.....	1
CHAPTER 2	2
LITERATURE REVIEW.....	2
2.1 DUO3D STEREO CAMERA.....	2
2.2 CMAKE.....	3
2.3 OpenCV	3
2.4 POINT CLOUD LIBRARY	4
2.5 BOOST	5
2.6 ARDUINO UNO AND THE CORRESPONDING IDE.....	5
CHAPTER 3	7
EXPERIMENTAL PROCEDURES.....	7

3.1 SOFTWARE REQUIREMENTS	7
3.2 HARDWARE REQUIREMENTS	14
3.3 DESIGN AND IMPLEMENTATION	18
3.3.1 Camera	19
3.3.1.1 Projective geometry	20
3.3.1.2 From world coordinates to camera coordinates.....	21
3.3.1.3 Pixel Aspect Ratio	22
3.3.1.4 1.4 Optical Center	23
3.3.1.5 Homogeneous coordinates.....	24
3.3.1.6 Distortion coefficients	25
3.3.1.7 1.7 Stereo Triangulation.....	30
3.3.1.8 Camera Calibration.....	34
3.3.1.9 1.9 StereoCam Class	37
3.3.2 Data Analysis	38
3.3.2.1 Visual Odometry	38
3.3.2.2 2.2 Key Point Detection	39
3.3.2.3 Pose change estimation	41

3.3.2.4 2.4 Obstacle detection	42
3.3.3 Path Planning.....	43
3.3.3.1 Global route planning.....	44
3.3.3.2 Avoidance Path Planning	47
CHAPTER 4	48
RESULTS AND DISCUSSION	48
4.1 STEREO CAMERA	48
4.1.1 Calibration	49
4.1.2 Avoidance Path Planning	53
REFERENCES	57
3.3.2.4 SOFTWARE REQUIREMENTS	

LIST OF TABLES

No.	Table Name	Page Number
3.1	RC Car Technical details	27

LIST OF FIGURES

No.	Figure Name	Page Number
2.1	Arduino Uno Specifications	6
3.1	RC Car	14
3.2	Ackermann Steering	14
3.3	Electronic Speed Controller LANSU LS4040-D	17
3.4	DUO3D Stereo Camera	20
3.5	Thales Theorem	22
3.6	Optical center of a camera	23
3.7	Homogeneous coordinates	24
3.8	Distortion due to lens	26
3.9	Types of distortions	27
3.10	Radial distortion	28
3.11	Original and corrected location of pixels	29
3.12	Undistorted images	29
3.13	Stereo Triangulation Case 1	31
3.14	Stereo Triangulation Case 2	31
3.15	Detection of matching pixels	32
3.16	Pixels when the cameras are horizontally aligned	33
3.17	Checkerboard with size (9,6)	35
3.18	Calibration	36
3.19	FAST (Features from accelerated segmented tests)	39
3.20	BRIEF's random sampling pairs	40

3.21	Obstacle map simulator with 20 cm square	43
3.22	Custom obstacle map with 5 cm square	43
3.23	Breadth first search path reconstruction	46
3.24	Dijkstra	47
3.25	Best first search	47
3.26	A*	47
4.1	Uncalibrated Left camera image	48
4.2	Uncalibrated Right camera image	49
4.3	Horizontally merged images	49
4.4	Uncalibrated Checkerboard	50
4.5	Calibrated Checkerboard with horizontal lines for visualizing	50
4.6	Disparity map	51
4.7	Dense 3D map	52
4.8	Dense 3D map of hand	53
4.9	Brute force match	53
4.10	Avoidance curve path (artwork)	54
4.11	Avoidance curve path (artwork)	54
4.12	Avoidance Simulator	55

CHAPTER 1

INTRODUCTION

This project focuses on the computer vision part of autonomous navigation using just a single stereo camera, similar to how humans perceive visual information.

A stereo camera is basically a camera with two lenses, and hence the obtained images can be processed just like the human vision does.

The first question we had to answer was why computer vision and why not other sensors such as radar, lidar etc., for autonomous guidance.

The first point that differentiates camera and other sensors is the cost. A LIDAR costs anywhere from 20000€ to 70000€ On the other hand, a camera is easily available from 20€ to 400€

Another reason was the possibility of using it for other operations simultaneously. They can be used to reconstruct the scene in 3D, detect traffic signals, lane lines, pedestrians etc. Moreover, Lasers and radars only provide the spatial data, without colours, and radars are also prone to present interferences in some environment.

Thirdly, since the popularity of cameras is huge, it is easier to find online support, with the operations being performed. There are many code libraries for controlling cameras and performing operations.

In this particular project, we use OpenCV. OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision. It is developed by various researchers, is cross-platform, and is free to use under the open source BSD license. This project was adopted from an existing project done by Alejandro Daniel Noel[8].

CHAPTER 2

LITERATURE REVIEW

The Literature review is divided into the following components that were required:

1. DUO3D Stereo Camera
2. CMake
3. OpenCV
4. Point Cloud Library
5. Boost
6. Arduino UNO and corresponding IDE

2.1 DUO3D STEREO CAMERA

A duo3d stereo camera is the main component of the whole vision system. It can be accessed by its own executable application such as DUO Dashboard, DUO Calibration, etc., but can also be accessed by codes in C++, OpenCV etc.,

The header DUOLib.h provides the necessary functions to access the camera. The main advantage of DUO3D stereo camera is that both the cameras are electronically connected, and hence are synchronised. Which means there is no delay between the frames captured by the left and right cameras. This makes reconstruction of the images easier. Other stereo cameras usually have a delay of about 20-30 ms between the frames, and they'll have to be synchronised before being used.

2.2 CMAKE

CMake is an extensible, open-source system that manages the build process in an operating system and in a compiler-independent manner. Unlike many cross-platform systems, CMake is designed to be used in conjunction with the native build environment. Simple configuration files placed in each source directory (called CMakeLists.txt files) are used to generate standard build files (e.g., makefiles on Unix and projects/workspaces in Windows MSVC) which are used in the usual way. CMake can generate a native build environment that will compile source code, create libraries, generate wrappers and build executables in arbitrary combinations. CMake supports in-place and out-of-place builds and can therefore support multiple builds from a single source tree. CMake also supports static and dynamic library builds. Another nice feature of CMake is that it generates a cache file that is designed to be used with a graphical editor. For example, when CMake runs, it locates include files, libraries, and executables, and may encounter optional build directives. This information is gathered into the cache, which may be changed by the user prior to the generation of the native build files.

CMake is used for compiling and building the project. There are various different functionalities implemented in different parts of the project and CMake helps in linking all the files and creates an executable file.

2.3 OpenCV

One of the Computer Vision parts of the project uses OpenCV for its processing.

DUO3D camera can be accessed by Opencv codes (in C++), and the basic 2D frame capture, display, flip etc., were performed using OpenCV functions.

To achieve this, we used the sample programs for OpenCV given in the website duo3d.com.

This example was understandable, and it helped us build the program further as per our requirement.

2.4 POINT CLOUD LIBRARY

The Point Cloud Library (PCL) is an open-source library of algorithms for point cloud processing tasks and 3D geometry processing, such as occur in three-dimensional computer vision. The library contains algorithms for feature estimation, surface reconstruction, 3D registration, model fitting, and segmentation. It is written in C++ and released under the BSD license.

These algorithms have been used, for example, for perception in robotics to filter outliers from noisy data, stitch 3D point clouds together, segment relevant parts of a scene, extract keypoints and compute descriptors to recognize objects in the world based on their geometric appearance and create surfaces from point clouds and visualize them.

Here we have used Point Cloud Libraries for the reconstruction of 3D images, construction and normalization of Disparity maps, and Point Cloud Visualizer etc.

2.5 BOOST

Boost is a set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. It contains over eighty individual libraries.

It can be added as a header file in any program and can be used. In this project, Boost has been used for some image processing applications, and also for multithreading, to improve the speed of the operations, and to use the available resources effectively.

2.6 ARDUINO UNO AND THE CORRESPONDING IDE

The Arduino or the Genuine Uno board is a micro controller board which is based on the ATmega328P microcontroller. The Uno is a USB board from Arduino and the first of its line. The biggest advantage of the Uno is that we need not use a hardware programmer. This means that uploading code onto the micro controller is just the matter of clicking a button on the screen. This is done through the Arduino Software that is available to download. This IDE provides a fantastic interface to write, modify and upload code onto the board. The following are the technical specifications as provided by Arduino.

Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

Fig 2.1 Arduino Uno Specifications

The Arduino Uno board is used to drive the motors in the required speed and direction, i.e., the control signals are sent via the Arduino.

CHAPTER 3

EXPERIMENTAL PROCEDURES

3.1 SOFTWARE REQUIREMENTS

3.1.1 CMake

Since we have used OpenCV in C++ for the whole project, we would have to install CMake as a prerequisite. It was done using terminal commands on the Ubuntu machine. The whole project was done on a Ubuntu 14.04 machine.

To install, first download and extract the version of CMake you want from the official CMake Webpage <https://cmake.org/download/>

Once you have extracted the file, update the version and build in the following commands to get the desired version.

```
version=3.11
```

```
build=2
```

```
mkdir ~/temp
```

```
cd ~/temp
```

```
wget https://cmake.org/files/v\$version/cmake-\$version.\$build.tar.gz
```

```
tar -xzyf cmake-$version.$build.tar.gz
```

```
cd cmake-$version.$build/
```

Now install the extracted source by running:

```
./bootstrap
```

```
make -j4
```

```
sudo make install
```

Once you are done installing, you can test the CMake version using the command

```
cmake --version
```

This command should return the version of cmake that is installed.

For example,

```
cmake version 3.11.X
```

3.1.2 OpenCV

Now that we have installed cmake on our machine, we can move on to installing OpenCV.

To install OpenCV on Ubuntu 14.04, type the following commands in the terminal:

Since OpenCV is largely used using python, it is recommended that you install python as well.

```
sudo apt-get install python3.5-dev python3-numpy libtbb2 libtbb-dev
```

```
sudo apt-get install libjpeg-dev libpng-dev libtiff5-dev libjasper-dev libdc1394-22-dev  
libeigen3-dev libtheora-dev libvorbis-dev libxvidcore-dev libx264-dev sphinx-common libtbb-  
dev yasm libfaac-dev libopencore-amrnb-dev libopencore-amrwb-dev libopenexr-dev  
libgstreamer-plugins-base1.0-dev libavutil-dev libavfilter-dev libavresample-dev
```

Now, we have installed all the dependencies required.

Next we get OpenCV from the sources on github.

```
$ sudo -s  
$ cd /opt  
/opt$ git clone https://github.com/Itseez/opencv.git  
/opt$ git clone https://github.com/Itseez/opencv\_contrib.git
```

Now to build and install OpenCV, we use the following commands:

```
/opt$ cd opencv  
  
/opt/opencv$ mkdir release  
  
/opt/opencv$ cd release  
  
/opt/opencv/release$ cmake -D BUILD_TIFF=ON -D WITH_CUDA=OFF -D
```

```
ENABLE_AVX=OFF -D WITH_OPENGL=OFF -D WITH_OPENCL=OFF -D  
WITH_IPP=OFF -D WITH_TBB=ON -D BUILD_TBB=ON -D WITH_EIGEN=OFF -D  
WITH_V4L=OFF -D WITH_VTK=OFF -D BUILD_TESTS=OFF -D  
BUILD_PERF_TESTS=OFF -D CMAKE_BUILD_TYPE=RELEASE -D  
CMAKE_INSTALL_PREFIX=/usr/local -D  
OPENCV_EXTRA_MODULES_PATH=/opt/opencv_contrib/modules
```

```
/opt/opencv/
```

```
/opt/opencv/release$ make -j4
```

```
/opt/opencv/release$ make install
```

```
/opt/opencv/release$ ldconfig
```

```
/opt/opencv/release$ exit
```

```
/opt/opencv/release$ cd ~
```

3.1.3 Point Cloud Library

It is fairly easy to install PCL 1.7.2 on Ubuntu:

```
sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
```

```
sudo apt-get update
```

```
sudo apt-get install libpcl-all
```

3.1.4 Boost

To install boost we use apt-get on the ubuntu terminal:

```
sudo apt-get install libboost-all-dev
```

```
sudo apt-get install aptitude
```

```
aptitude search boost
```

3.1.5 DUO3D Stereo Camera

To install the DUO3D Stereo Camera, you must have purchased a DUO3D Camera from the official website. Once you've purchased the camera you would need to login to the account and download the corresponding driver for linux, and then install as follows:

Assuming we have downloaded and extracted the package to ~/Documents/CL-DUO3D-LIN-1.1.0.30/

we now install the required tools

```
sudo apt-get install build-essential qt5-default git make
```

We must now build and load the DUO Kernel Module (duo.ko) for our version of Linux kernel. Go into the ~/Documents/CL-DUO3D-LIN-1.1.0.30/DUODriver/ folder and type:

```
chmod u+x duodriver.run
```

```
./duodriver.run
```

We can now load the module using

```
sudo insmod duo-1024.ko
```

Note: On some systems, since this driver tells DUO to use 1024 byte bulk packets (if DUO is connected to USB 3.0 port), this command results in USB packets being truncated by Linux kernel to 512 bytes resulting in corrupted image. If you see the packet loss in DUO Dashboard on your machine, please unload the driver and use the following:

```
sudo insmod duo-512.ko
```

If there is a need to unload the driver, use the following


```
sudo rmmod -f duo
```

Once we have determined that duo-1024 works on the system, we can go ahead and install it using:

```
sudo cp duo-1024.ko /lib/modules/$(uname -r)/kernel/drivers/duo.ko
```

```
echo 'duo' | sudo tee -a /etc/modules > /dev/null
```

```
sudo depmod
```

Or if duo-1024 does not work, to install the driver use the following commands:

```
sudo cp duo-512.ko /lib/modules/$(uname -r)/kernel/drivers/duo.ko
```

```
echo 'duo' | sudo tee -a /etc/modules > /dev/null
```

```
sudo depmod
```

After successful installation, we connect the duo to the system.

To verify that the device is connected to the system you can view the node duo0 appears in /dev directory on the file system.

3.2 HARDWARE REQUIREMENTS

The project involves testing/application on a car, and hence we would require a fully functional 1/10 scaled car.

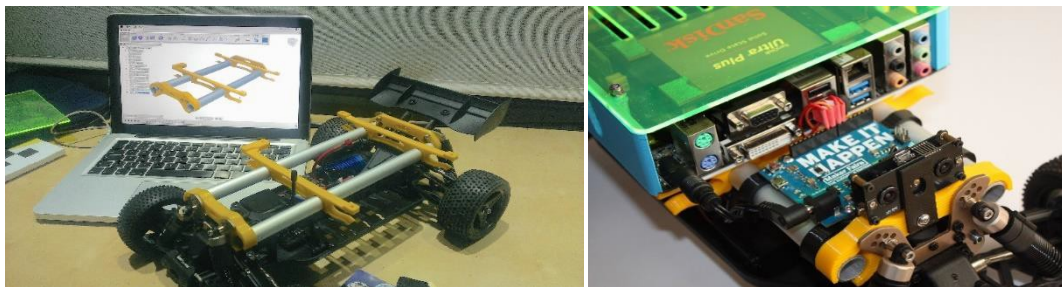


Fig 3.1 RC Car

We can get a car with the whole chassis and wheels included or we can build a car from scratch. In this particular project we decided to make use of an existing car here. The car was designed as per the Ackerman geometry and differential gearing.

Ackermann steering geometry is a geometric arrangement of linkages in the steering of a car or other vehicle designed to solve the problem of wheels on the inside and outside of a turn needing to trace out circles of different radii.

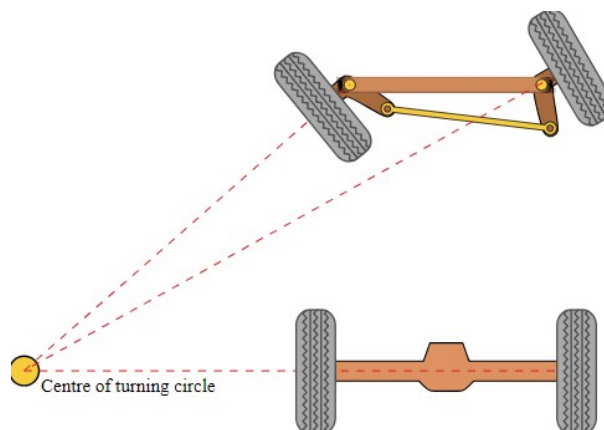


Fig 3.2 Ackermann Steering

The intention of Ackermann geometry is to avoid the need for tyres to slip sideways when following the path around a curve. The geometrical solution to this is for all wheels to have their axles arranged as radii of circles with a common centre point. As the rear wheels are fixed, this centre point must be on a line extended from the rear axle. Intersecting the axes of the front wheels on this line as well requires that the inside front wheel is turned, when steering, through a greater angle than the outside wheel.

Rather than the preceding "turntable" steering, where both front wheels turned around a common pivot, each wheel gained its own pivot, close to its own hub. While more complex, this arrangement enhances controllability by avoiding large inputs from road surface variations being applied to the end of a long lever arm, as well as greatly reducing the fore-and-aft travel of the steered wheels. A linkage between these hubs pivots the two wheels together, and by careful arrangement of the linkage dimensions the Ackermann geometry could be approximated. This was achieved by making the linkage not a simple parallelogram, but by making the length of the track rod (the moving link between the hubs) shorter than that of the axle, so that the steering arms of the hubs appeared to "toe out". As the steering moved, the wheels turned according to Ackermann, with the inner wheel turning further. If the track rod is placed ahead of the axle, it should instead be longer in comparison, thus preserving this same "toe out".

The driven wheels of a vehicle must be able to rotate at slightly different speeds when the vehicle goes around corners, otherwise the tyres would scrub off their tread abnormally fast. Differential gearing allows this.

One of the side effects of this gearing is when the vehicle is jacked up and the engine is off and out of gear turning one driven wheel causes the opposite wheel to rotate in the reverse direction.

For a better understanding of this mechanical concept it is recommended to visit

<http://datagenetics.com/blog/december12016/index.html> (Ackerman geometry)

<http://www.learnengineering.org/2014/05/working-of-differential.html> (Differential gear)

<https://www.youtube.com/watch?v=oYMMdjbmQXc> (Ackerman Geometry, Video explanation)

Other required components for the car include the motors (thrust, servo), an electronic speed controller (ESC), a RF Remote control (if needed).

The specifications of the motors used are as follows:

3.2.1 10t 3500KV brushless motor (thrust motor)

Diameter: 3.17mm

Size: Outside diameter 35.8mm* length 52mm

Weight: 178g

Max Amps: 48A

Resistance: 0.0221

IO A: 1.4

Maximum Power: 500W

Motor uses high quality materials Alloy case- High quality magnets- High temperature resistance copper wires – Good quality bearings.

Dustproof design for motor use for RC 1/10 car.

3.2.3 Electronic Speed Controller

LANSU LS-4040-D Brushless speed controller.

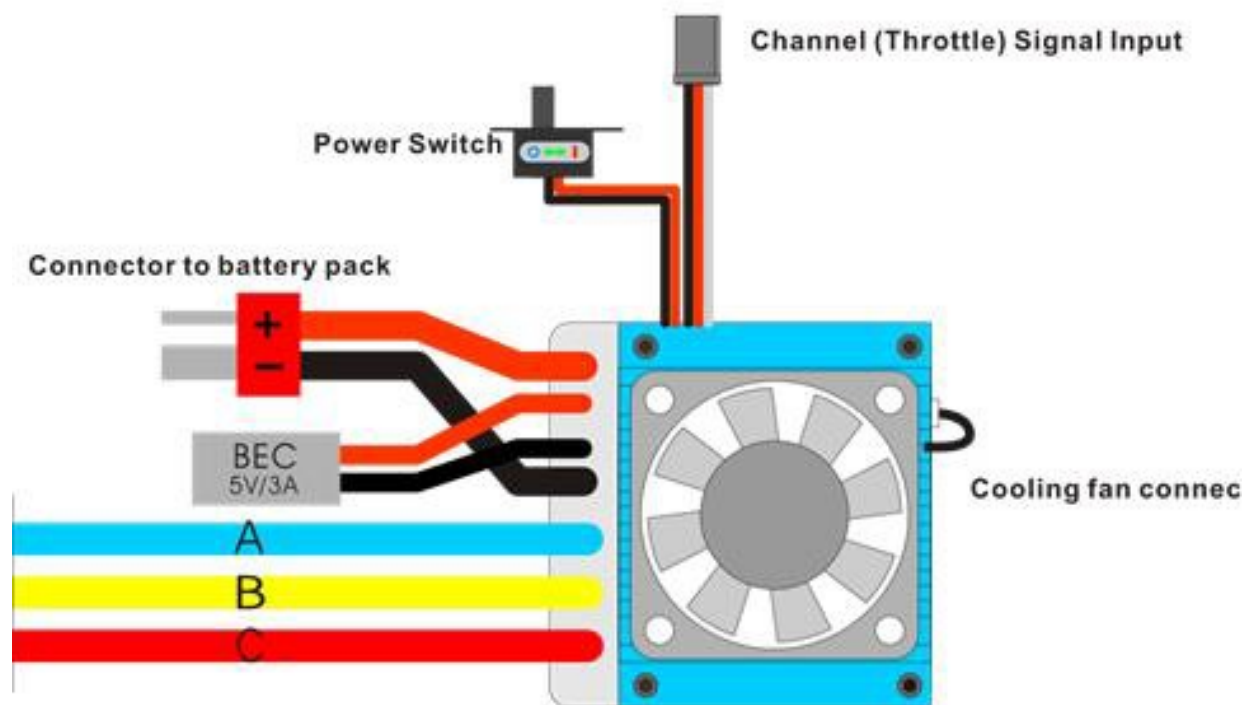


Fig 3.3 Electronic Speed Controller LANSU LS 4040-D

Input Voltage: 7-11.1 V

Compatible with NiMH, NiCd, and LiPo Batteries.

Compatible with 1/10th RC Cars.

Table 3.1 RC Car Technical details

RC car: technical details.	
Measures	0.35 (L) x 0.25 (W) x 0.2 (H) meters.
Weight	2.5 kg
Motor type	Electronic, brushless
Throttle control	Electronic
Steering control	Servomotor
Motion	Four---wheel---traction
Car power	7.4 V battery
Computer and Arduino power	11.1 V battery

3.3 DESIGN AND IMPLEMENTATION

For any autonomously guided vehicle the very first objective is to get used to the sensor, which in our case is a stereo camera. As mentioned above the camera used here is DUO3D stereo camera, in which both the cameras are electronically synchronised, making sure that frames taken are at the same time, and hence making sure that the processes will not lead to any kind of ambiguations with real time processing.

We initially set out to use Raspberry Pi 3 Model B+ as the source of all processing. But as we progressed, we found that Point Cloud Library (PCL), could not be completed, as FLANN (Fast Library for Approximate Nearest Neighbours) could not be compiled. There was another option to install PCL using binary installers, and not compiling from sources. But that too did not work out as it is supported only for stable Debian systems, and not for Raspbian. Hence, we switched over to a system running Linux 14.04.

The camera was mounted on this computer and an enclosure was designed for the system using PLA (biodegradable plastic), laser cut methacrylate and aluminium tubings using 3D printing, so that the PC is guarded and is easy to remove for testing and transportation purposes.

Once the computer was set, all the dependencies were installed.

3.3.1 Camera

In this project we use stereo cameras instead of single cameras. They consist of two camera sensors attached to the same frame and parallel to each other. This setup makes it possible to perform different computer vision techniques, such as scene reconstruction, visual odometry, visual SLAM, and obstacle detection and location.

The code for this project supports DUO3D camera. The camera sends stereo frames over USB, so it is relatively simple to retrieve images. OpenCV provides the VideoCapture class for getting images from generic USB cameras and for the DUO3D stereo camera, Code Laboratories Inc. provides a specialised API.

The advantage of the DUO3D camera is that the two cameras are electronically synchronised, which means that they take images at the exact same time. This is a required feature for scene reconstruction and visual odometry, because otherwise any small movement would make image correspondence erratic.



Fig 3.4 DUO3D Stereo Camera

In the following sections we shall see the mathematical principles used for image treatment, matching and 3D reconstruction.

3.3.1.1 Projective geometry

Almost every object in our world can be sensed by the light it reflects. Cameras receive projections of this reflected light into an array sensor that can record images of a scene. The mathematical study of cameras is very important to gather reliable visual data that computers can use for measuring real-world features.

Having a mathematical model of a camera makes it possible to transform a point from 3D coordinates to projection coordinates and vice versa by means of projective geometry.

The simplest camera model is the pinhole model. A pinhole camera is compounded of a sensor inside an opaque enclosure with a small focus hole –ideally a point– in front of it. Light enters

through the focus and is projected onto the sensor's surface (note that light rays always follow a straight line).

3.3.1.2 From world coordinates to camera coordinates

Let a point in space be $Q(X, Y, Z)$ and a point on the image plane be $q(x, y)$. The distance between the focal point and the image plane, which is formally called focal distance, will be from now on f .

Applying the Thales theorem, which states that similar triangles share the proportions of their hypotenuses and legs respectively, results in the simple relation shown below:

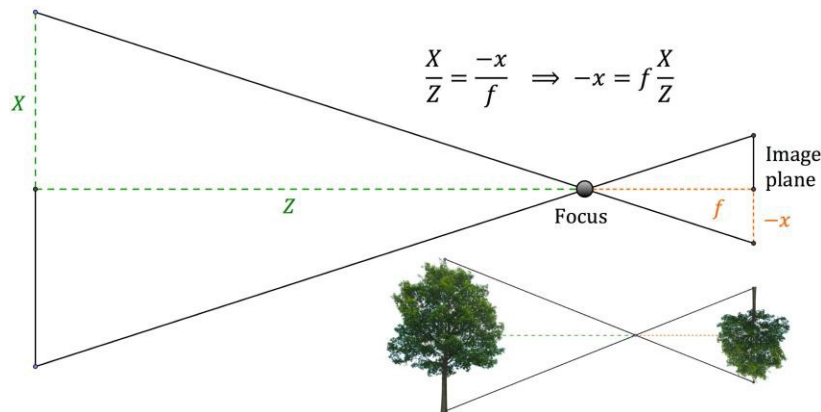


Fig 3.5(a) Thales Theorem

Moreover, the negative sign of the x coordinate of q can be avoided by defining a virtual image plane. This virtual plane is the mirror of the image plane from the focus point.

Note that since the drawing is the central section of the projection cone, the same equation applies for the y coordinate of q , exchanging X by Y .

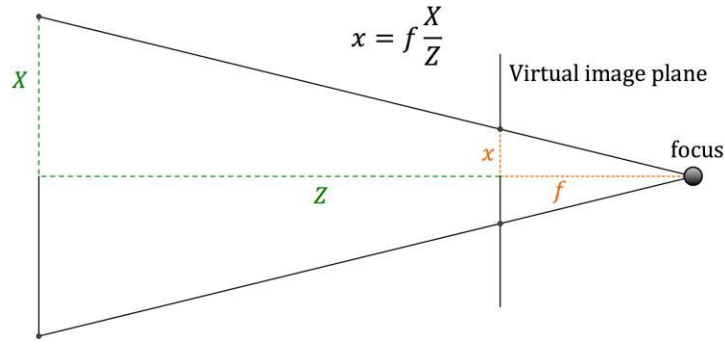


Fig 3.5(b) Thales theorem

3.3.1.3 Pixel Aspect Ratio

Pixels are not always square, in fact, they commonly are rectangular. The coefficient between the pixel's height and width is called the aspect ratio (this should not be confused with the image aspect ratio). Let the pixel aspect ratio be r . Now the y direction on the image plane will be stretched r times with respect to the x direction. The coordinates of q will be:

$$x = f \frac{X}{Z} \quad y = rf \frac{Y}{Z} \quad (3.1)$$

For convenience, the aspect ratio is merged with the focal distance to create a per-axis focal length:

$$f_x = f \quad f_y = rf \quad (3.2)$$

This results in the following projection equations:

$$x = f_x \frac{X}{Z} \quad y = f_y \frac{Y}{Z} \quad (3.3)$$

3.3.1.4 Optical Center

There is often a subtle shift between the image sensor's physical center and its optical center, which is the point where the optical axis intersects the sensor plane. The optical center has the property that receives the light---ray which path is perpendicular to the sensor plane. This point is important, as it is the one with the smallest distance to the pinhole and is considered the origin of the camera coordinate system.

To correct this displacement, c_x and c_y shifting components are introduced in the equations — they usually take small values (2 ~ 3 pixels) on most modern cameras:

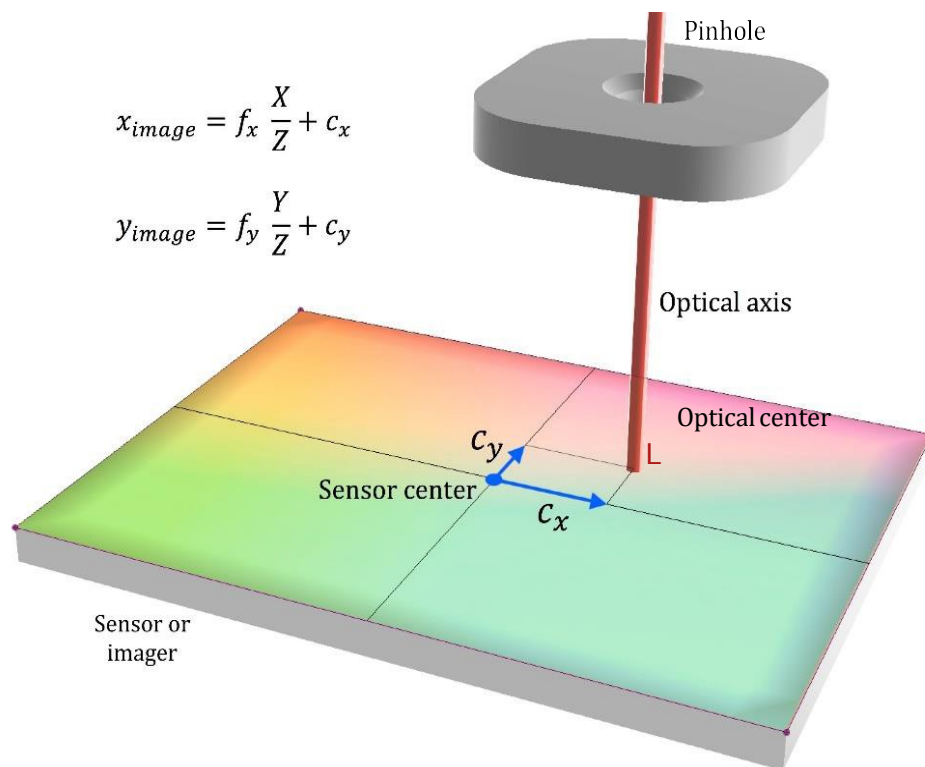


Fig 3.6 Optical Center of a Camera

3.3.1.5 Homogeneous coordinates

Note that the coordinates of Q and q have proportional relations, but there is no way to measure the real magnitude of X , Y or Z from their projected coordinates x and y (which can be measured in pixels). To overcome this issue, the coordinates of q are represented on the homogeneous space, where q acquires an extra dimension that holds its scale. Two points whose coordinates are proportional in the euclidean space, are the same point in the homogeneous space:

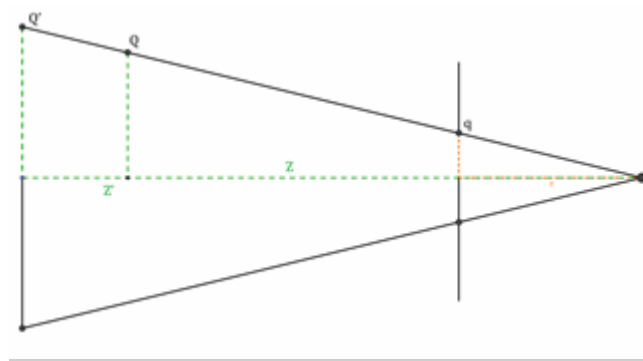


Fig 3.7 Homogeneous coordinates

A point p in homogeneous coordinates verifies that $k \cdot p = p$, where k is a scalar value.

Observe on the image on the left, that either Q and Q' project to the same point q on the image plane.

The Euclidean coordinates of q are recovered dividing all coordinates by w ,

$$q = \begin{bmatrix} \frac{x}{w} \\ \frac{y}{w} \\ 1 \end{bmatrix} \quad (3.4)$$

Because the projection focus is at the coordinates O (0,0,0) in Euclidean space, the homogeneous coordinate takes the same value as z.

Returning to the projection equations, which were written in Euclidean coordinates, they can be easily converted to homogeneous coordinates multiplying them by the scaling value:

$$q = \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x X + c_x Z \\ f_y Y + c_y Z \\ Z \end{bmatrix} \quad (3.5)$$

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (3.6)$$

This can also be expressed as the result of a matrix that contains all the intrinsic parameters arranged in a way that makes projection operations very efficient and fast.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad (3.7)$$

This matrix is called the camera matrix and is represented as K. The rapid notation usually used for the above matrix equation is:

$$q = KQ \quad (3.8)$$

3.3.1.6 Distortion coefficients

Pinhole cameras reproduce proportions perfectly but, they have a big drawback: The hole must be so small that only one light ray per direction passes through. This makes images be dark because very little light reaches the sensor.

The solution is to replace the pinhole with a big lens, which redirects many light rays to make them converge to a single point on the imager plane. This greatly increases the amount of light that receives the sensor, thus reducing the exposure time to small fractions of a second.

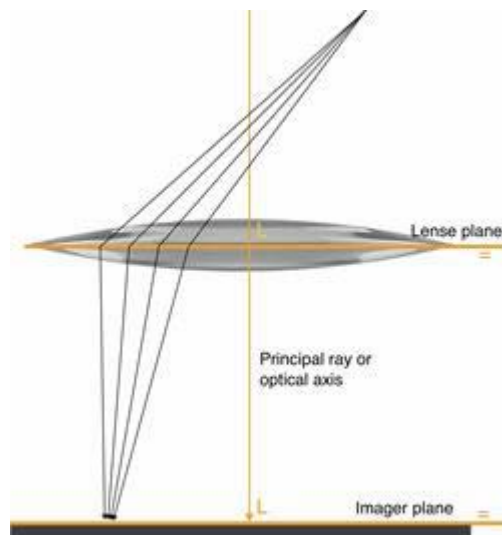


Fig 3.8 Distortion due to lens

But lenses introduce another problem: distortions. As the direction of light rays is modified, the shape of the lens affects the location of projections.

Of course, it is possible to mathematically model a perfect lens that does not introduce distortions, using parabolic curves. But producing precision lenses is expensive and so the common shape is the intersection of two semi spheres.

The described model equations for the pinhole cameras would not be very useful on lens-based camera if the distortion introduced by the lens was not removed before.

There are two main kinds of distortions:

- Radial distortions: These are caused by lens magnification varying with the radius and make straight lines appear curved. They are the most common type of distortion and

can be categorised into barrel, pincushion and moustache distortions depending on the curves they form.

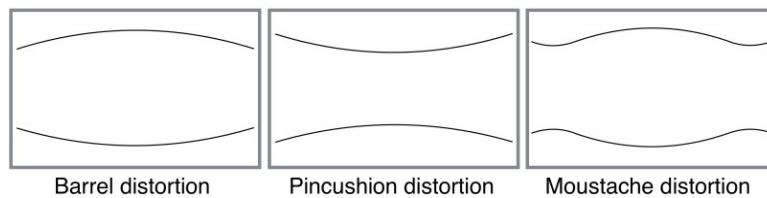


Fig 3.9 Types of distortions

- Tangential distortions: These may happen when the lens is not perfectly parallel to the imager sensor. They are less common than the radial distortions and are unlikely to appear even on modern cheap cameras.

Radial distortions become more obvious at the edges the images but have nearly no effect around the center.

The most common procedure for parameterising radial distortions is to fit distortion curves with relatively high degree polynomials.

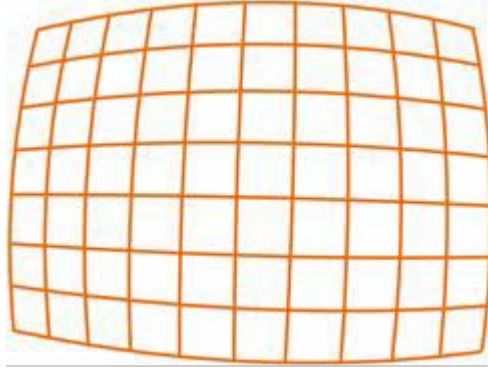


Fig 3.10 Radial distortion

Let r be the radius from the center of the imager to the location of any pixel. The value of r is between 0 and 1, where 0 would be a point at the center and 1 a point at the farther edge. The fitted polynomials result from the Taylor series expansion around $r=0$:

$$f(r) = a_0 + a_1 r + a_2 r^2 + a_3 r^3 + a_4 r^4 \dots \quad (3.9)$$

The degree of these polynomial curves must be even, so the curves are symmetric around r . Because of this, all uneven-indexed coefficients will be 0.

Furthermore, because the center point is on the optical axis, which is perpendicular to the lens plane and thus is the only axis that does not distort light ray paths, $f(0)$ must be 0. This implies that $a_0 = 0$ and makes the above Taylor series become a Maclaurin series.

Finally, it is usually not necessary to go further than 6 degrees, so that first three even powers of r are enough for our distortion model. Three radial distortion coefficients, often named k_1 , k_2 and k_3 , control the effect of each power of r .

The resulting distortion-removing equations are:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (3.10)$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (3.11)$$

These equations remap each pixel to its new “corrected” location – as seen in the vector field, where red dots represent the original pixel locations and green dots represent the corrected pixel locations.

The remapped picture is cropped to a valid region, which conserves the resolution of the camera. If needed, the remapping function, which is shown in a later section, merges or interpolates pixels.

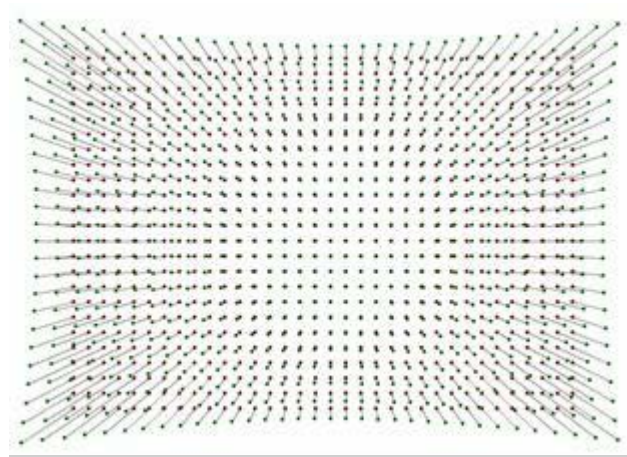


Fig 3.11 Original(red) and corrected(green) location of pixels

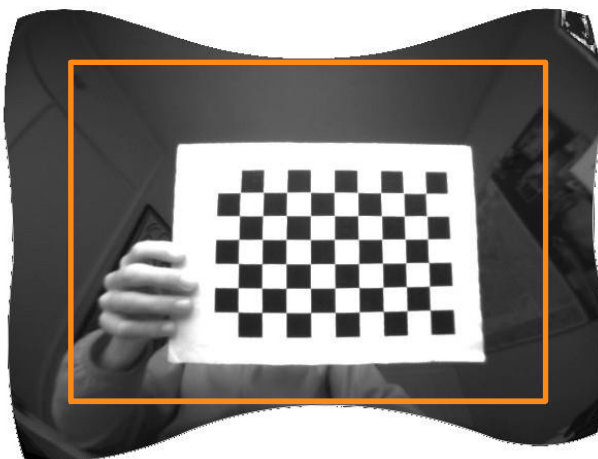


Fig 3.12(a)Undistorted image



Fig 3.12(b)Undistorted and cropped

3.3.1.7 Stereo Triangulation

Now that we have modelled the projection for a single camera, we proceed to study the relation between two cameras – a stereo pair. With two points of view, it is now possible to triangulate the depth of a pixel and therefore reconstruct a 3D scene.

Let Q be a 3D point of the scene, O_1 and O_2 the centers of projection (centers of the lenses) of the left and right cameras respectively, b the horizontal distance between O_1 and O_2 (usually named baseline) and f the formerly defined focal distance.

The projection of point Q towards O_1 and O_2 produces a point on each camera plane which coordinates on the X axis are x_1 and x_2 . It is always true that $x_2 \leq x_1$.

Let d be the difference between these two coordinates, which in fact is called disparity:

$$d = x_1 - x_2 \tag{3.12}$$

By rearranging the camera centers we can appreciate that Z_0 , f , b and d maintain the following proportional relation:

Figure 1 consists of two ray diagrams illustrating the effect of translating the origin of the coordinate system. The left diagram shows a point source Q at height b from the lens plane. The image planes are at distances f and b from the lens plane. The right diagram shows the same setup after translating the origin O_2 to O_1 , resulting in a new image plane at distance b from the lens plane and a new source point Q' .

Rotate the right coordinate frame vertically around O_2 and translate it to O_1 .

$$x_I = f_x \frac{Q_x}{Q_z} \iff Q_x = x_I \frac{Q_z}{f_x} \quad (3.13)$$

31

By convention, the origin of the coordinate system on a stereo camera is the center of the lens of the left camera. Therefore, the coordinates Q are with respect to O_1 .

To bring it to a bigger scale and reconstruct a whole scene, the first step is to compute a dense disparity map. This step requires solving the correspondence problem, which means finding the same pixels on both left and right images.

Note that it is not always easy to find matching pixels on both images. The image below illustrates two stereo correspondences. The one in orange color is easy to detect as it is located on a corner where neighbouring pixels have a large difference in color. But the one in green is harder to detect and requires more complex algorithms.

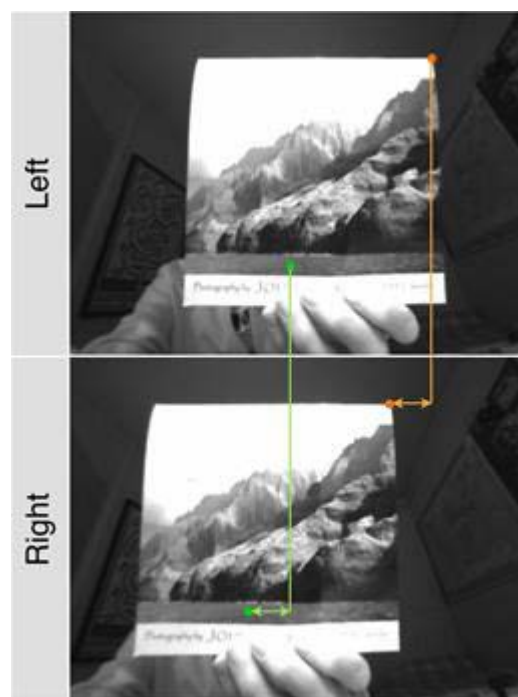


Fig 3.15 Detection of matching pixels

Instead of picking a pixel from one image and comparing it to all the pixels on the other image, which would be very inefficient, there are some rules that constrain the search to a small region of pixels. These rules are due to the epipolar geometry of the stereo system.

If the cameras are perfectly horizontally aligned and their image planes are on the exact same plane in space, corresponding pixels are always on the same pixel row on both images.

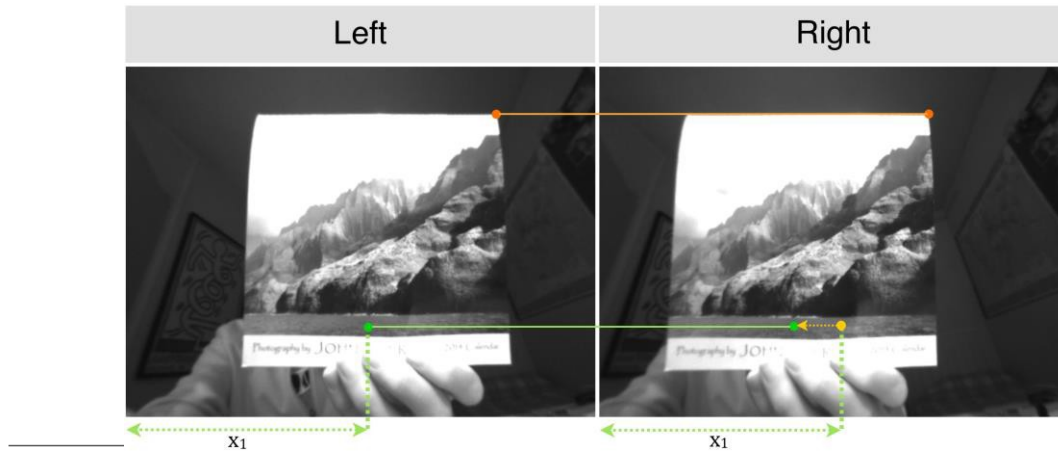


Fig 3.16 Pixels when the cameras are horizontally aligned

Disparity values can only range from 0 to image width and are always directed towards the middle point between the two cameras (see yellow arrow in the previous comparison image). So, if we take a pixel on the left image and want to find its correspondence, we can apply the following constraint:

$$\begin{aligned} y_2 &= y_1 \\ x_2 &\in [0, x_1] \end{aligned} \quad (3.15)$$

Where x_1, y_1 are the pixel coordinates on the left image and x_2, y_2 the pixel coordinates on the right. This greatly increases efficiency as the maximum number of comparisons for each correspondence is x_1 , although the matching pixel is usually found way before reaching $x_2 = 0$.

The dense disparity algorithm used in the code for this project is the *Semi Global Block Matching*. It works by processing groups of pixels instead of the individual pixels and thus, it gathers much more data to compare between frames. Even though, there can be areas in the

image that are so homogeneous in colour that the algorithm cannot find any suitable match. In that case it leaves a default value that is interpreted as a null data point. Mapping all disparities into a greyscale image – where darker means farther, lighter means closer and black means that no correspondence was found.

Once the disparity map is computed the next step is computing a depth map, where each pixel contains the 3D coordinates of its corresponding 3D point. This can be done either with custom code using the previous equations or with the OpenCV's built-in function

reprojectImageTo3D(InputArray disparity, OutputArray 3DImage, InputArray Q).

Representing depth maps is a bit more tricky than representing disparity images. A specialised visualiser is needed to render and navigate the point cloud. Our code uses the PCL library for this purpose.

The images below show four views of a depth map from a stereo frame of myself, which was taken in real time. Points are coloured red to green depending on their z component to better appreciate depth.

3.3.1.8 Camera Calibration

The previous section assumed that the two cameras were completely parallel and only shifted in the X axis direction, which made matching pixels coincide in row. This, in practice, is very unlikely to happen, as small angles between both cameras can drastically affect Epipolar lines.

We also are using the pinhole camera model, thus assuming that the images were already undistorted.

This section describes the calibration procedure which is used to parameterize lens distortions and camera intrinsics and extrinsics.

The first step in the calibration procedure is detecting points in a calibration pattern, which coordinates will be later passed to a calibration function. In the case of OpenCV the calibration function. In the case of OpenCV the calibration pattern is a chessboard with a custom number of squares.

OpenCV provides a function for detecting the internal corners of the pattern:

(bool) findChessboardCorners(InputArray image, Size boardSize, OutputArray corners);

Where `boardSize` is an OpenCV's struct containing the number of internal corners—in our case, it is `Size(9, 6)`. The function returns true if all the corners were found.

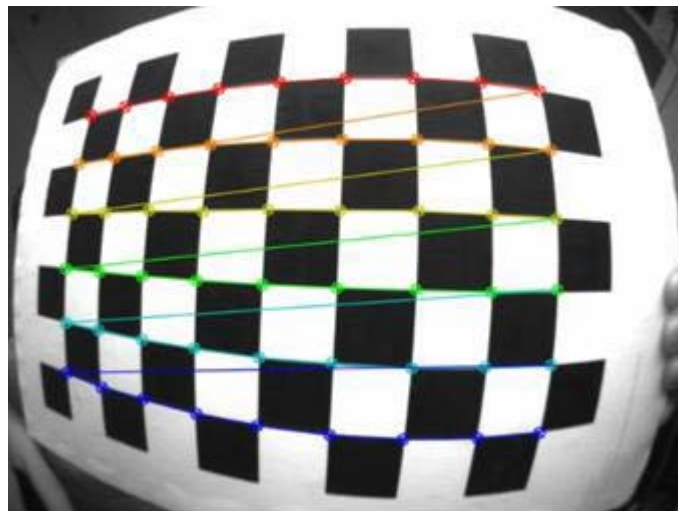


Fig.3.17 Checkerboard with size (9,6)

The corners' coordinates are in pixel units, which are natural numbers. OpenCV provides an iterative method for further refining these coordinates by analysing the surroundings of each corner and finding a more precise location—for example, coordinates $\mathbf{c_n}(233, 125)$ may become $\mathbf{c_n}'(233.43, 124.87)$:

```
cornerSubPix(InputArray image, InputOutputArray corners, Size winSize, Size zeroZone,
cv::TermCriteria criteria);
```

Where corners is the same array that was filled with the imprecise corner locations, winSize is the rectangle of surrounding pixels used for computation, zeroZone is an inner area of pixels to ignore –this is to ensure that the matrix deNined by winSize is invertible, which is a needed feature for the algorithm– and criteria is used to stop iterating when such criterion is met.

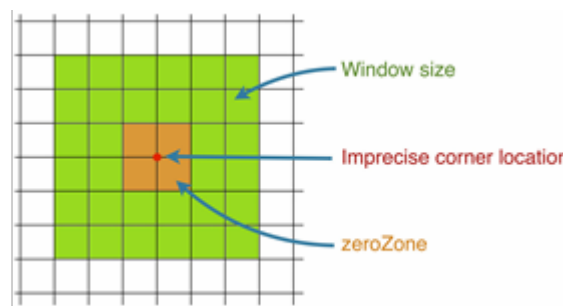


Fig 3.18 Calibration

In our code:

```
winSize = Size(11, 11);
```

```
zeroZone = Size(-1, -1);    //Negative numbers disable ignoring an inner area. criteria =
```

```
TermCriteria(CV_TERMCRIT_ITER + CV_TERMCRIT_EPS, 30, 0.01)
```

Criteria types (OpenCV internals)

Number of iterations Accuracy

For best results, several images of the calibration pattern should be taken (10 to 20 are usually enough).

Running the calibration algorithm, computes intrinsic parameters, distortion coefficients and calibration pattern translation and rotation for each image and for each camera.

Finally, for undistorting and rectifying the images, instead of applying a function for each stereo frame, a mapping array that relates each pixel in the raw images to its corrected location is computed. The following code shows how to compute rectification maps and use them to undistort and rectify left and right images.

3.3.1.9 StereoCam Class

All the topics discussed in this section are implemented in the project's code and grouped into a class named StereoCam. The capabilities of this class include:

- Support for Duo3D camera.
- Real-time calibration utility: it detects pattern corners in real time and displays them on a window. By pressing the 'n' key the program takes a new stereo frame. When a number of images have been taken (16 pairs by default) the calibration functions are called. After a few seconds, the resulting parameters are saved to a persistent file and a window opens showing rectified pairs in live-stream.
- Disparity map generation: it uses Semi-Global Block Matching (SGBM) and includes the option to open a display window and a window with sliders to tune each of SGBM's parameters and to see the results in real time.
- Depth map generation: this converts disparity maps into 3D points. A complementary function opens a window with a 3D view of the resulting point-cloud which can be zoomed, orbited and panned.

3.3.2 Data Analysis

Once the stereo images are rectified they are used for two different purposes. They are first used to update visual odometry, and secondly to detect obstacles.

There are many more applications on which to use images, some examples are mapping, pedestrian detection and optical flow —which is used to detect moving obstacles. The possibilities can be further expanded with image segmentation techniques, which are capable of classifying objects, materials, and other features.

3.3.2.1 Visual Odometry

Visual odometry is the process of tracking the position of a stereo-camera by comparing successive images [3].

The first step to perform odometry is to identify matching points between images taken at different times. The matching direction and the reprojection¹⁵ of those points will be used to compute the rotation and translation of the camera between frames.

In this case we cannot apply any Epipolarity rules because the camera may move in any direction. Therefore, the algorithm will need to compare all the pixels of one image with all the pixels of the other image. This is a very intensive process, so we need to perform it only on a selection of points: those that will be easy to identify in both images.

We call this special points key-points. They are usually salient features such as color blobs, corners or edges.

3.3.2.2 Key Point Detection

The algorithm consists of three steps:

Detecting key points: Key points can be corners, blobs, etc. The detector we use is called FAST (Features from Accelerated Segmented Test) and is a corner detector. Corners are pixels in which colour levels differ from their neighbour's according to a pattern.

FAST uses a heuristic approach: it compares, in a circular pattern, pixel's brightness with a threshold. This requires less computing power but introduces noise (points that are not real corners). Then a high-speed test is performed to filter false corners. The test checks if pixels P1 and P9 have a brightness difference higher than another threshold. If they do not, the test is performed with pixels P5 and P13. If both pairs do not pass the test, the corner is dismissed.

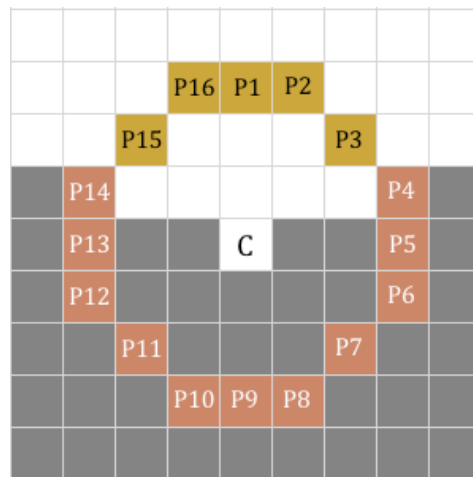


Fig 3.19 FAST (Features from Accelerated Segmented Test)

Computing feature descriptors: A descriptor algorithm computes a description for each key-point that is as exclusive as possible. This is later used to match them between pictures. They pick pairs of neighbour pixels according to a pattern and compare their brightness. For instance,

if b_1 is the brightness of the first pixel and b_2 is the brightness of the other pixel, the comparison $b_1 \geq b_2$ can be true or false. The comparison results are written on a binary string (1 for true and 0 for false).

The descriptor we use in our code is the BRIEF descriptor, which is one of the fastest and simplest. It chooses pairs randomly, instead of using a pattern.

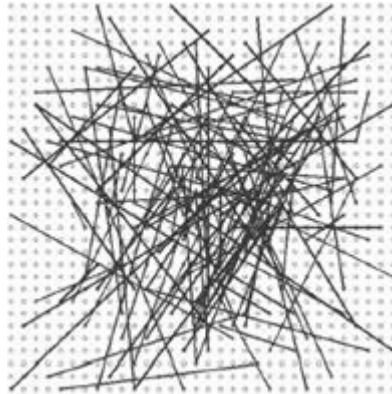


Fig 3.20 BRIEF's random sampling pairs

Matching key points: In this step, key points from different images are matched using their descriptors. There are two main matching algorithms: Brute Force and FLANN based. The Brute Force algorithm, which is the one we use, takes each key---point from an image and compares it with each of the key---points on the other image. The one whose descriptor string has the lowest Hamming distance to the descriptor string of the key--- point that is being compared is considered the good match.

Our program also does a cross check, which means that the matches are computed from the first image to the second image and then from the second image to the first image. The matches that are not equal on both attempts are dismissed.

The FLANN-based algorithm is a heuristic approach, but the efficiency difference with Brute Force does not become noticeable for such small number of key-points –which I limited to 200.

3.3.2.3 Pose change estimation

The relative pose (rotation and translation) of the camera with respect to its last pose is estimated by minimising the cost function:

$$F(R,t) = \sum_i \left\| k(RQ_{t_0}^{(i)} + t) - q_{t_1}^{(i)} \right\| + \left\| k(R^T Q_{t_1}^{(i)} - t) - q_{t_0}^{(i)} \right\| \quad (3.16)$$

Where t_0 refers to the first image and t_1 refers to the second image and are the 3D locations of a point reprojected at t_0 and at t_1 , respectively, k is the intrinsics matrix of the camera and R and t are the rotation and translation matrices:

$$R = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \quad (3.17)$$

$$t = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \quad (3.18)$$

What this function does is:

1. Reproject matching key-points to their 3D coordinates I their corresponding camera coordinate frame as described in previous sections (Stereo triangulation). We call 3D points $Q_{t_0}^{(i)}$ the ones reprojected from the stereo frame at t_0 and $Q_{t_1}^{(i)}$ the ones reprojected from the stereo frame at t_1 .
2. Rotate and translate $Q_{t_0}^{(i)}$ to its location in t_1 with new values for R and t and project it to the camera plane at t_1 :

$$q_{t_1}^{(i)} = k(RQ_{t_0}^{(i)} + t) \quad (3.19)$$

3. Calculate the error (e_1) between the projection of the reprojection ($q_{t_1}^{(i)'}$) and the actual projection of that point at t_1 ($q_{t_1}^{(i)}$).

$$e_l^{(i)} = \|q_{t_l}^{(i)'} - q_{t_l}^{(i)}\| = \|k(RQ_{t_0}^{(i)} + t) - q_{t_l}^{(i)}\| \quad (3.20)$$

4. Similar to step 2. Rotate and translate $Q_{t_l}^{(i)}$ to its location in t_0 with new values for R and t and project it to the camera plane. Note that since we are now moving the points from t_1 to t_0 we need to do the opposite rotation and translation:

$$q_{t_0}^{(i)} = k(R^T Q_{t_l}^{(i)} - t) \quad (3.21)$$

5. Repeat step 3 to calculate the new error:

$$e_0^{(i)} = \|q_{t_0}^{(i)'} - q_{t_0}^{(i)}\| = \|k(R^T Q_{t_l}^{(i)} - t) - q_{t_0}^{(i)}\| = \|k(RQ_{t_0}^{(i)} + t) - q_{t_0}^{(i)}\| \quad (3.22)$$

6. Sum the errors of the reprojection—projection process for all the points:

$$F(R, t) = \sum_i e_l^{(i)} + e_0^{(i)} = \sum_i \|k(RQ_{t_0}^{(i)} + t) - q_{t_l}^{(i)}\| + \|k(R^T Q_{t_l}^{(i)} - t) - q_{t_0}^{(i)}\| \quad (3.23)$$

Minimising this function yields the rotation and translation matrices that best describe the movement of the camera between frames t_0 and t_1 .

3.3.2.4 Obstacle detection

There are many levels of detection, some common examples are:

By volume: Size, distance, etc. By kind: Cars, people, doors, etc. By meaning: gestures, signs, etc. By behaviour: static, mobile, etc.

By material: grass, wood, steel, etc.

I have opted for a simple approach, as the objective is to avoid collisions in simple cases. The obstacle detector detects objects only volumetrically. This means that it does not detect what kind of object it is or whether it is mobile or static. Instead, it is based in the assumption that the world and all its elements are static on the time each frame is taken. To detect obstacles, we simply compute a disparity map and calculate the 3D location of the pixels that are nearer than a threshold. These points are identified as obstacles and are introduced in an obstacle map.

Obstacle Maps:

The obstacle points which Y component is less or equal to the car's height are threshold into flat squares to make the planning algorithms simpler. The rest of the points are discarded. The squares' measures can be adjusted to make planning more precise or more efficient, depending on the situation.

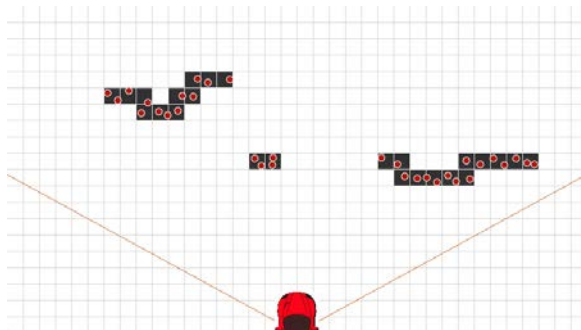


Fig.3.21 Obstacle map simulator with 20 cm squares (red circle represents the detected points and black squares the threshold).

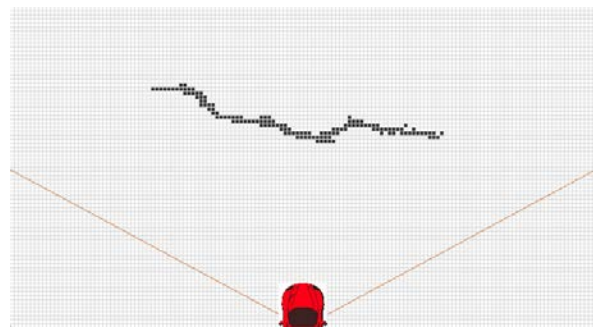


Fig.3.22 Custom obstacle map simulator with 5 cm

3.3.3 Path Planning

Although the optimum path to reach a point is always the direct straight line to it, it is often not possible to follow this line due to the environment. Autonomous cars, for instance, need to navigate through obstructed places where sometimes there is even no possible way to the target location. A set of computer algorithms process the data generated during the odometry and obstacle detection to find alternative routes to the destination.

We divide this section into two parts:

- Global route planning
- Avoidance path planning

3.3.3.1 Global route planning

In many scenarios a mission can largely benefit from a pre-planned path. For example, a wheeled robot that must move along a building can calculate optimal paths from a digital map of the facility. This greatly reduces the chance of encountering obstacles or getting lost, but with the inconvenience that in many cases such map is not available.

One approach is to manually navigate the environment while the vision system creates the map. However, creating a reliable map using a vision system is not an easy task. Even today's most advanced projects on this area still require some input from another sensor, database or human corrections or otherwise they lose precision. Another approach is to use a rotating laser rangefinder (also known as LIDAR), but these sensors have the inconvenience that they are very costly.

For the purpose of this project, we will skip creating this software part but still provide a brief explanation on how an optimal path to a location would be calculated if we had a map.

Consider a map made of squares as a graph with $A \times B$ nodes, each of them connected to its 8 neighbours. The role of a path finding algorithm is to traverse the graph to find a list of adjacent nodes that get from the origin node to the target node avoiding those nodes marked as walls. Most of the times, this list of adjacent nodes is required to be the shortest path.

A very popular path finding algorithm is the Breadth First Search. Most the other path finding algorithms are just optimisations based on this approach.

Breadth First Search's principle is simple; it expands from the origin node towards its neighbours and repeats the expanding operation from neighbour to neighbour until it reaches the target node. The path is reconstructed simply by linking each node with the one from which it came from, starting from the target node. In fig, the red star node is the origin and the green star node is the target. The arrows represent the inverse direction of the neighbouring expansion. To draw the path, we simply follow the arrow directions starting from the green star node.

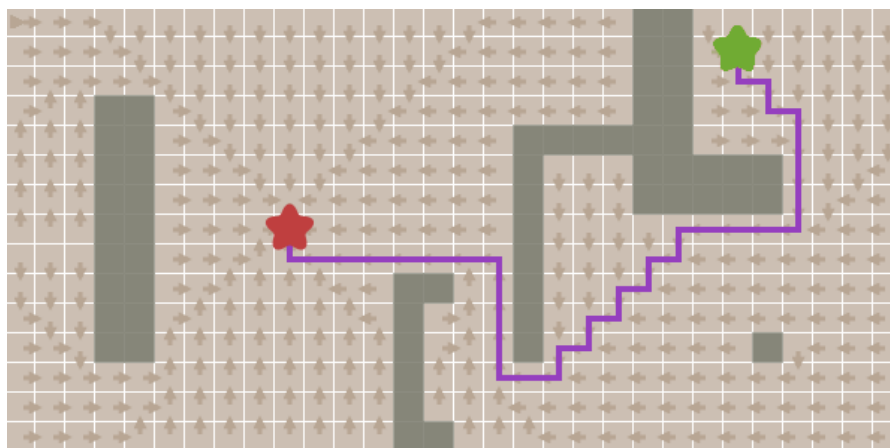


Fig 3.23 Breadth First Search Path Reconstruction. Image from Red Blob Games.

Other frequently used algorithms for path planning are Dijkstra, Best-First Search and A* (pronounced “a star”) (see figures). All the three are based on the Breadth First Search and add weighting values for the nodes depending on their distance and other characteristics. The first two algorithms also add a heuristic optimisation technique to make them more efficient. Instead of expanding isotropically the y expands towards the target node with the hope to reach it sooner, which is not always the case.

The problem of Best-First Search is that it does not guarantee to find the shortest path. A* was created by merging the Best-First Search and the Dijkstra algorithms with the aim to be fast but still guarantee to find the shortest path.

Dijkstra labels each node with the taxicab distance to the origin node and Best-First Search does the same but with the distance to the target node.

A* does the trick of summing both values. The shortest path is the one with the lowest sum, as can be seen on fig 3.27, that follows the arrow directions as in fig 3.25.

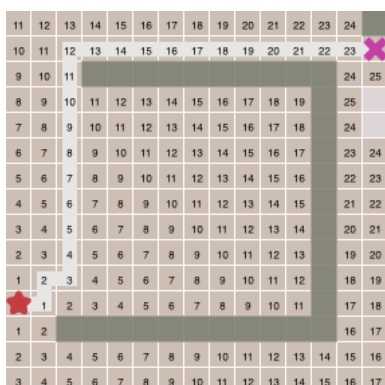


Fig 3.24 Dijkstra

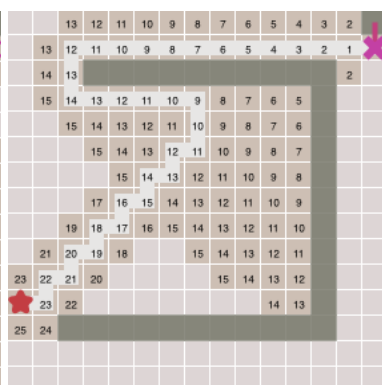


Fig 3.25 Best –First Search

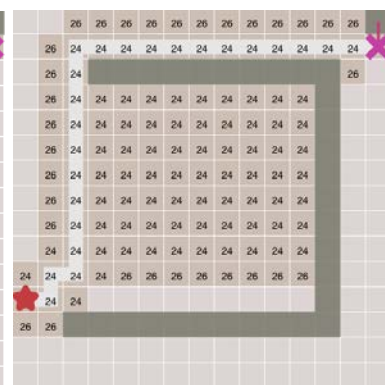


Fig 3.26 A*

3.3.3.2 Avoidance Path Planning

The Robot can come up with something blocking its path that was not expected during the planning of its mission. The aim of the avoidance software is to deal with these unexpected changes and provide alternative paths on the go.

For this project, a custom avoidance path---planning algorithm was developed.

CHAPTER 4

RESULTS AND DISCUSSION

Many functionalities have been implemented in this project. We will look at them one by one.

4.1 STEREO CAMERA

Code for using DUO3D Stereo Camera has been written using OpenCV and the Duo Library, providing functions to capture continuous frames etc.



Fig 4.1 Uncalibrated Left Camera Image



Fig 4.2 Uncalibrated Right Camera Image



Fig 4.3 Both the Left and Right Images are horizontally merged and lines are drawn for easier comparison of the pixels of the images.

4.1.1 Calibration

As explained in section 3.3.1.8, cameras suffer many different types of distortions, as with the following one, it is radial distortion. Calibration is performed to eliminate these distortions.

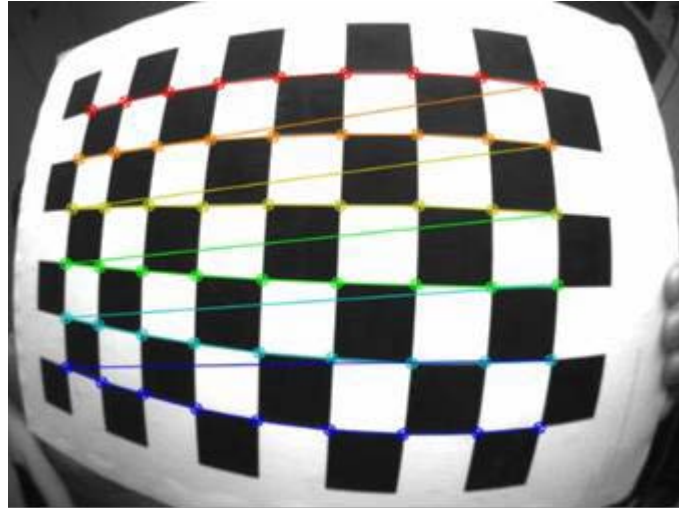


Fig 4.4 Uncalibrated Checkerboard

The image below shows calibrated and rectified images with horizontal lines to help visualize better.

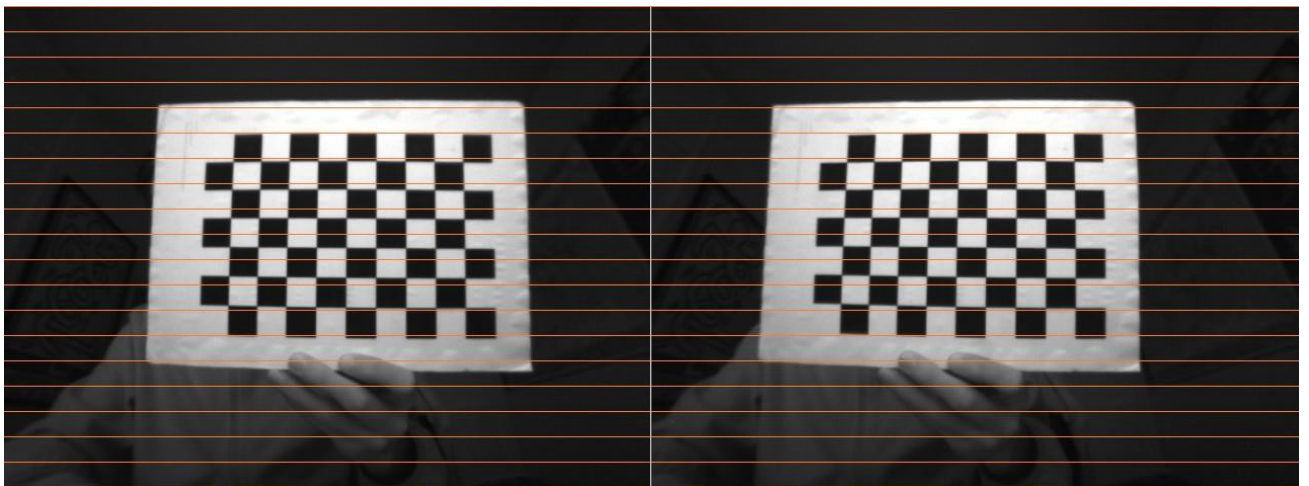


Fig 4.5 Calibrated checkerboard with horizontal lines for visualization

The next step in the project is to produce the disparity maps using the frames.

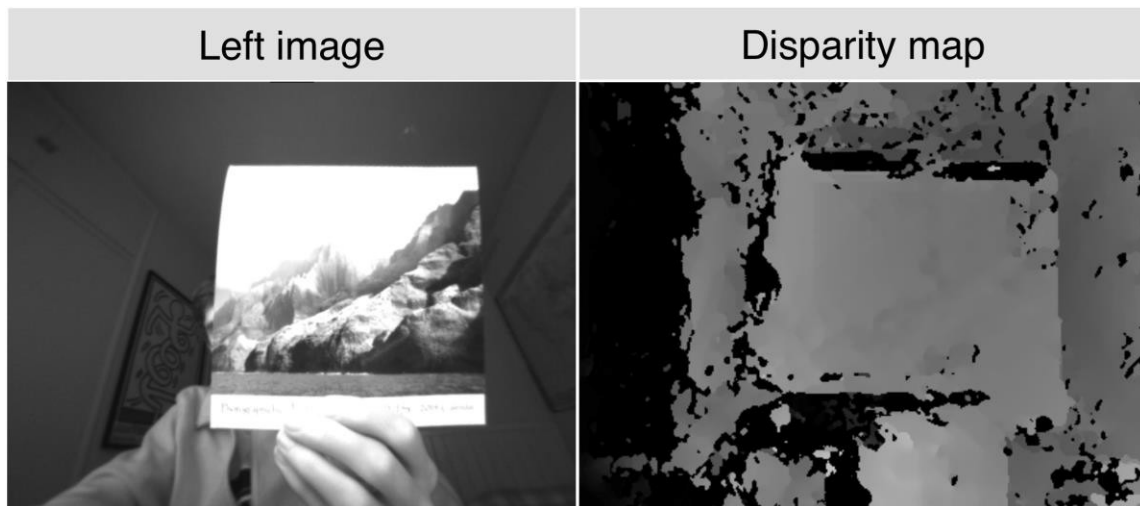


Fig 4.6 Disparity map

As mentioned earlier, a group of pixels are processed instead of comparing individual pixels and hence, gathers much more data to compare between frames.

Mapping all disparities into a greyscale image –where darker means farther, lighter means closer and black means that no correspondence was found– yields Fig 4.6.

Once the disparity map is computed the next step is computing a depth map, where each pixel contains the 3D coordinates of its corresponding 3D point. Depth map was coded but could not be verified. The following 3D images were captured from the application DUO Dashboard, which gives an idea as to how the depth map is supposed to look like.

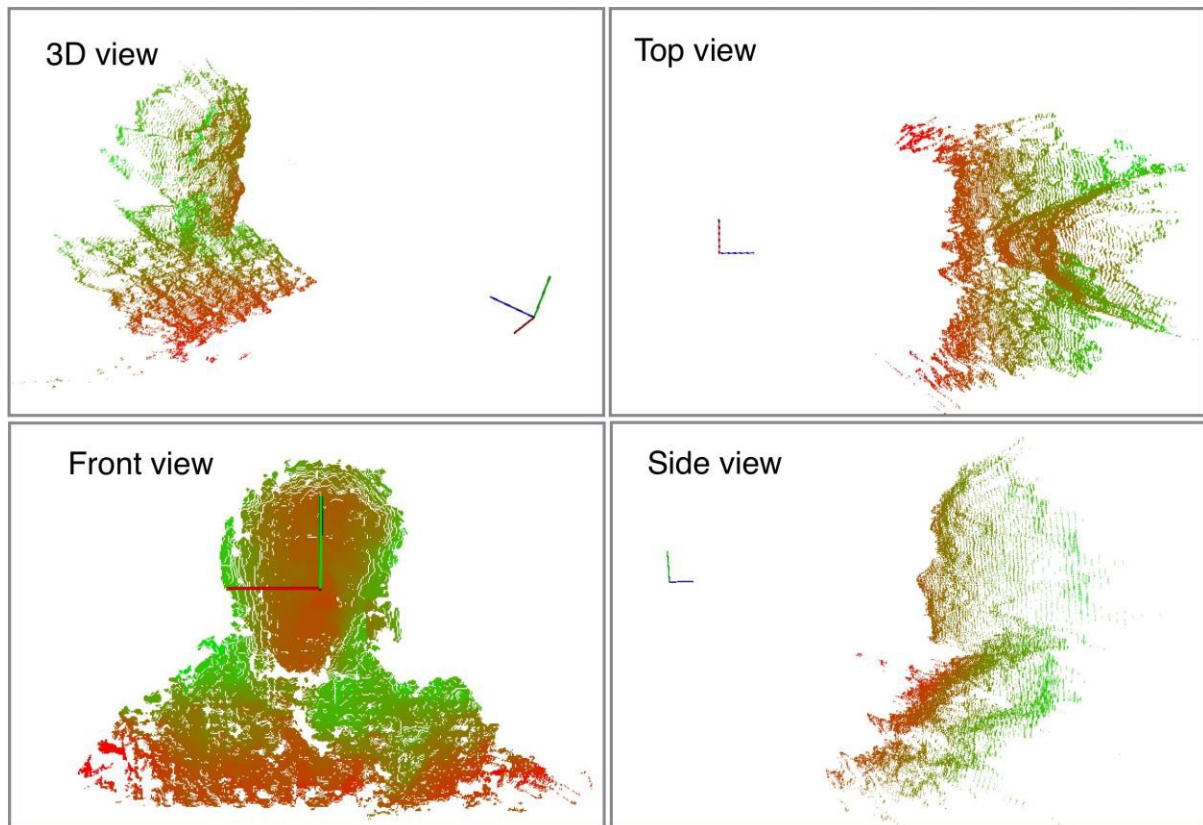


Fig 4.7 Dense 3D map of myself

The images show four views of a depth map from a stereo frame of myself, which was taken in real time. Points are coloured red to green depending on their z component to better appreciate depth.

The following two views are to demonstrate that this method can detect horizontal surfaces fairly well.

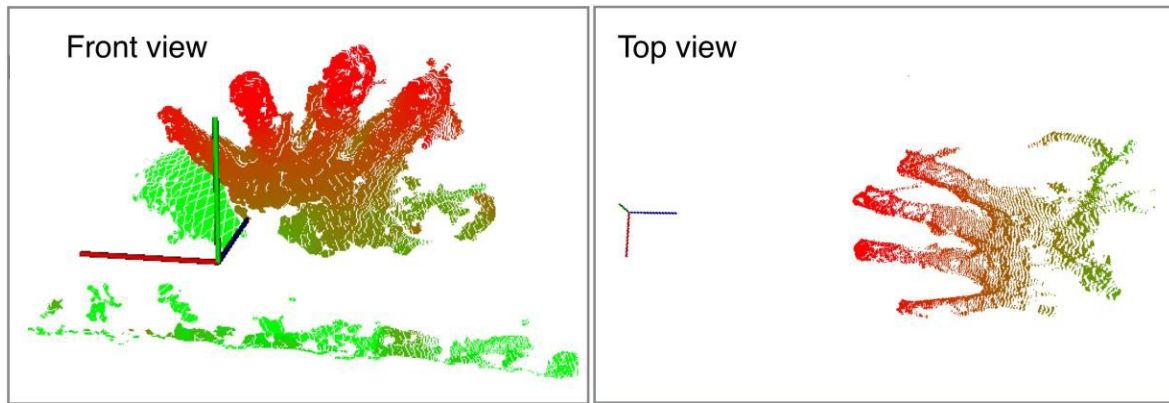


Fig 4.8 3D map of hand

Later we will use this 3D data to detect obstacles and to generate avoidance paths.

We have applied brute force method for visual odometry and the results are as obtained.

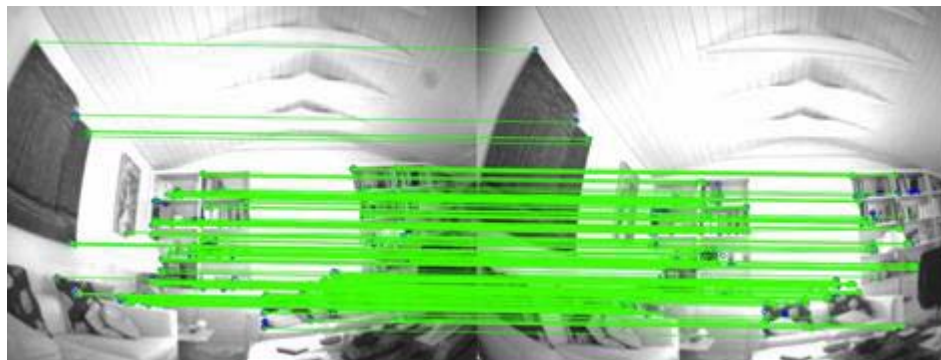


Fig 4.9 Brute Force match

The FLANN based algorithm follows a heuristic approach, but the efficiency with Brute force does not become noticeable for small number of key-points which has been limited to 200.

4.1.2 Avoidance Path Planning

For this project the Path Planning algorithm that was used works in the following manner.

Step 1: If there's any obstacle, it calculates all the ranges of curve radii that would not produce any collision. Then it chooses the curve radius that is closest to the original trajectory.

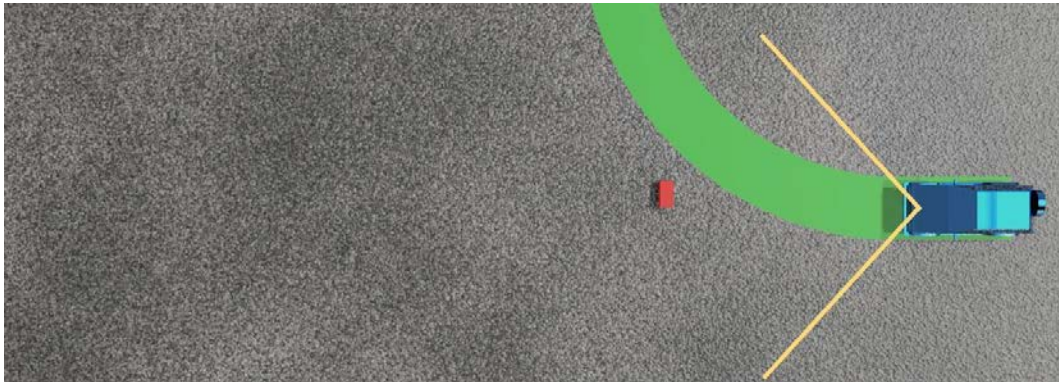


Fig 4.10 Avoidance curve path (artwork)

Step 2: Once an obstacle is avoided, and if there is no other obstruction detected, the algorithm plans a path that returns the vehicle smoothly to the planned trajectory. This path is made up of two tangent arcs.

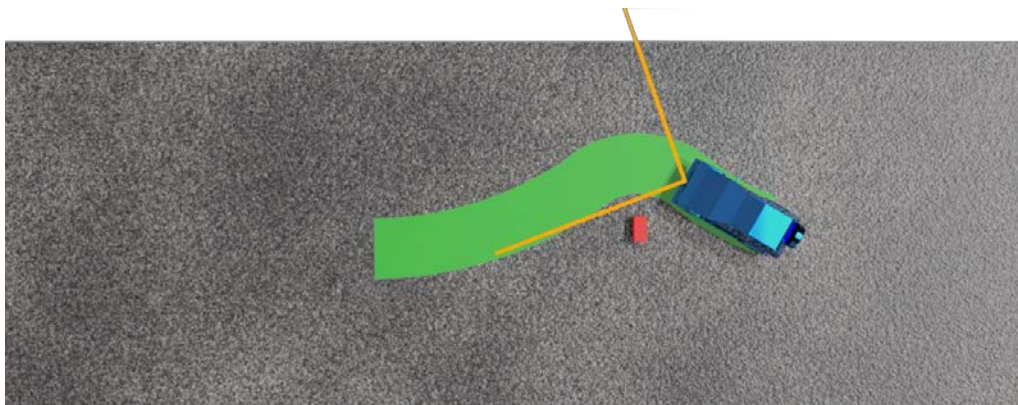
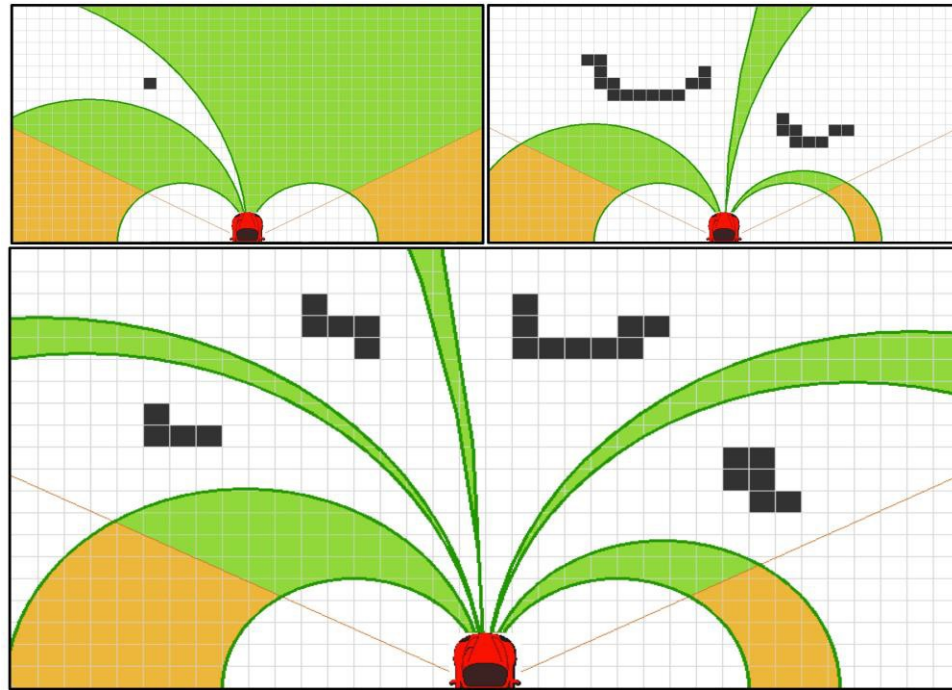


Fig 4.11 Avoidance curve path (artwork)

If the vision system detects more obstacles, then it keeps on tracing avoidance paths. This system could fail when there is no possible path (for example, a corner). In this case, the on-board computer sends a directive to the Arduino microprocessor to undo movements until the vision system can find another way.

The following screenshots show the simulator that was created by Alejandro Daniel Noel to develop the avoidance algorithm. The simulator allows the user to paint obstacles with the cursor while holding the ‘a’ key (and erase obstacles when holding ‘s’ key) and performs the avoidance algorithm when the ENTER key is hit.



The algorithm finds all the radius ranges that do not cause any collision.

The orange areas are those out of the camera's field of view.

Fig 4.12 Avoidance Simulator

4.2 FUTURE SCOPE

This project has touched only the very basic of autonomous driving. Obstacle avoidance has been implemented as of now. This can further be developed to simultaneously localise and map the surroundings. This is a major step towards autonomous driving as having a map of the surrounding environment will enable proper path finding to the destination. It will be a long process as obtaining a data set of that length will take time.

Detection and processing of traffic light signals is another development that can be made. It will give the car enough intelligence to follow basic traffic rules. Lane maintenance also can be introduced. Other traffic rules depending on the country can be also be fed.

A long process forward, this project can be completely developed for autonomous driving, and a proper self-driving car can be obtained.

PERSONAL RESULTS

Personally this has been a tremendous learning experience. Starting off knowing nothing about Computer Vision, this was a huge domain. It took me some time to get used to the basics, and hence the start was slow. But once I could understand it all, it was a fascinating field. I have come to understand that any project/ learning curve is exponential, with slow beginnings, almost constant until a point, after which the curve shoots up. I have also learnt that regular reports of results is essential, doesn't matter if the results are positive or negative.

Overall, this was an amazing experience where I could improve my knowledge by drastic amounts.

HOW TO ACCESS THE CODE

- The code can be accessed by going to the CL-DUO3D-LIN-1.1.0.30/DUOSDK/Samples/OpenCV/Sample-01-cvShowImage.
- Build the files.
- The built files will be found in CL-DUO3D-LIN-1.1.0.30/DUOSDK/Samples/bin/x64
- Now run the application.
- To run the simulator, Open the Cmakelists.txt file and add PathPlaner, ObstacleScenario and Simulator to the sources, and build again.

REFERENCES

- [1] **Kyle Simek**, PhD, “Extrinsic camera parameters” Article.
<http://ksimek.github.io/2012/08/22/extrinsic/>
- [2] **Kyle Simek**, PhD, “Intrinsic camera parameters” Article.
<http://ksimek.github.io/2013/08/13/intrinsic/>
- [3] “Camera calibration and 3D reconstruction”. OpenCV documentation.
- [4] **Martin Peris** “3D reconstruction with OpenCV and Point Cloud library”. Personal blog article. <http://blog.martinperis.com/2012/01/3d---reconstruction---with---opencv---and---point.html>
- [5] **M. Peris** “OpenCV: Stereo matching”. Personal blog article.
<http://blog.martinperis.com/2011/08/opencv---stereo---matching.html>
- [6] **Marcos Nieto**, “Detection and tracking of vanishing points in dynamic environments”. PhD Thesis, Universidad Politécnica de Madrid, 2010.
<http://marcosnietoblog.wordpress.com/2012/03/31/vanishing---point---detection---c---source---code/>
- [7] **Marcos Nieto**, “Detection and tracking of vanishing points in dynamic environments”. PhD Thesis, Universidad Politécnica de Madrid, 2010.
<http://marcosnietoblog.wordpress.com/2012/03/31/vanishing---point---detection---c---source---code/>
- [8] **Alejandro Daniel Noel**, “Visual based guidance for autonomous vehicles”. Article, Universitat Autònoma de Barcelona, 2014.
http://futuretechmaker.com/projects/cv_autonomous_car