Fall 2014

Computer vision libraries for trailer truck testbed using open source computer vision libraries

Krishnan Raghavan

Department:

COMPUTER VISION LIBRARIES FOR TRAILER TRUCK TESTBED USING OPEN SOURCE COMPUTER VISION LIBRARIES

by

KRISHNAN RAGHAVAN

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

2014

Approved by

Dr. Levent Acar, Advisor
Dr. Jagannathan Sarangapani
Dr. Randy H. Moss

ABSTRACT

Computer Vision is a field that aims at understanding and analyzing images from the real world to produce numerical and symbolical data. It is a first step at duplicating the capabilities of human vision by electronically understanding the image and perceiving its features. This work aims at providing some of the features of a human eye to a trailer truck. These features include getting a 3D wireframe from continuous images and prediction of the next position of the objects in view, while the truck is moving.

The thesis has been divided into 3 sections. First section is acquiring images in real time. Second section is the preprocessing of images to achieve edges and points in the image, and the third section is converting the edges and the points into 3D wireframe and predicting of the next position of the image. OpenCV library and Point cloud libraries are used in this process to facilitate operations on 2D and 3D images.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Levent Acar for his guidance and for his continued support. I also want to thank Dr. Jagannathan Sarangapani and Dr. Randy H. Moss for serving in the committee.

I would especially like to thank Mr. Parth Desai, Mr. Arul Mathi Maran Chandran and Mr. Jason Hagerty for all the discussions and help.

I would like to thank my family and friends for their continued support and encouragement, that pushed me towards the completion of this project.

This research was supported by Department of Electrical and Computer Engineering Missouri University of Science and Technology.

TABLE OF CONTENTS

Page

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

The hardware setup is a model of a miniaturised trailer truck powered with motors and sensors with the ability to move autonomously. This trailer truck test bed was designed by Mr. Pravin Dhake in [1] and Dr. Robert Woodley in [2]. The main thrust of this work is to give the trailer truck some visual information about its environment. This information is gathered from a single camera . The camera provides with images, that are used to perceive the unknown dimension. A set of function libraries were developed which allows the user to develop control algorithms based on the visual information.

This test bed consists of a miniaturized trailer bed, a single board computer and software libraries such as the data acquisition library, the control library and a computer vision library. The system also used, open source image processing libraries like OpenCV and Point cloud to facilitate the input, output and certain low level routines in the work . These libraries have a dearth of documentation and support available that really facilitate real-time acquisition of images and their processing. The libraries are scalable and can be used in the control of the trailer truck.

There are many potential applications of the designed system including numerous military and automotive applications. Similar robustness and reliability requirements are desired in this system.

The thesis is divided into two sections. First section describes all the previous hardware, upgrades and the corresponding specifications, The second dection setals the software libraries and their dependencies along with the different functions available in the library and their usage.

2. HARDWARE

2.1. HARDWARE COMPONENTS

The previous hardware was setup by Mr. Pravin Dhake in [1] and Mr. Robert Woodley [2]. In their work, Woodley and Dhake put together a trailer truck with control and data acquisition libraries. This trailer truck has the ability to move automatically with the position given to it. In the course of this project the same truck is used but with updated components and libraries are developed to perform 3D modeling. Table 2.1 tabulates all the hardware used in the system.

Table 2.1. Overview of all the Hardware on the Truck

Name of the Part	Description
Controller Board	AAEON PFM-945C
Frame Grabber	Sensoray model 911
Storage for controller board	SSD-104 SATA
PCI-104 to PC-104 converter	Connect Tech PCI-104 to PC-104 Adapter
A to D Converter	AIM Multi-IO 104
Motor controller board	Mesa Electronics 4I27 Motor controller
Accelerometer	Dimension Engineering ACCM3D
Steering Motor	Futuba FP-S148
Driver Motor	RS-550 DC permanent magnet motor
Potentiometer	Bourns Potentiometer

2.1.1. Controller Board-AAEON PFM-945c. It is a low power PC-104 form factor based board, that is very suitable for our application because of its atom processor and its form factor, more details about the board are given in [3]. The board is supposed to run a real time operating system and should do both the image processing and the motion control functions. The board needs a 12V @1.5A power supply to function and is capable of handling both PCI-104 and PCI-express buses. This is an upgrade over the GX533 board which was used as part of Dhake's work in [1]. This board is powered by an atom

processor by Intel and is capable of running both Linux and Windows. Through extensive testing, it was found that the board runs very well with a stripped down version of Ubuntu 12.04. Figure 2.1 shows a picture of the board, and Table 2.2 lists the basic specifications of the board.

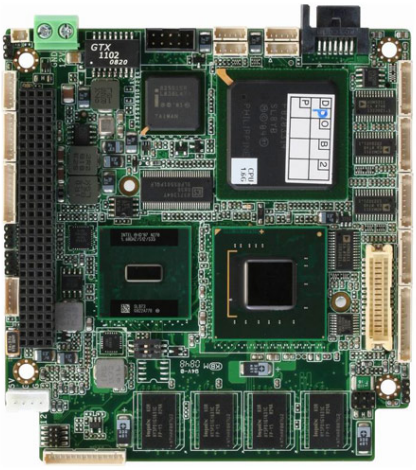

Figure 2.1. Controller

Table 2.2. Controller

Processor	Onboard Intel Atom N270 1.6 GHz Processor
Memory	Onboard DDR2 400/533 Memory 512 MB/ 1 GB
Chipset	Intel 945 GSE
Communication Interface	PCI-104 Express (PCI-104 + PCI-104e)
Board Size	4.05" (L) x 3.77" (W) (102.8mm x 96mm)
Power Requirement	1.56 A @ 12V

2.1.2. Frame Grabber Board-Sensoray Model 911. This is a PCI-express form factor based frame grabber, which captures four channels of analog video on a composite input [4]. The board supports 120 fps in NTSC and 100 fps in PAL settings. It supports capturing of raw frames, which can be formatted as RAW, or RGB frames. The software development kit provided by the manufacturer gives an idea about the code, that needs to be written to acquire frames from the camera and to write it onto the hard disk in real time.

The camera used in this system is connected through a composite to RCA connector cable. The board is connected to the camera using a 24-pin header, which connects to a terminal board 911TA. External power is not needed as all power is derived from the bus but can be supplied if needed using a four-pin Molex connector. Figure 2.3 shows a top view of the 911 board.

2.1.3. Storage Memory. A Solid-state Drive board is used for interfacing storage memory through PCI-104 bus . The board used is provided by Connect Tech, which provides a hard drive to be used over the PCI-104 bus. It does not require any external power supply, all power is provided by the bus itself. A 120 GB SSD from MUSHKIN is used. Figure 2.2 shows a picture of the board.

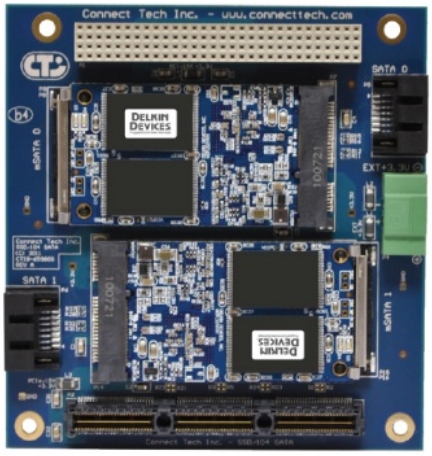

Figure 2.2. SSD Board

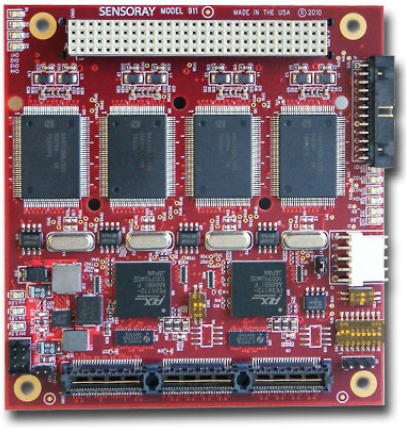

Figure 2.3. Framegrabber

2.1.4. PCI-104 to PC-104 Converter. This Interface board connects the old PC-104 form factor-based boards to the newer PCI-104 bus. This board is used to interface the older ADC and motor controller with the controller board. The API provided by Connect Tech is useful in interfacing the boards. Figure 2.4 shows a picture of the board.

2.1.5. A/D Converter. AIM104 multi IO is an 8-bit PC-104 module providing 8 Opto-isolated digital inputs and 2 analog outputs, that can be used as 16 single ended inputs or 8 differential inputs. The A/D card is connected to the controller board using the

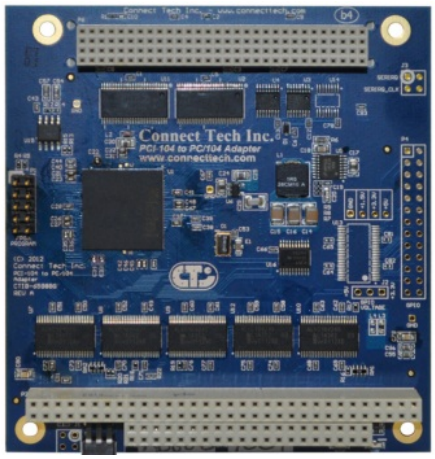

Figure 2.4. Converter Board

Figure 2.5. ADC Card

Connect Tech PC-104 to PCI-104 plus adapter. This A/D board is used to connect to the potentiometer, steering motor and accelerometer using interfacing circuits given in [1, 2], Figure 2.5 shows a picture of the board.

2.1.6. Motor Controller. The card used is a 4I27 motor controller card from MESA Electronics with a 7I25 H-Bridge driver to power and operate the motor. It is stacked on the PC-104 bus through the conversion adapter. This is a 2-axis DC servo motor controller that uses a PID filter to set to controller parameters. 4I27 is equipped with two LM629 processors, that can be set to control two separate motors separately without any intervention from the host computer. The 7I25 needs an external power supply to power the motors. The 7I25 is connected to the main 4I27 board using a 50-pin connector.

This assembly also consists of an optical shaft encoder, that gives input about the speed of the motor being driven by the H-Bridge driver. The motor feedback can be directly given to the 7I25 H Bridge. Figures 2.6 and 2.7, show each component separately.

2.1.7. Accelerometer. The accelerometer in the test bed is a Dimension Engineering DE-ACCM3D 3D analog accelerometer. It can measure static and dynamic accelerations. The accelerometer provides x, y and z outputs according to the voltages in the x, y

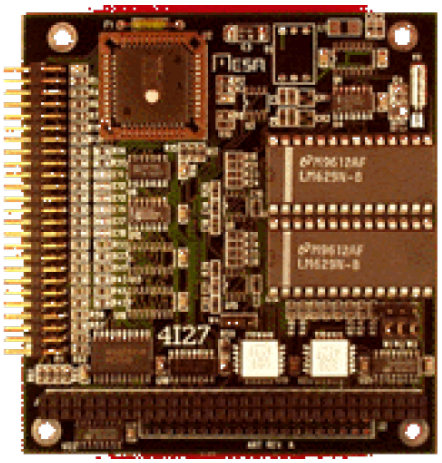

Figure 2.6. Motor Controller

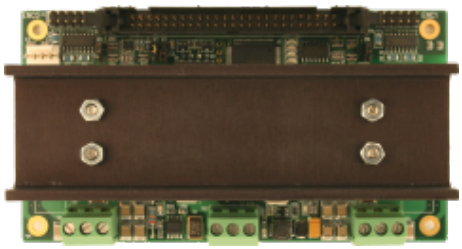

Figure 2.7. 7I25 H-Bridge

and z directions. The accelerometer is powered by an on board 3.3V voltage source. Figure 2.8 shows a picture of the board as given in [1, 2].

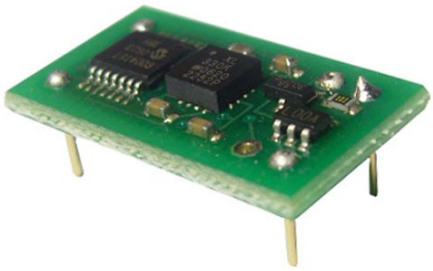

Figure 2.8. Accelerometer

Figure 2.9. Potentiometer

2.1.8. Potentiometer. The test bed vehicle consists of a 50K precision potentiometer from Bourns embedded in the socket between the trailer and the truck. This potentiometer is used to measure the trailer cab angle, which is critical to avoid jackknifing of the truck. Potentiometer uses a 5v DC power supply. Figure 2.9 shows a picture of the potentiometer.

2.1.9. Steering Motor. Steering motor used in this project is a DC servomotor, which has been modified to work with the current configuration as shown in [2]. The

Table 2.3. Partition of the OS

Partition	Size
Swap	3 GB
Boot	250 MB
Home	20 GB
/	56.75 GB
Store	40GB

FUTUBA FP-S148 is connected to the ADC from which it gets the signal for steering. Figure 2.10 shows a picture of the steering motor.

Figure 2.10. Steering Motor

Figure 2.11. DC Motor

2.1.10. Driver Motor. This is a Bane Bots permanent magnet DC brush motor. This motor has an optical shaft encoder connected to it and is controlled using the LM-629 motion controller through the H-bridge driver. The H-Bridge driver supplies power to the motor. Figure 2.11 shows a picture of the driver motor as shown in [1, 2].

2.2. HARDWARE CONNECTIVITY

Table 2.3 provides the partition of the storage memory during the operating system Installation. The Figure 2.12 lists the connected components.

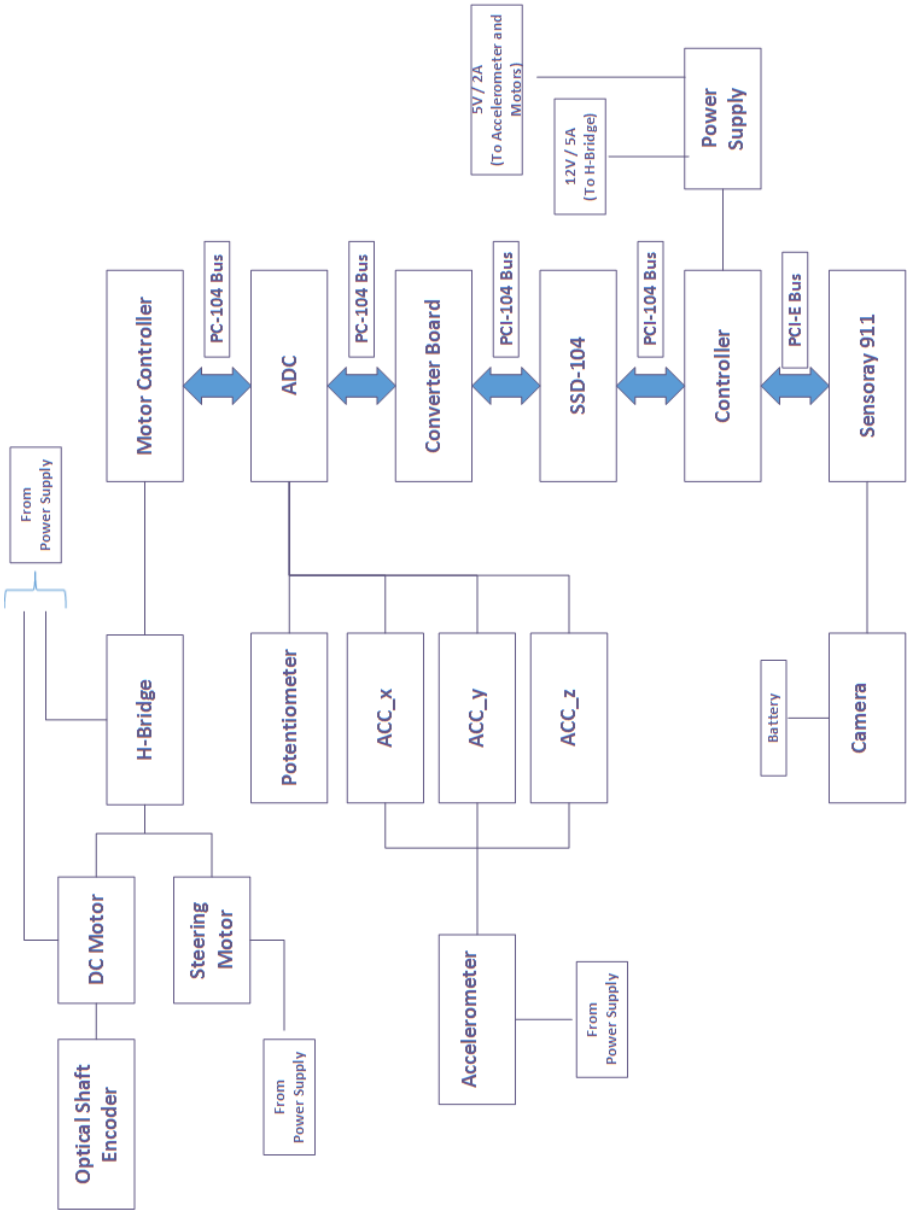

Figure 2.12. Block Diagram of the Hardware

Table 2.4. Overview of Hardware Update Times

Sensors	**Actuators**	**Hardware**	**Range(Data)/Capacity**	**Range(i/p - o/p)**	**Update Time across Port**
Truck Pot			12 bits across ADC	i/p - o/p 0-5V	500 microsecond
Steering Pot			12 bits across ADC	i/p - o/p 0-5V	500 microsecond
Shaft Encoder			Max frequency 200 MHZ	i/p - 5V	256 microsecond
Accelerometer			12 bits across ADC	i/p - o/p 1.33-1.66V	500 microsecond
Camera			130 fps	6V	500 millisecond
	Steering Motor			i/p PWM pulses from motor controller	256 microsecond
	Driving Motor			i/p PWM pulses from motor controller	256 microsecond
		ADC	-5-5V	12 bit	500 microsecond
		DAC	-5-5V	12 bit	320 microsecond

3. SOFTWARE AND INSTALLATIONS

3.1. OPERATING SYSTEM

The hardware is the most essential part of the system but it cannot work without the proper software installed on it. The controller board PFM-945c needs an operating system to function, the board supports various types of operating systems but one of the easiest and stable one is Ubuntu 12.04. A minimalistic version of the operating system that only has the command line interface was installed on the host computer. The building of the system is divided into three parts:

- Command Line Operating system installation.
- Installation of X11 and a desktop manager and the essential software component that are not available in the command line version.
- Installation of image processing libraries like OpenCV, PCL libraries, and the installation of drivers for the converter board.

3.1.1. Keypoints About Operating System Installation.

- Unity should not be used as a desktop manager. Unity requires an accelerated GPU, which is not available in our system. The main board has a 945M series graphical chipset [3].
- A complete install of Ubuntu results in an unbootable system. The installation starts running the boot script and then goes into an infinite loop. At that time, a command line mode can be used to uninstall unity and the default accounts manager and a combination of *gdm* and *openbox* or *xdm* and *xfce* software can be installed. Even though this method is successful, the final system turns out to be too heavy and needs

a lot of resources to run. A better approach is to perform a command line installation and install the software components that are needed, separately.

- Any version of operating system that needs 3D acceleration to run its GUI is not recommended.
- The disk burning software RUFUS is necessary to set up the USB disks, Other softwares may not work as the controller board may not recognize the created boot script.

3.2. REAL TIME APPLICATION INTERFACE (RTAI)

Real-time Application Interface (RTAI) is an extension for original Linux kernel, which enables real-time services. This is accomplished by RTAI tasks (RTASK) in kernel space or user space. Kernel tasks run in kernel space within kernel modules. User space tasks are user applications threads, which are switched to RTASKs with a given priority. Tasks can also be made periodic. The user space real-time framework is named LXRT. Installation instructions for installing a Real time Kernel and RTAI application are listed below [1, 6].

The program ”apt-get” is a Ubuntu maintained repository from where most of the needed packages and dependencies can be installed. The following commands needs to be executed in order to install an RTAI patched kernel on the system. All the commands with an arrow in front are to be run in a Linux terminal with root enabled.

- First step is to install all the dependencies. Installation is done by executing the following commands.

```
General:
--> sudo apt-get install cvs subversion build-essential git-core
--> sudo apt-get install g++-multilib gcc-multilib
```

```
Rtai:
--> sudo apt-get install libtool automake libncurses5-dev
--> sudo apt-get kernel-package
```

- Second step is installation of all the components. There are two options to perform this step. The first option is use a RTAI kernel from EMC and the second option is to build a kernel form scratch. If the hardware in the system is very specific, then its better to create a special kernel. In this system, a kernel was built from scratch by using the following procedures.

- Step 1: Downloading the required source code and patching the downloaded kernel.

```
--> sudo su
--> cd /usr/src/
--> cvs -d:pserver:anonymous@cvs.gna.org:/cvs/rtai co vulcano
--> sudo ln -s vulcano rtai
--> git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/
    linux-stable.git
--> cd linux-stable
--> git branch linux-3.8-rtai origin/linux-3.8.y
--> git checkout linux-3.8-rtai
--> git apply /base/arch/x86/patches/hal-linux-3.8.13-x86-4.patch
--> git commit -m 'applied rtai patch hal-linux-3.8.13-x86-4.patch'
--> exit
```

- Step 2: Downloading the newest debian package from the site

  https://launchpad.net/ canonical-kernel-team/+archive/ppa/+packages and saving it to /usr/src.

- Step 3: Making of the configuration file and building of the required debian packages.

```
--> sudo su
--> cd /usr/src
--> dpkg-deb -x linux-image-3.8.0-35-generic.deb linux-image
--> cp linux-image/boot/config-* linux-stable/.config
--> cd linux-stable
--> make menuconfig
--> make -j `getconf _NPROCESSORS_ONLN` deb-pkg LOCALVERSION=-rtai
--> exit
```

- Step 4: Installation of the created debian packages.

The RTAI installation guide is an important source for detailed instruction of the configuration of the kernel. Kernel installation is a delicate task and even a small error in the configuration may result in crashing the system. To select all the proper options an understanding of the hardware is needed. The command *make localmodconfig* could be used. This command goes through the current system and calculates modules that are currently active and accordingly makes the configuration file for the kernel. Installation of the built kernels can be done using the following commands.

```
--> cd /usr/src
--> sudo dpkg -i linux-image-3.8.13-rtai_3.8.13-rtai-1_i386.deb
--> sudo dpkg -i linux-headers-3.8.13-rtai_3.8.13-rtai-3_i386.deb
```

- Step 5: Rebooting the new installed RTAI-kernel.

```
RTAI (https://www.rtai.org)
(cvs-Version)
--> cd /usr/src
--> sudo cvs -d:pserver:anonymous@cvs.gna.org:/cvs/rtai co magma
--> sudo ln -s magma rtai
--> cd /usr/src/rtai
--> sudo make menuconfig

The directories are:
1. Installation: /usr/realtime/
2. Kernel source tree: /usr/src/linux-headers-3.8.13-rtai/

-->Under Machine, the number of CPU's in the machine could be chosen.
(Using cat /proc/cpuinfo, the number of processors could be verified)
-->The following commands can be used to install the path variables
for the newly installed kernel.
--> sudo make install
--> sudo sed -i 's/\(PATH=\"\)/\1\/usr\/realtime\/bin:/' /etc/
environment
```

Running the commands below for each open shell would setup the required software.

```
--> export PATH=/usr/realtime/bin:$PATH
```

The following commands insert modules into bash such that the application runs at startup.

```
/sbin/insmod /usr/realtime/modules/rtai_smi.ko
/sbin/insmod /usr/realtime/modules/rtai_hal.ko
```

```
/sbin/insmod /usr/realtime/modules/rtai_lxrt.ko
/sbin/insmod /usr/realtime/modules/rtai_fifos.ko
/sbin/insmod /usr/realtime/modules/rtai_sem.ko
/sbin/insmod /usr/realtime/modules/rtai_mbx.ko
/sbin/insmod /usr/realtime/modules/rtai_msg.ko
/sbin/insmod /usr/realtime/modules/rtai_netrpc.ko
/sbin/insmod /usr/realtime/modules/rtai_shm.ko
```

Using the commands above one can create a script, which can be used to load the modules at startup using the following command.

```
--> sudo chmod +x X.sh
```

After installation of the main components of RTAI, the next primary task is to install the image processing libraries, that are necessary for the purpose of processing images. The most important of them are the OpenCV and Point Cloud Libraries that are used in the course of the project.

3.3. OPENCV

Open Source Computer Vision library facilitates various image processing operations on images. OpenCV is released under a BSD license, that is free for both academic and commercial use. It has C++, C, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. OpenCV library was designed for computational efficiency and with a strong focus on real-time applications. Certain primary inbuilt functions, that have been used here in this project are explained below and more details about the library can be found in [7].

3.3.1. Mat. Class Mat consists of two variables, the matrix header and the data pointer. The matrix header consists of some primary details about the image like size, address of the image, etc. The data pointer points to the pixel data of the image. The Mat image container is a 2D matrix consisting of values between 0-255 for each channel of the image data namely R, G and B.

Example:

```
Mat Image;
```

3.3.2. Imread. This section defines a function to import images from the hard disk into the image container Mat, for this purpose OpenCV provides Imread function. Imread can read image from a file on the disk and loads it in the image container Mat. It supports multiple formats of images like BMP, TIFF, JPG, and PNG, etc.

Example:

```
Mat I = imread (Name of the image file in single quotes or a pointer
to the filename, format of the loaded image);
```

Format of the loaded image can be of three types:

- CV_LOAD_IMAGE_UNCHANGED loads the image as it is (including the alpha channel if present).
- CV_ LOAD_ IMAGE_GRAYSCALE loads the image as an intensity one.
- CV_LOAD_IMAGE_COLOR loads the image in the RGB format.

3.3.3. Imshow. Imshow helps to display the loaded image on the screen. The window on which the image may be displayed should be declared before using the function.

Example:

```
imshow (mat, " name of the window ");
```

3.3.4. Clone. It is used to copy the data from one image container to another.

Example:

```
img.clone();
```

3.3.5. Gaussian Blur. The operation of canny edge detection demands certain pre-processing operations on the image. One of them is actually blurring the image such that all the edges in the image are distinctly highlighted and there is reduction in noise. A type of blur that is used for this purpose is Gaussian blur. Here the image is multiplied by a gaussian kernel, and the resultant is smoothened or blurred image.

Example:

```
gaussianblur (source, destination, size of the kernel);
```

3.3.6. Canny Edge Detection. Canny Edge detection is one of the most popular methods in the field of edge detection. Canny requires that a Gaussian blurred image is provided. Canny uses hysteresis thresholding to find localized edges in an image [8].

Example:

```
Canny(source_image, detected_edges, lowThreshold, lowThreshold*ratio,
kernel_size );
```

Lowthreshold defines the high and low point of the hysteresis. More details about the various functions that are available can be found in [7].

Installation of OpenCV :

The following commands can be used to install OpenCV library in an Ubuntu based system. This can be done by creating simple script and executing it.

```
--> version="$(wget -q -O - http://sourceforge.net/projects/
opencvlibrary/files/opencv-unix | egrep -m1 -o '\"[0-9](\.[0-9])+' |
cut -c2-)"
--> echo "Installing OpenCV" $version
--> mkdir OpenCV
--> cd OpenCV
--> echo "Removing any pre-installed ffmpeg and x264"
--> sudo apt-get -qq remove ffmpeg x264 libx264-dev
--> echo "Installing Dependenices"
--> sudo apt-get -qq install libopencv-dev build-essential
                  checkinstall cmake pkg-config yasm
                  libjpeg-dev libjasper-dev
                  libavcodec-dev libavformat-dev
                  libswscale-dev libdc1394-22-dev
                  libxine-dev libgstreamer0.10-dev
                  libgstreamer-plugins-base0.10-dev
                  libv4l-dev python-dev python-numpy
                  libtbb-dev libqt4-dev
                  libgtk2.0-dev libfaac-dev
```

```
                libmp3lame-dev libopencore-amrnb-dev
                libopencore-amrwb-dev libtheora-dev
                libvorbis-dev libxvidcore-dev
                x264 v4l-utils ffmpeg
--> echo "Downloading OpenCV" $version
--> wget -O OpenCV-$version.zip http://sourceforge.net/projects/
opencvlibrary/files/opencv-unix/$version/opencv-"$version".zip/
download
--> echo "Installing OpenCV" $version
--> unzip OpenCV-$version.zip
--> cd opencv-$version
--> mkdir build
--> cd build
--> cmake -D CMAKE_BUILD_TYPE=RELEASE -D
    CMAKE_INSTALL_PREFIX=/usr/local -D WITH_TBB=ON -D
    BUILD_NEW_PYTHON_SUPPORT=ON -D WITH_V4L=ON -D
    INSTALL_C_EXAMPLES=ON -D INSTALL_PYTHON_EXAMPLES=ON -D
    BUILD_EXAMPLES=ON -D WITH_QT=ON -D WITH_OPENGL=ON..
--> make -j2
--> sudo checkinstall
--> sudo sh -c 'echo "/usr/local/lib" >/etc/ld.so.conf.d/opencv.conf'
--> sudo ldconfig
```

3.4. POINT CLOUD LIBRARIES

This library is for 3D image processsing. The Visualization component of the library has been used in the course of this project. The installation of this library can be done using a prebuilt binary that is available for download from PCL website http://pointclouds.org/. Another method is to use the following commands to install point cloud library on any Ubuntu based system.

```
--> sudo add-apt-repository ppa:v-launchpad-jochen-sprickerhof-de/pcl
--> sudo apt-get update
--> sudo apt-get install libpcl-al
```

4. STORAGE CLASSES

This section details the different storage classes available in the system. The storage classes are designed to give efficient and fast storage in the system. The definitions are given below.

4.1. GLOBAL VARIABLES AND DATA STRUCTURES TO STORE DATA

4.1.1. Storage Container for the Images. OpenCV defines Mat [Section 3.3] as the default storage container for images. This custom storage container consists of a variable of type Mat, which is an OpenCV defined storage class for images. The container also consists of a variable defining the total number of points detected in the image.

Synopsis:

```
struct ImageMat{
      Mat img;
      int number_points_detected;
      ImageMat(){
       number_points_detected=0;
      }
};
```

Example:

The method to define an object of the above defined structure is given below.

```
main(){
    .
    .
    .
    ImageMat I;
    .
    .
    .
 }
```

4.1.2. Storage Container for Point. The storage of a point consists of a structure named coordinate and a *structure* named *RGB*. The structure consists of a variable of type *time_t*. The variable *time_t* defines current time at which a point is detected in the image. To track the points between corresponding frames and the successive images, a *label* is introduced. Figure 4.1 shows the storage structure.

Synopsis:

```
struct point{
    string label;
    time_t t;
    coordinates P;
    RGB C;
};
```

Example:

The method to define an object of the above-defined structure is given below.

```
main(){
    .
    .
    .
    point P;
    .
    .
    .
}
```

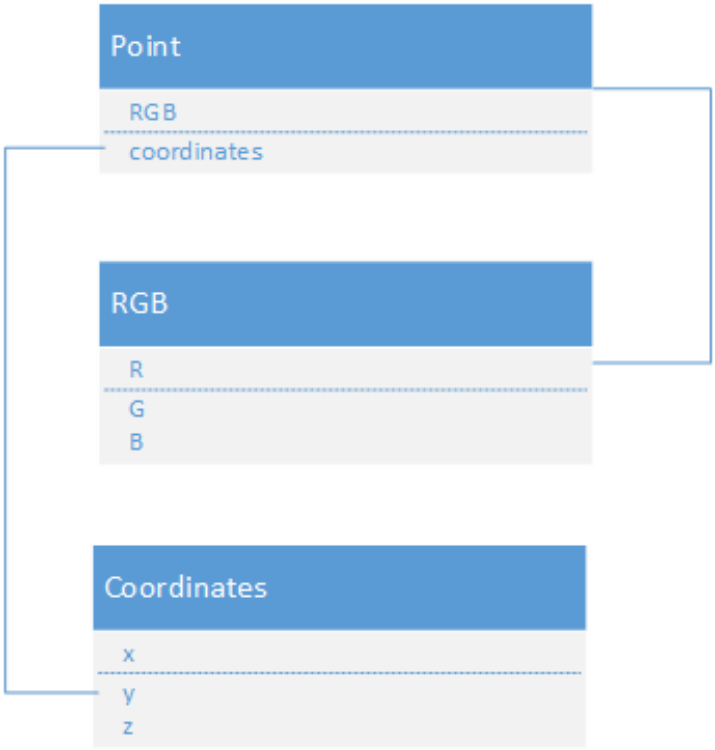

Figure 4.1. Structure point definition

4.1.3. Storage Container for Color. This container consists of three variables named *R, G and B* corresponding to the three color parts of an image. Each of these has a range from 0-255, corresponding to each channel in the image as shown in Figure 4.2.

Synopsis:

```
struct RGB{
        int R;
        int G;
        int B;
    RGB (){
        R=0;
        G=0;
        B=0;
    }
};
```

Example:

The method to define an object of the above-defined structure is given below.

```
main(){
    .
    .
    .
    RGB R;
    .
    .
```

```
    .
}
```

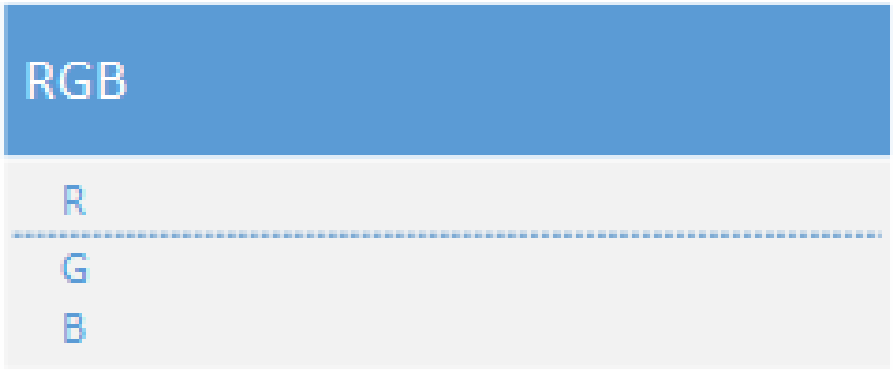

Figure 4.2. Color

4.1.4. Storage Container for Coordinates. This data structure defines three variables *x, y and z* which are the three coordinate values of a point in space. This can be used as a vector to store coordinates of any point in space. These variables are declared as float, so they can record different types of values as shown in Figure 4.3.

Synopsis:

```
struct coordinates{
    float x;
    float y;
    float z;
    coordinates(){
        x=0.0;
        y=0.0;
        z=0.0;
    };
```

Example:

The method to declare a structure is given below.

```
main(){
    .
    .
    .
    coordinates R;
    .
    .
    .
}
```

Figure 4.3. Coordinates

4.1.5. Storage Container for Lines. A structure containing an object of type points is used to store lines, consists of two variables, one that stores the vertices of the line and the other one points to the next vertex in the image. In a chain of points , if a point is found, that is same as the starting point then it describes a closed polygon otherwise a piecewise straight line.

Synopsis:

```
struct lines{
    point vertex;
    lines *next;
};
```

Example:

The method to declare a structure is given below.

```
main(){
    .
    .
    .
    lines *L=new lines();
    .
    .
    .
}
```

4.2. PERMANENT STORAGE

Permanent storage of data is very important, and the retrieval and storage should take minimum time. For this purpose four structures are defined as part of the system. There are structures respectively for storing images; point; lines and motion. All the four structures can be accessed using a key, which is defined by the user. All the keys, that are used for storage, are also stored in a variable array, and the user can read, edit and remove

the keys anytime using the functions provided in the library. The usage and definition of these structures and functions are given below.

4.2.1. Maps/Associative Array Structure to Store an Image. This structure consists of three variables, one is an associative array for the storage of image, and the other two variables are the storage for keys, which are used to store data. The primary key is a user specifiable string. The second or the inner key is time. An image may be stored at different times, giving the user the ability to track the behavior of an image at different times as shown in Figure 4.4.

Synopsis:

```
struct I_List{
map < pair<string, time_t>, ImageMat> Images_List;
std::vector<std::string> keyouterImages;
vector<time_t> keytimeImages;
};
```

4.2.2. Maps/Associative Array Structure to Store Points. The permanent storage of points is done in this associative array. The primary key is a string that is specified by the user, this string can also be the label of a point in the image. The second or the inner key is time. A certain kind of point may be stored at different times, giving the ability to track the behavior of the point at different times as shown in Figure 4.4.

Synopsis:

```
struct P_List{
map < pair<string, time_t>, point > Points_List;
vector<string> keyouterPoints;
```

```
vector<time_t> keytimePoints;
};
```

4.2.3. Maps/Associative Array Structure to Store Lines. A structure similar to the one presented in Section 4.2.2. is used to store lines. An associative array to store the lines and two other arrays to store the keys is also provided in the system as shown in Figure 4.4.

Synopsis:

```
struct L_List{
map < pair<string, time_t>, lines* > Lines_List;
vector<string> keyouterLines;
vector<time_t> keytimeLines;
};
```

4.2.4. Maps/Associative Array Structure to Store Camera Motion. The permanent storage of camera motion is done in this associative array. The primary key is the time. The structure is similar to the one in section 4.2.2. It has one other variable to store the key as shown in Figure 4.4.

Synopsis:

```
struct M_Lidst{
map < time_t, Matx34d > TrackMotion;
vector<time_t> keytimemotion;
};
```

There are functions provided in the library, that can be used for storing data using the associative array and also can be used for editing and removing the keys from the array. The description about these functions is given in Low Level function Section. Figure 4.4 explains the storage of data in an associative array.

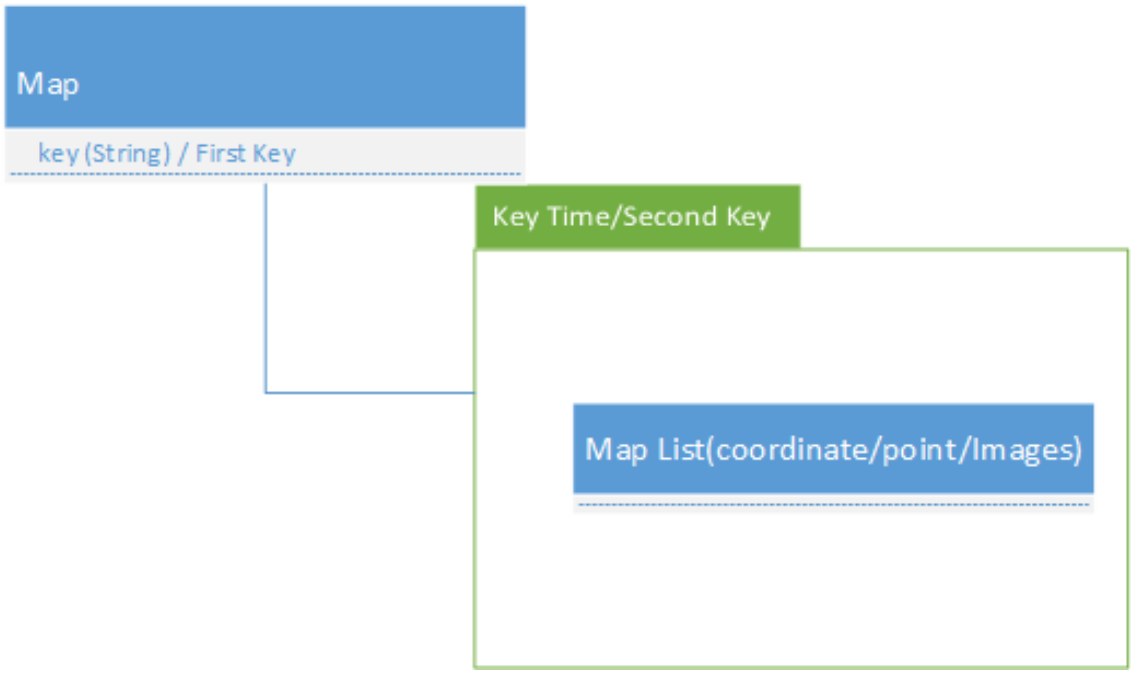

Figure 4.4. Map Memory Storage

4.2.5. Maps/Associative Array Structure to Store Coordinates_List. This list is to keep track of all the points that are detected in the images. Whenever a point is labeled or found, they are stored in this array. When a point is detected, it is compared with the points in this list and if a close proximity is established, then the points in the proximity are labeled the same.

Synopsis:

```
std::vector<point> coordinates_list;
```

4.2.6. Maps/Associative Array Structure to Store TrackMotion. This list keeps track of all the motion. The camera motion matrix, that is the transformation between, the two positions of the camera, is stored in this array.

The first entry in this matrix is the initial projection matrix, that is the transformation between the image plane and the real world plane. This matrix is tracked with respect to time.

Synopsis:

```
map < time_t, Matx34d > TrackMotion;
```

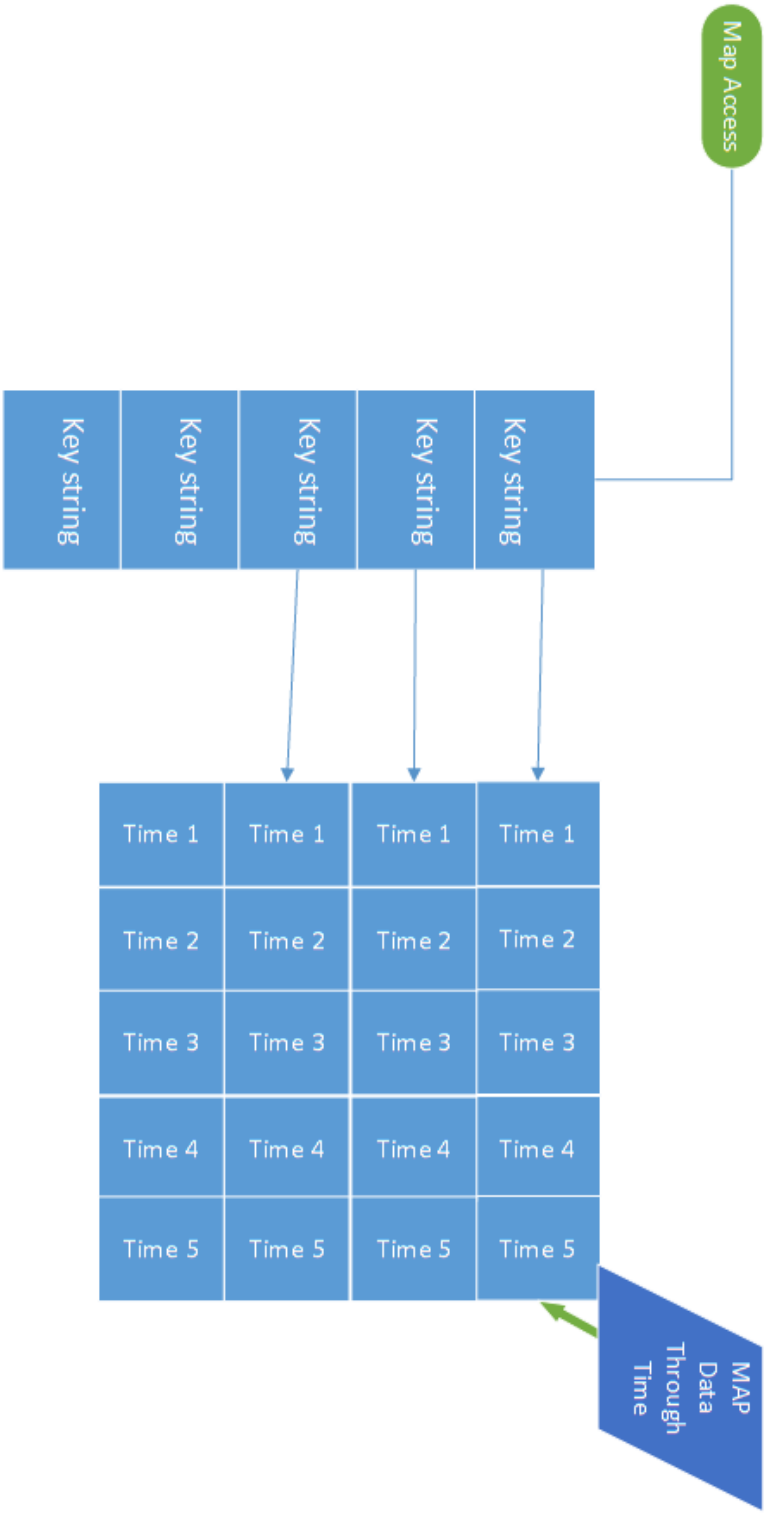

Figure 4.5. Associate Memory Access

5. COMPUTER VISION LIBRARIES

The libraries installed in this system are:

- The hardware motion libraries that were developed by Pravin as shown in [1, 10]
- Computer Vision Libraries.

This section overviews the Computer Vision Libraries, and it is divided into two subsections. The first section presents the details of the wireframe libraries, and the last section is about the 3D estimation and visualization libraries.

5.1. WIREFRAME LIBRARIES

Defined below are the different functions, that can be used to generate a wireframe from an image.

5.1.1. Image Acquisition. This Library function is to acquire an image from the camera. Depending on user's choice, the image can be stored either on *SSD* or on the *Memory*. The function can also be used to peruse a previously stored image on the disk. For both cases, a filename and location choice need to be specified.

Synopsis:

```
ImageMat *ImageAcquisition(int choice, string filename, string key,
I_List &I_object);
```

Example:

To use the function defined above the location choice and the filename needs to be specifieds.

```
main(){
    .
    .
    .
    int choice;

    cout<< "Enter choice for data acquisition"<< endl;
    cout <<"choice =1 (Stores into SSD with the
           filename of your choice)"<<endl;
    cout <<"Choice =2 (Stores into matrix (Image_mat)"<<endl;
    cin >> choice;
    string filename;
    if(choice==1){
    /* enter filename */
        gets(filename);
    }
    /* define key by the user */
    Iinp= ImageAcquisition(2, s1, s1, I_object);
    .
    .
    .
}
```

5.1.2. Thresholding. This function could be used for thresholding the image. This function consists of two parameters *Cthreshold* and *RGBparameter*. Cthreshold is an object of structure *coordinates* having x,y, and z values. RGBparameter is an object of structure

RGB storing RGB threshold values. Currently only thresholding of RGB is provided but the user can add a threshold for points. Refer to Figure 5.1.

Synopsis:

```
ImageMat *thresholdRGB(ImageMat* Iparameter, coordinates Cthreshold,
RGB RGBthreshold, string key, I_List I_object);
```

Example:

```
main(){
.
.
.
Iinp = ImageAcquisition(2, s, s, I_object);
ImageMat *Ithresh = thresholdRGB(Isrc, Cthreshold, RGBthreshold, key,
I_object);
.
.
.
}
```

5.1.3. Preliminary Edge Detection. This function detects the edges of the image; The edge detection is preliminary and results in edges that are serrated and not continuous. It is useful in circumstances where the outline of the contents is needed. If a wireframe from the image is desired, preliminary edge detection must be performed. The algorithm is explained in Figure 5.2.

Synopsis:

```
ImageMat *PrelimEdgeDetection(ImageMat *Isrc, string key, I_List &
I_object);
```

Example:

```
main(){
.
.
.
Iinp = ImageAcquisition(2, s, s, I_object);
ImageMat *Iinp= PrelimEdgeDetection(Iinp, "OriiginalImage", I_object);
.
.
.
}
```

5.1.4. Vertex Detection. This function detects the sure edges of the image, detects the vertices, and combines these vertices to form lines. Prerequisite is that an edge detected image is obtained from prelimedgedetection. The function returns all the points that were detected.

Synopsis:

```
ImageMat *VerticesDetection(ImageMat *I, string key, P_List &P_ob,
L_List &L_ob, I_List &I_ob);
```

Example:

An example is shown below. The keys that are required to store the points in an associative array needs to be specified. If the keys are not to be permanently be stored then the string '*no*' needs to be sent.

```
main(){
.
.
.
ImageMat *Iinp= PrelimEdgeDetection(Iinp, "OriginalImage", I_object);
Iinp = VerticesDetection(Iinp, "VERTICESIMAGE", P_object, L_object,
I_object);
.
.
.
}
```

The above functions can be used to generate a wireframe from an image. The process involves finding the edge and the vertices/hull points from an image. These points can be stored at the user's discretion. The user can specifies the value of keys to each of the function to store the data. There are functions provided for the retrieval and removal of keys and the stored data.

The next set of library functions can be used to generate a 3D model from the set of images. A user can use both wireframe and a plain image to create the model. It is advisable that they use the input image to find the various parameters needed to convert the model as it results in increased accuracy but results in decreased speed, hence a tradeoff must be made. While using plain images one needs to convert the image into gray scale.

5.2. VISUALISATION AND MOTION LIBRARIES

The next set of functions would aim at reconstructing a 3D model from a pair of images. A typical system for the construction of 3D models consists of a stereo rig meaning two-fixed camera, where the relative position of the two cameras are known. This operation takes place in three phases. In the first phase, a set of matched points i.e. pixels in the two views that are the images of the same point in the real world are established between the two images. In the second phase, the identified matched points are used to derive the relative locations, orientations and other parameters of the cameras. This process usually requires iterative solution of a set of non-linear equations. In the third phase the locations of 3D points are computed. In the current scenario, the modelling is done using a single camera; hence an attempt is made to simulate the stereo vision by moving the camera. This technique is known as structure from motion and is a popular technique in situations where there is no calibrated stereo rig to emulate the left and the right eye. Traditional stereoscopy systems, such as human visual system, utilize multiple viewing angles of the same object in order to do triangulation and get a depth perception, which is the primary objective.

The methodology of Visual Structure from Motion VSFM is described in [11, 12]. The retrieval of 3D model from images is basically divided into a few steps:

- Point matching and feature extraction,
- Estimating the camera motion from the pair of images,
- Point triangulation,
- Display visualizer.

5.2.1. Step 1: Feature Extraction and Matching. The first algorithmic step is *point matching* and *feature extraction*. The main goal here is the matching of one point in one image to the corresponding point in the other image. During this matching, there are

several tasks that the algorithm has to perform. First, it has to compare the epipolar lines of the images pixel by pixel. For every pixel on one line, the counterpart on the corresponding epipolar line in the other image needs to be identified. After the matching has been done it is possible to calculate how much the pixels have moved in the corresponding images.

Another method does not involve feature matching. In this method, the optical flow is calculated. The result is the overall flow in the image, but when the motion is large, and the features move substantially in the image, optical flow may fails because pixel movement is usually confined to a search window. To overcome this problem a hybrid method is implemented in which feature matching is done and these pairs are used as an input flow to calculate the overall motion map. In this method features are extracted first. There are two methods provided by OpenCV library to extract image features. The first method is called *SIFT*, *Scale Invariant Feature Transform* and the second method is *SURF*, *Speeded Up Robust Features*. Even though points can be detected through the vertices detection function mentioned in the previous section, there is a necessity for recovering the orientation and the details about the neighbors from the image. For this purpose descriptors are neeeded to be detected and recorded from the image. The following section explains the details of the extraction and registration of features.

The *SIFT* algorithm can be broken down into four main stages: (1) scale-space peak selection; (2) point localization; (3) orientation assignment; and (4) point descriptor. The first stage is to search for the points of interest over location and scale. The image is constructed in a Gaussian Pyramid, where the image is down sampled and blurred at each level. These blurred images at each level are used to compute the Difference of Gaussians (DoG), that locates the edges and the corners within an image. Points of Interest are then extracted in stage 2 by locating the maxima/minima pixels within the different scales of the DoG at the sub-pixel accuracy. Once points of interest have been located, an orientation is assigned based on the gradient orientation of the pixels around the point as in [13].

The *SURF Extraction* method approximated the Laplacian of Gaussian with the Difference of Gaussian for finding the invariant features, the same variables are approximated by SURF using the Laplacian of Gaussian with the Box Filter. One big advantage of this approximation is that, the convolution with box filter can be easily calculated with the help of integral images. And it can be performed in parallel for different scales. Also, the SURF algorithm relies on determinant of the Hessian matrix for both scale and location. SURF is a local invariant point of interest detector-descriptor, similar to its predecessor, the accurate but slower SIFT algorithm. In this thesis SURF algorithm is utilized due to its unique rich descriptors and relatively quick processing time. The mathematics and more details refer to [14].

Both of these algorithms are robust and accurate for determining the features in an image. In this thesis, these algorithms are used to find out the surrounding and neighbors of each special corner in the image. SURF is used for detecting the points, and SIFT is used for defining the descriptors of the same point. *descriptors* belong to an OpenCV provided class to store the surroundings and the orientation of the Gaussian. Experiments showed that SIFT is slower but more effective in finding descriptors but SURF is more suitable in finding features quickly. Hence the implementation is a combination of both. The points that were detected initially are first filtered by the SIFT algorithm. Some previously detected points may be unsuitable for matching and such points are rejected by the SURF algorithm and the remaining points are used for matching the corresponding points between the two images.

Once features are extracted and their descriptors are calculated, the next step is to match the points and get the relative motion between the points that provides an initial estimation to be used in calculating the overall optical flow. The function, which performs the operations mentioned above, is given below.

Motion Map

Description

After points in each frame are detected, disparity between two consecutive images needs to be determined. To determine this disparity two images at two different time instances are provided to the motion map function. This function will calculate optical flow between these images. Optical flow shows where the specific pixels have moved in images, which gives an idea about disparity, which is needed to calculate the third dimension. The user needs to decide which image will be given to the function. The user has the choice of using detected points or standard images. In situations where both the images are actually same the function will give an error message and proceeds forward to the next set of images if the operation is looping as shown in Figure 5.7.

Synopsis:

```
int CalculateMotionMap(Mat img_1,
                       Mat img_2,
                       vector<KeyPoint> &keypoints_1,
                       vector<KeyPoint> &keypoints_2,
                       vector<KeyPoint>& fullpts1,
                       vector<KeyPoint>& fullpts2,
                       vector<KeyPoint> &imgpts1_good,
                       vector<KeyPoint> &imgpts2_good,
                       vector <DMatch> &matches);
```

Example:

The parameters *img_1* and *img_2* are the two images, that are the input to this function. The points, *keypoint_1* and *keypoint_2* are the two sets of points that are detected in the image the points.The parameters *imgpts1_good* and *imgpts2_good* are the two sets of matched points that are detected in both images. The parameter *fullpts* are all the points which are detected in both the images. The variable Matches provides the corresponding point in the second image for a point in the first image. There are two functions in the library, that convert the library detected points into the keypoints of OpenCV. These functions can be used to convert the user-detected points and send them as parameters to the motion map.

```
main(){
.
.
.

ImageMat *Iinp = new ImageMat;
ImageMat *Iinp1 = new ImageMat;

string s = "4.jpg";
Iinp = ImageAcquisition(2, s, s, I_object);

string s1 = "5.jpg";
Iinp1 = ImageAcquisition(2, s1, s1, I_object);

Iinp= PrelimEdgeDetection(Iinp, "EdgeImage1", I_object);
```

```
Iinp = VerticesDetection(Iinp, "VERTICESMAGE1", P_object, L_object,
I_object);

Iinp1= PrelimEdgeDetection(Iinp1, "EdgeImage2", I_object);
Iinp1 = VerticesDetection(Iinp1, "VERTICESMAGE2", P_object, L_object,
I_object);

/* Wireframe */

vector<KeyPoint> pt1,pt2;
vector<KeyPoint> imgpts1;
vector<KeyPoint> imgpts2;
vector<KeyPoint> imgpts1_good;
vector<KeyPoint> imgpts2_good;
vector<DMatch> match;
int sp = CalculateMotionMap(Iinp->img,
                            Iinp1->img,
                            imgpts1,
                            imgpts2,
                            pt1,
                            pt2,
                            imgpts1_good,
                            imgpts2_good,
                            match);
.
.
```

```
.
}
```

5.2.2. Step 2: Camera Callibration and Estimation of Camera Matrices. When working with monocular camera, the spatial relation between the two positions of the camera needs to be known. This relation is given as a rotation and a translation matrix, that provides the transformation between the first and the second positions of the camera. One method to calculate the matrix involves Scalar Vector Decomposition of Essential matrix as described by Hartley and Zisserman in [15].

The primary prerequisite is to know the internal parameters of the camera, which are basically the focal length, and length distortion parameters. They are characterized by matrix K known as the camera callibration matrix and can be calculated using an algorithm proposed by Tsai in [16]. Using this algorithm, the process of calculating the camera calibration matrix can be automated by providing the camera with images of different orientations of a chessboard pattern. The resulting matrix is known as the camera projection matrix. This matrix provides the conversion between, the world coordinate and the camera coordinate frames.

A simple contrast-based algorithm can recognize the black-white intersections in the chessboard squares. The application of the algorithm generates the camera's internal matrix. There is a camera calibration function provided in the library that makes use of the chessboard pattern to detect all the internal parameters of the camera.

To calculate the unknown dimension of a point in the image, a tansformation called Fundamental Matrix needs to be determined. It is the algebraic representation of epipolar geometry as explained in [15]. To elaborate, the epipolar geometry is the intrinsic projective geometry between two view or images of the same scene. It is independent of scene structure, and it only depends on the camera's internal parameters and relative position and

orientation. The fundamental matrix F encapsulates this intrinsic geometry. It is a matrix of size 3×3 and rank 2. Defined below are some definitions as given in Hartley and Zissermann [15], that elaborate more on the details of the epipolar geometry.

- An epipole is the point of intersection of the line joining the camera centers also known as baseline, with the image plane.
- The epipolar plane is a plane containing the baseline.
- An epipolar line is the intersection of an epipolar plane with the image plane. All epipolar lines intersect at the epipole. An epipolar plane intersects the two image planes in epipolar lines, and defines the correspondence between the lines.

A particular point captured by a camera has a two dimensional representation in the camera coordinate system described by

$$x = \begin{bmatrix} x1 \\ x2 \\ 1 \end{bmatrix}$$

The same point described in a different image would have different values described by

$$x' = \begin{bmatrix} x1' \\ x2' \\ 1 \end{bmatrix}$$

The fundamental matrix relates the two points in such a way that $x^T F x' = 0$ as described in [15]. In [15, 17] the methods to derive the Fundamental matrix are explained. In Figure 5.6, C represent the camera centers. The point X is the point in the environment which is pictured by camera in two views as x and x'. The line joining the two camera centers is the baseline and the plane containing the points X, x and x' is denoted by π.

It is shown in Figure 5.6 that the points x, x' and X are coplanar and lie in plane π. It can also be seen in the figure that the rays from the camera centers of both the images intersect at X and the rays are also coplanar.

Hence given a pair of images, it can established that for each point x in one image, there exists a corresponding line l' in the second image of the same scene, that is the line connecting camera center and the world point. A point in the second image corresponding to the point in the first image must lie on the epipolar line as the epipolar line is the projection in the second image of the ray joining point x and the camera center C of the first image. Thus, there exists a mapping from a point in one image to its corresponding epipolar line in the other image. F, the fundamental matrix encapsulates this information OpenCV provides functions to iteratively find the Fundamental Matrix as explained in [15].

For a callibrated camera, a normalized coordinate system provides more accurate mapping information, In this case, a specialisation of the Fundamental matrix known as Essential matrix is utilized. This matrix can be expressed as

$$E = K'^{T} F K$$

where F is the Fundamental matrix and K is the camera callibration matrix.

Once the essential matrix is known, the camera matrices defining the transformation may be retrieved from E. In contrast with the fundamental matrix case, where there is a projective ambiguity during reconstruction, the camera matrices may be retrieved from the essential matrix up to scale and a four-fold ambiguity, meaning there are four possible solutions, except for overall scale, which cannot be determined.

The projection matrix P, that can be derived from the essential matrix is the transformation between the relative positions of the camera but include the effect of the camera callibration matrix. It is given as the product of $P = K[R|t]$ where $[R|t]$ is given by

$$\begin{bmatrix} \cos\theta & \sin\theta & Tx \\ -\sin\theta & \cos\theta & Ty \\ 0 & 0 & 1 \end{bmatrix}$$

So if the calibration matrix K is known, then the matrix $K^{-1}P$ is called the normalized camera matrix, the effect of the known calibration matrix having been removed. The resulting projection matrix is the the transformation between the cameras with normalized coordinates and is used to predict the position of the camera. For this purpose the normalized matrix is saved at each motion sequence to track the object motion from the initial reference point which is assumed to be identity.

The function described below is provided in the library to calculates the required camera matrices.

Find Camera Matrices

Description

This functions calculates both the essential matrix and the Fundamental matrix.

Synopsis:

```
bool FindCameraMatrices(const Mat& K,
                        const Mat& Kinv,
                        const Mat& distcoeff,
                        const vector<KeyPoint>& imgpts1,
                        const vector<KeyPoint>& imgpts2,
                        vector<KeyPoint>& imgpts1_good,
                        vector<KeyPoint>& imgpts2_good,
                        Matx34d& P, Matx34d& P1,
                        vector<DMatch>& matches,
                        vector<CloudPoint> &outCloud);
```

Example:

The variables *K* and *Kinv* are the camera internal parameters as provided by the callibration of the camera.

```
main(){
.
.
.
v::Mat K, Kinv ;
cv::Mat cam_matrix, distortion_coeff;
std::vector<CloudPoint> pointcloud;
std::vector<cv::KeyPoint> correspImg1Pt;
```

```
cv::FileStorage fs;


fs.open("out_camera_data.xml",cv::FileStorage::READ);
fs["Camera_Matrix"]>>cam_matrix;
fs["Distortion_Coefficients"]>>distortion_coeff;
K = cam_matrix;
invert(K, Kinv); //get inverse of camera matrix
cout << Kinv<<endl;
int a;
cout<< "cam matrixes"<< K<< endl;


vector<KeyPoint> pt1,pt2;
vector<KeyPoint> imgpts1;
vector<KeyPoint> imgpts2;
vector<KeyPoint> imgpts1_good;
vector<KeyPoint> imgpts2_good;
vector<DMatch> match ;


Matx34d P = cv::Matx34d(1,0,0,0,
                        0,1,0,0,
                        0,0,1,0);
Matx34d P1 = cv::Matx34d(1,0,0,50,
                         0,1,0,0,
                         0,0,1,0);
```

```
int sp = CalculateMotionMap(Iinp->img,
                            Iinp1->img,
                            imgpts1,
                            imgpts2,
                            pt1,
                            pt2,
                            imgpts1_good,
                            imgpts2_good,
                            match);


bool s = FindCameraMatrices( K,
                        Kinv,
                        distcoeff,
                        imgpts1,
                        imgpts2,
                        imgpts1_good,
                        imgpts2_good,
                        P,
                        P1,
                        matches,
                        &outCloud);
.
.
.
}
```

5.2.3. Step 3: Triangulation and Image Reconstruction. After the camera matrices are calculated, the next step in the process is to calculate the unknown dimension of the image, this calculation is done by triangulation of the corresponding points and the camera location. The linear triangulation method is the most common one, described, for instance, in [15].

The method involves calculating the unknown dimension by searching for the best possible solution which satisfies the fundamental matrix relation of the corresponding points. This search is reduced by knowledge of the epipolar geometry of corresponding points. The point in the 3D space must lie on the intersection of the line passing through the camera center and the point on the image plane for both the images. The result of this search is the identified 3D space coordinate points.

The end solution on triangulation is the third dimension of a point corrected upto a scale factor. The scale factor error can be removed but it needs a set of ground control points. This would require extra hardware and is not feasible as part of this project. At the end of this step points with a third dimension detected and stored in a point cloud. It is a collection of the points, computed in the earlier steps, displayed in an OpenGL implemented viewer.

Triangulate Points

Description

This function calcualtes the 3rd dimension and it is stored as part of the point structure defined in this system. Figure 5.8

Synopsis:

```
double TriangulatePoints(const vector<KeyPoint>& pt_set1,
                         const vector<KeyPoint>& pt_set2,
                         const Mat& K,
                         const Mat& Kinv,
                         const Mat& distcoeff,
                         const Matx34d& P,
                         const Matx34d& P1,
                         vector<CloudPoint>& pointcloud,
                         vector<KeyPoint>& correspImg1Pt);
```

In this function, *pt_set1* and *pt_set2* are all the points detected in the images. *K* and *Kinv* are the camera callibration matrices . The variable pointcloud is a vector array in accordance with the point cloud library that is used in this system to visualize the 3D model. This function is called from inside the FindCameraMatrices function which gives the point cloud as output, but if the user needs to call this function from main, it could be done after calling FindCameraMatrices function.

5.3. FUNCTIONS FOR DISPLAY, STORAGE AND OTHER PRIMARY LOW LEVEL FUNCTIONS

Convert to Grayscale

Description

Image input in OpenCV is generally three-channel image (RGB) and this function can be used to convert into grayscale.

Synopsis:

```
ImageMat *imageconvertgrayscale(ImageMat *I);
```

Retrieve Image from Storage

Description

This function retreives image from the permanent storage using the time key and the string key.

Synopsis:

```
ImageMat retreiveimages(I_List &I, time_t t, string key);
```

Retrieve Lines from Storage

Description

This function retreives lines from the permanent storage using the time key and the string key.

Synopsis:

```
lines *retreivelines(L_List &L, time_t t, string key);
```

Retrieve Points from Storage

Description

This function retreives points from the permanent storage using the time key and the string key.

Synopsis:

```
point retreivepoints(P_List &P, time_t t, string key);
```

Retrieve Camera Motion from Storage

Description

This function retreives image from the permanent storage using the time key.

Synopsis:

```
void retreivemotion(M_List &M, time_t t);
```

Erase Lines from Storage

Description

This function can be used to erase lines from the permanent storage.

Synopsis:

```
void eraselines(L_List &L, time_t t, string key);
```

Erase Points from Storage

Description

This function can be used to erase points from the permanent storage.

Synopsis:

```
void erasepoints(P_List &P, time_t t, string key);
```

Erase Images from Storage

Description

This function can be used to erase images from the permanent storage.

Synopsis:

```
void eraseimages(I_List &I, time_t t, string key);
```

Erase Camera Motion from Storage

Description

This function can be used to erase camera motion from the permanent storage.

Synopsis:

```
void erasemotion(M_List &M, time_t t );
```

Store Images into Storage

Description

This function can be used to store images into the permanent storage.

Synopsis:

```
void storeimage(I_List &I, ImageMat M, string key);
```

Store Points into Storage

Description

This function can be used to store points into the permanent storage.

Synopsis:

```
void storepoints(P_List &P, point *p, string key);
```

Store Lines into Storage

Description

This function can be used to store lines into the permanent storage.

Synopsis:

```
void storelines(L_List &L, lines l, string key);
```

Store Motion into Storage

Description

This function can be used to store motion into the permanent storage.

Synopsis:

```
void storemotion(M_List &M, Matx34d P);
```

Remove the keys

Description

Two overloaded functions are provided to remove the keys from the storage of keys.

Synopsis:

```
void remove(vector<time_t> &v, time_t & item);
void remove(vector<string> &v, string & item);
```

Display all the keys

Description

This function displays all the string keys.

Synopsis:

```
void Display_All_Keys_string(list<string> L);
```

Display all the keys

Description

This function displays all the time keys.

Synopsis:

```
void Display_All_Keys_time(list<time_t> L);
```

Compare Points

Description

This function can be used to compare two points to know if they are equal in the line.

Synopsis:

```
int comparepoints(lines *temp, lines *temp1);
```

Convert Points into lines

Description

This function can be used to convert detected points into lines.

Synopsis:

```
lines *convert(point *V);
```

Display Points on an Image

Description

This function can be used to display all the detected points on an image.

Synopsis:

```
Mat Display_Points_onImage(point *Vertexes, Mat im_bw1);
```

Assign labels to all the points

Description

This function can be used for assigning labels to all the points useful for tracking purposes.

Synopsis:

```
point *assignlabels(point *V);
```

Matches to Points

Description

This function can be used to convert from OpenCV matches into keypoints defined by the library.

Synopsis:

```
void matches2points(const vector<KeyPoint>& train,
                    const vector<KeyPoint>& query,
                    const std::vector<cv::DMatch>& mtches,
                    std::vector<cv::Point2f>& pts_train,
                    std::vector<Point2f>& pts_query);
```

Keypoints to Points

Description

This function converts the feature points in an image to points in 2d, which can be stored into the custom defined data structures provided by the library.

Synopsis:

```
void PointsToKeypoints(const vector<Point2f>& in,
                       vector<KeyPoint>& out);
```

Points to keyPoints

Description

This is an OpenCV required conversion for reconstruction. It converts the points in 2d into feature points in an image.

Synopsis:

```
void PointsToKeypoints(const vector<Point2f>& in,
                       vector<KeyPoint>& out);
```

Points to keyPoints

Description

This function can be used to align all matched points in both the images for triangulation using the matches.

Synopsis:

```
void GetAlignedPointsFromMatch(const vector<cv::KeyPoint>& imgpts1,
                               const vector<cv::KeyPoint>& imgpts2,
                               const vector<cv::DMatch>& matches,
                               vector<cv::KeyPoint>& pt_set1,
                               vector<cv::KeyPoint>& pt_set2);
```

Scalar Vector Decomposition

Description

This function is used to take scalar vector decomposition of essential matrix.

Synopsis:

```
void TakeSVDOfE(Mat_<double>& E, Mat& svd_u, Mat& svd_vt, Mat& svd_w);
```

Decompose Essential Matrix

Description

This function is used to decompose E into rotational and the translation matrix.

Synopsis:

```
bool DecomposeEtoRandT(Mat_<double>& E, Mat_<double>& R1,
                       Mat_<double>& R2, Mat_<double>& t1,
                       Mat_<double>& t2);
```

Check Coherent Rotation

Description

This function can be used to check the credibility of rotation matrix

Synopsis:

```
bool CheckCoherentRotation(cv::Mat_<double>& R);
```

Get Fundamental Mat

Description

This function calculates the fundamental matrix. Refer [15, 18].

Synopsis:

```
Mat GetFundamentalMat(const vector<KeyPoint>& imgpts1,
                      const vector<KeyPoint>& imgpts2,
                      vector<KeyPoint>& imgpts1_good,
                      vector<KeyPoint>& imgpts2_good,
                      vector<DMatch>& matches);
```

Test Triangulation

Description

This function tests the Essential matrix to see which of the four solutions found is the correct one, For this one calculates the third dimension and records the one for which z is positive to determine the best and valid solution.

Synopsis:

```
bool TestTriangulation(const vector<CloudPoint>& pcloud,
                       const Matx34d& P,
                       vector<uchar>& status);
```

Recover Point

Description

This function can be used to get library defined point structure from PCL specialized point cloud

Synopsis:

```
point *Recoverpoint(vector<CloudPoint>& pointcloud, point *P);
```

Convert into Point Cloud

Description

This function can be used to convert a point structure point into a point cloud, which can be visualized.

Synopsis:

```
pcl::PointCloud<pcl::PointXYZ>::Ptr convertintopointcloud(point *p,
              pcl::PointCloud<pcl::PointXYZ>::Ptr cloud,
              int size);
```

Iterative Triangulation

Description

This function can be used to do an iterative triangulation to find the third dimension of a point. At least 10 iterations are recommended. Refer [15, 18].

Synopsis:

```
Mat_<double> IterativeLinearLSTriangulation(Point3d u,
                                            Matx34d P,
                                            Point3d u1,
                                            Matx34d P1);
```

To display the 3D point cloud

Description

This function can be used to obtain visualization of point cloud.

Synopsis:

```
void visualization(pcl::PointCloud<pcl::PointXYZ>::Ptr source_cloud);
```

Viewer Generator

Description

This function can be used to generate a viewer for the Point Cloud

Synopsis:

```
boost::shared_ptr<pcl::visualization::PCLVisualizer> simpleVis(pcl::
PointCloud<pcl::PointXYZ>::Ptr cloud);
```

To create a mesh from the 3D point cloud

Description

This function can be used to create a mesh from the point cloud, basically triangulates the points into mesh.

Synopsis:

```
pcl::PolygonMesh pointcloudmesh(pcl::PointCloud<pcl::PointXYZ>::Ptr
cloud);
```

5.4. OTHER FUNCTIONS

This section includes functions, which are necessary for the purpose of the completion of the project and initialization of the various hardware in the system. This includes the camera calibration function that is not required to be done every time one starts running the code but will be necessary if one changes the camera which will lead to change in the internal parameters of the camera. The functions follows the algorithm given by Tsai in [16]. OpenCV also provides us with various functions, which are helpful in doing the camera calibration, The camera calibration is done using a chess board pattern. The function returns the camera calibration matrix and the distortion parameters.

Synopsis:

```
Mat cameracalibration(Mat &distortion);
```

Another function, is for reading images from the camera through the frame grabber board. This function consists of a modified source code of the sample application given by Sensoray. The function is called from image acquisition and involves activating the frame grabber and reading the image writing it on the hard drive and returning the file name and loading it onto the image matrix.

Synopsis:

```
void ImagetoSSD(string filename);
```

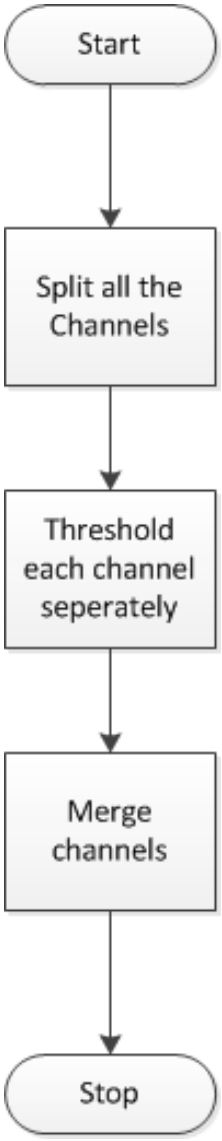

Figure 5.1. Threshold Function

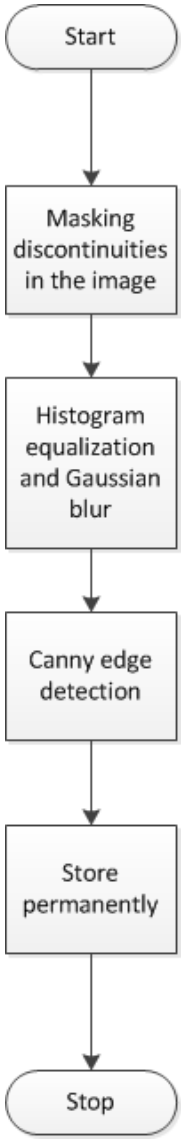

Figure 5.2. Preliminary Edge Detection

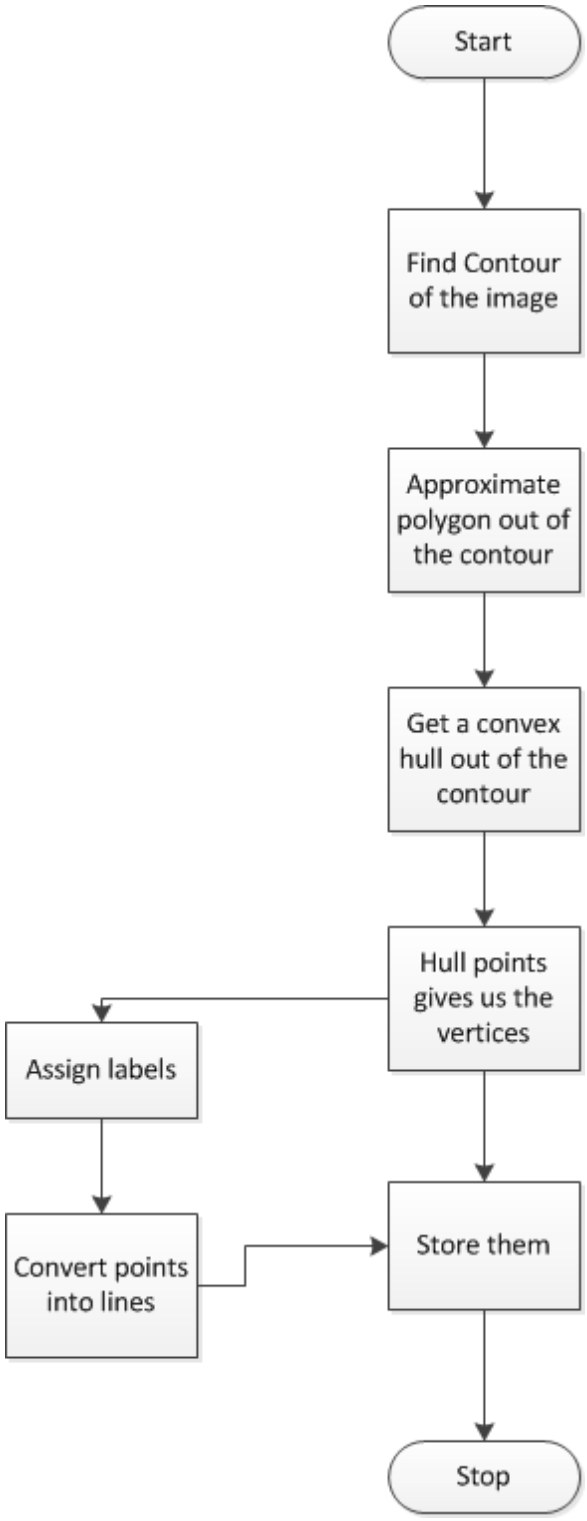

Figure 5.3. Vertices Detection

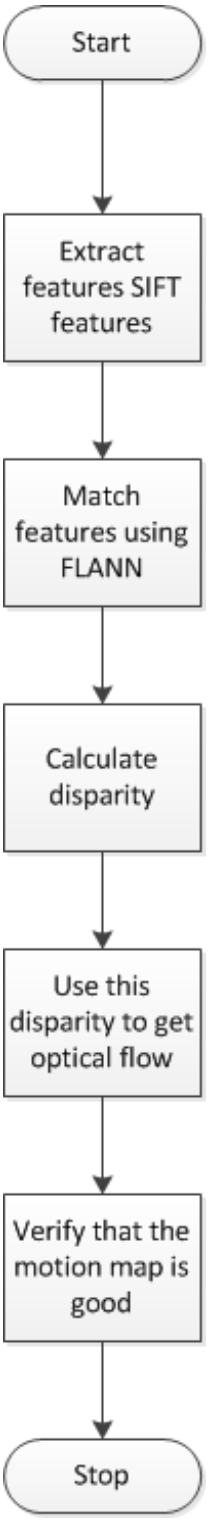

Figure 5.4. Motion Map

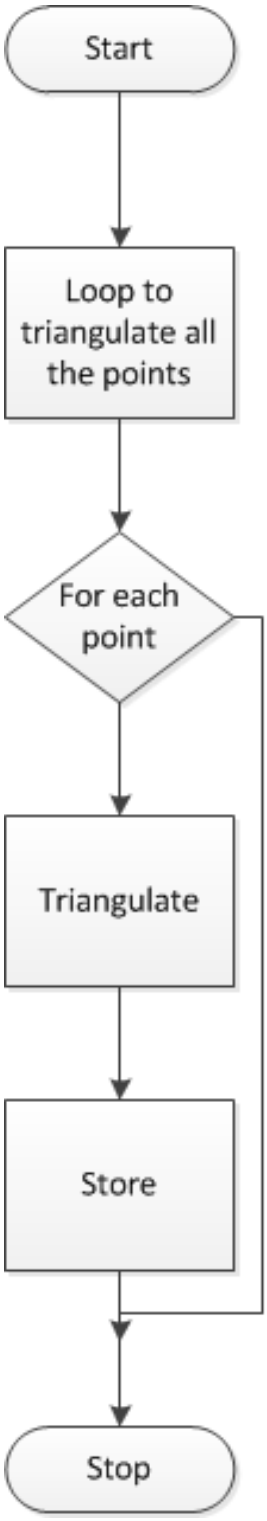

Figure 5.5. Triangulation

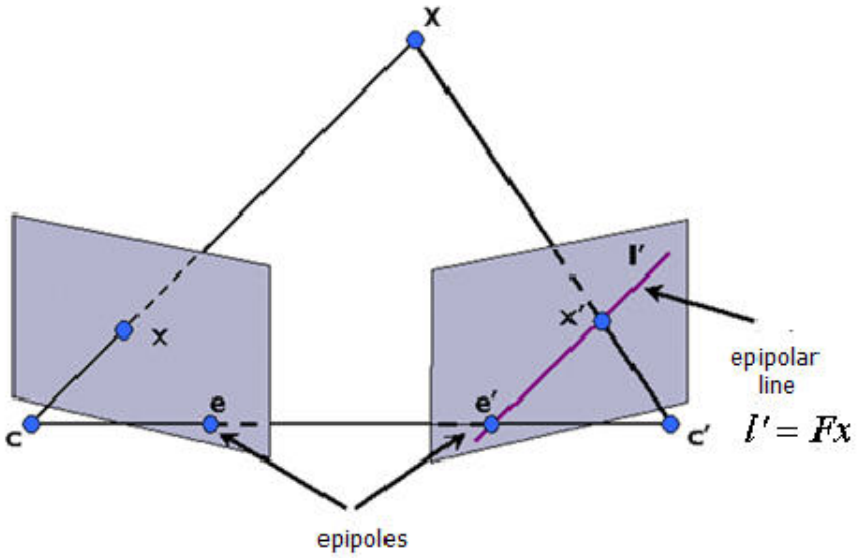

Figure 5.6. Epipolar Geometry

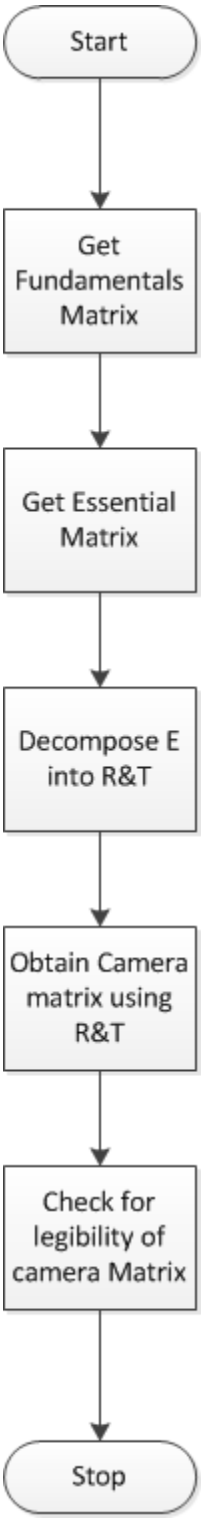

Figure 5.7. Finding Camera Matrices

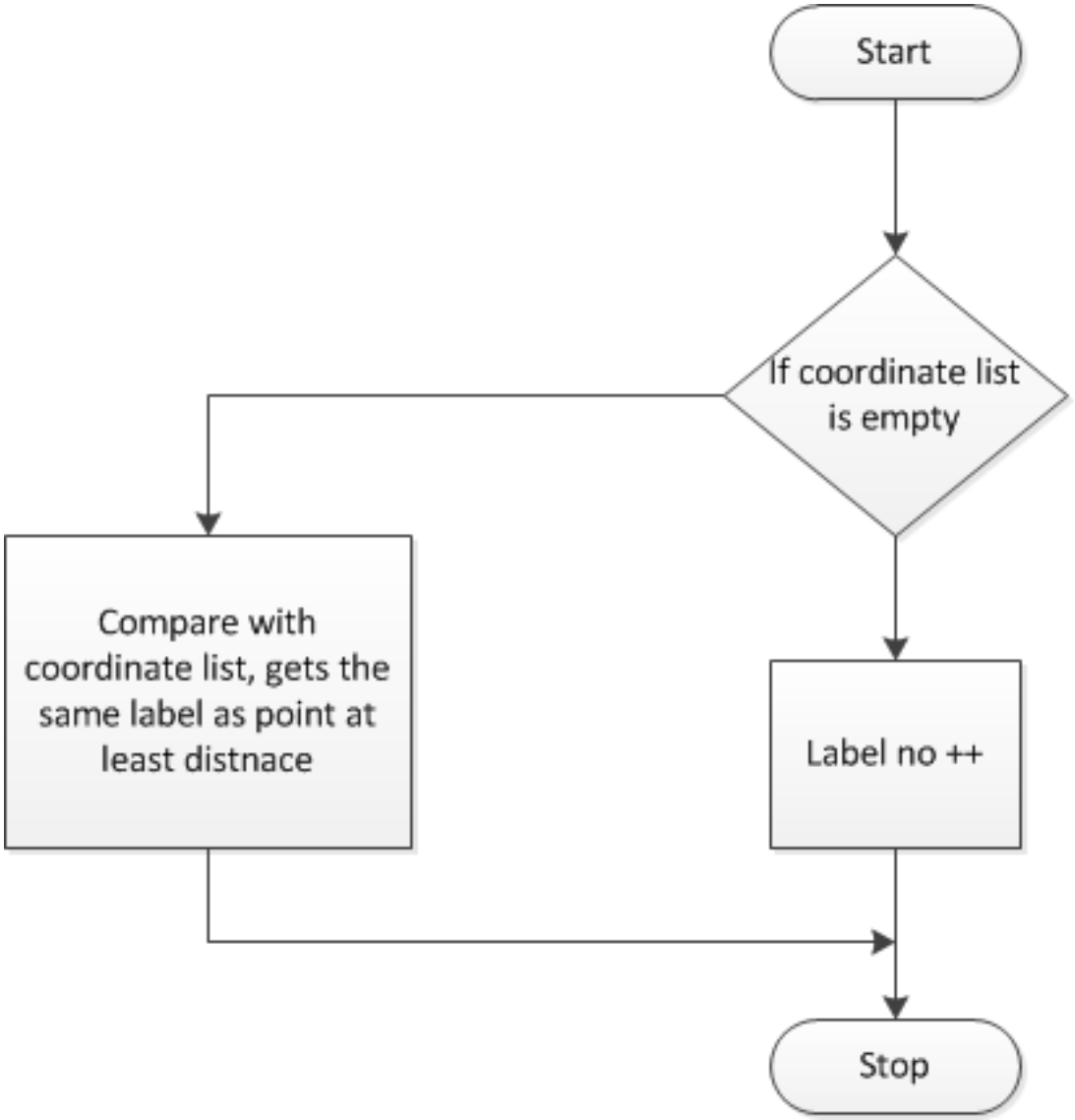

Figure 5.8. Assigning Labels

6. DISPARITY FROM TRUCK DATA

In the previous section, the third dimension of a point is detected using corresponding features detected from multiple images using a technique known as thr Structure From Motion. In this section the third dimension is detected using sensor data available from the truck. This data includes total number of rotations from the optical shaft encoder and the steering Angle from the potentiometer.

6.1. COORDINATE INFORMATION FROM SENSOR VALUES

The optical shaft encoder in the truck operates by translating the rotation of the shaft into light interruptions in the form of electrical pulses. The device gives output value as the total rotations performed by the wheel.

If the rotation is assumed as ω, then the distance travelled is given by

$$d = \omega \times l = 2 \times \pi \times r$$

where l is the circumference of the truck wheel and r is the radius of the truck wheel. This result is the distance travelled by the truck.

The potentiometer in the truck returns value that is propotional to the steering angle of the truck and if x and y are the plane coordinates of the current position of the truck then,

$$y = d \times cos\theta, x = d \times sin\theta,$$

where θ is the steering angle. At this point the z position of the truck is assumed to be zero. In future, if a gyro sensor is added to the system then the third coordinate of the point can be directly detected from the sensor.

The coordinates are stored as $\delta x, \delta y, \delta z$ where δ represents the change in the value. The incremental changes are added together to obtain the total change in the x and y from the initial position.

6.2. FUNDAMENTAL MATRIX FROM SENSOR VALUES

The points detected in the previous section are in 3D space. To convert them into the image plane. $x' = K \times x$, where x is the current point in the 3D space and x' is the point in the image plane . The transformation between the x and x' in the world plane is given by

$$T = \begin{bmatrix} cos\theta & sin\theta & tx \\ -sin\theta & cos\theta & ty \\ 0 & 0 & 1 \end{bmatrix}$$

Here tx and ty are the $\delta x, \delta y$ and θ is the change in the angle from the previous position. This transformation matrix in the image plane is given as $P' = KP$ in [15].

For example if the first point is taken at the origin then it can be represented as

$$P = K[I|0]$$

in the image plane. Where K is the camera matrix, the second point can be represented as

$$P' = K[R|t]$$

and if the Psuedo inverse P^+ is given as

$$P^+ = \begin{bmatrix} K^{-1} \\ 0 \end{bmatrix}$$

and the camera center C is given as

$$C = \begin{bmatrix} 0 \\ \\ 1 \end{bmatrix}$$

then the Fundamental Matrix is given by

$$F = [P'C]P'P^{+}$$

From F the essential matrix can be computed and from the triangulation of the points in the plane, an estimate of the z coordinate can be determined as shown in [15].

Calculate xyz from Sensor Values

Description

This function uses the truck sensor values to calculate the x and y coordinate.

Synopsis:

```
void calcualtexyz(float Encoder_data,
                  float theta,
                  coordinates &xyz_Data );
```

Calculate Fundamental Matrix from xyz

Description

This function uses x, y coordinates and the camera matrices to calculate the fundamental matrix .

Synopsis:

```
void CameraMatrixxyz(coordinates &xyz_Data, const Mat& K,
                     const Mat& Kinv, const Mat& distcoeff,
                     Matx34d& P, Matx34d& P1, vector<CloudPoint> &
                     outCloud);
```

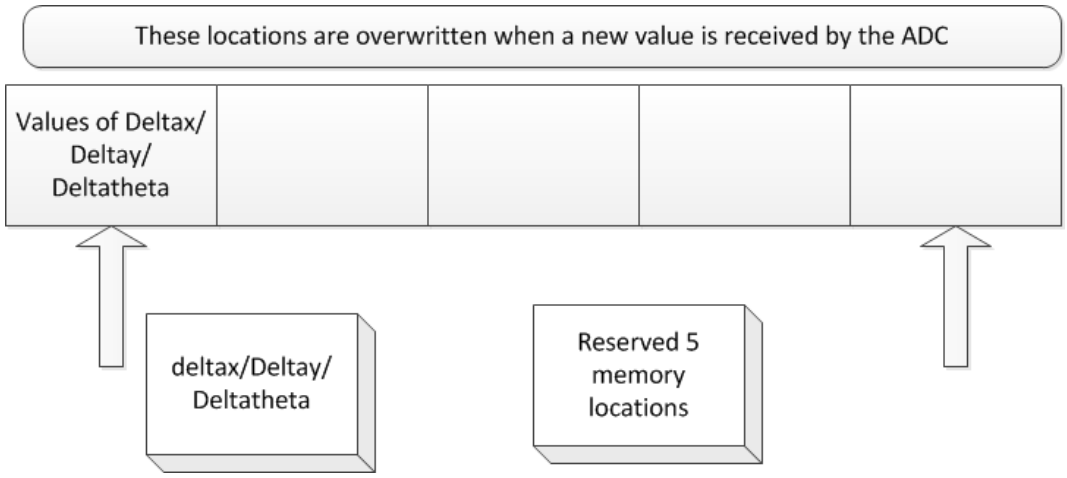

Figure 6.1. Storage δx

After calculating the camera matrices, functions from the previous section can be used to calculate the Z coordinate. This algorithm is known as triangulations.

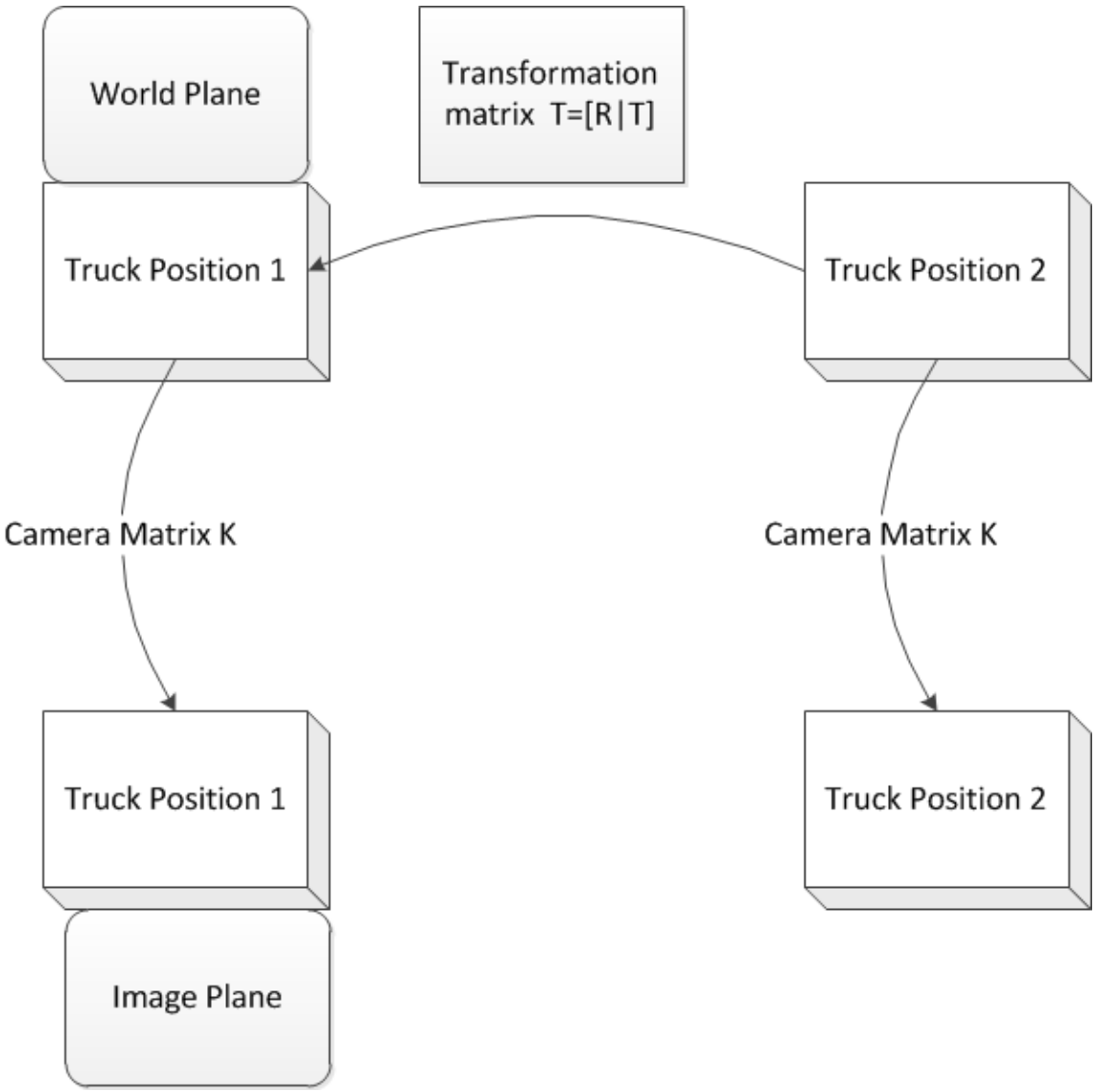

Figure 6.2. Transformation

7. RESULTS AND OBSERVATION

The goal of this work was to give the user a set of functions, that could be used to calculate the third dimension of a point in the image plane and provide visualization of it. The role of point cloud libraries and the OpenCV libraries was instrumental in achieving the task. The library is written in C++ and object oriented programming was used wherever possible. For the purpose of demonstrating the use of these libraries, a small program was written and executed. The code snippet below provides an example on the use of different components in the library. The algorithm is explained in Figure 7.1 The program is executed on the host hardware in real time. The program is divided into three parts, first part is the input of images, the second part is the use of the storage functions in the library and part three is to use different functions available for image processing purposes. The contents of the program are given below.

```
#include "function definitions.h" // This header fie needs to be
included to use the library. The user does not need to include the
general C++ header files as they are already included inside this
header file.

int main ()
{
// Declaring an object of Lines
    L_List L_object ;
```

```
// Declaring an object of Images
    I_List I_object;
// Declaring an object of Point
    P_List P_object;
// Declaring an object of Motion
    M_List M_object;
//Declare objects of two image matrices
    ImageMat *Iinp = new ImageMat;
    ImageMat *Iinp1 = new ImageMat;
/* Declare a variable size, which defines what size the input images,
should be resized to. Resizing is important because of the
computational efficiency which needs to be achieved.*/

    Size size(640, 480);

// Here the filenames for both the images are defined, These filenames
 could also be files whicha lready stored on the disk.
    /*
    string s = "1123.jpg";
    string s1 = "1124.jpg";
    */
```

```
// The choice of image acquisition is 2 here but could also be one,
where in one loads an image already on the disk.
    Iinp = ImageAcquisition(2, s, s, I_object);
    Iinp1 = ImageAcquisition(2, s1, s1, I_object);
//Resizing the curent image to a size specified by the user
    resize(Iinp->img,Iinp->img, size);//resize image
    resize(Iinp1->img,Iinp1->img, size);
// Code snippet demonstrates how to use the associative array for
storage and how to remove and display keys
    /*
    for (int i = 0 ; i < 10 ; i ++){
        storeimage(I_object, *Iinp, "Hello");
        cout << "The size of the key is: "<<I_object.keyouterImages.
        size()<<endl;
        cout << "The size of the key is: "<<I_object.keytimeImages.size
        ()<<endl;
    }
```

```
// Store an image using a key "hi"
   storeimage(I_object, *Iinp, "hi");
   cout << "The size of the key is: "<<I_object.keyouterImages.size()
   <<endl;
   cout << "The size of the key is: "<<I_object.keytimeImages.size()
   <<endl;

// This is how to remove an image
   cout << "Now removing "<< endl;

   time_t t = time(0);
   eraseimages(I_object, t, "Hello");
   cout << "The size of the key is: "<<I_object.keyouterImages.size()
   <<endl;
   cout << "The size of the key is: "<<I_object.keytimeImages.size()
   <<endl;
*/

// This set of code calculates and outputs the wireframe for each of
the two images.
/*
   Iinp= PrelimEdgeDetection(Iinp, "EdgeImage1", I_object);
```

```
   Iinp = VerticesDetection(Iinp, "VERTICESMAGE1", P_object, L_object,
    I_object);

   Iinp1= PrelimEdgeDetection(Iinp1 , "EdgeImage2",I_object);

   Iinp1 = VerticesDetection(Iinp1, "VERTICESMAGE2", P_object,
   L_object, I_object);
*/

// Defining objects for the calculation of features adn motion map
cv::Mat K , Kinv;
cv::Mat cam_matrix, distortion_coeff;
std::vector<CloudPoint> pointcloud;
std::vector<cv::KeyPoint> correspImg1Pt;

// Read the camera internal parameters from the file
cv::FileStorage fs;
fs.open("out_camera_data.xml", cv::FileStorage::READ);
fs["Camera_Matrix"]>>cam_matrix;
fs["Distortion_Coefficients"]>>distortion_coeff;
K = cam_matrix;
invert(K, Kinv); //get inverse of camera matrix
cout<< "cam matrixes"<< K<< endl;
```

```
// Declaring variables for the calculation of camera matrices
vector<KeyPoint> pt1, pt2;
vector<KeyPoint> imgpts1;
vector<KeyPoint> imgpts2;
vector<KeyPoint> imgpts1_good;
vector<KeyPoint> imgpts2_good;
vector<DMatch> match;


    Matx34d P = cv::Matx34d(1,0,0,0,
                            0,1,0,0,
                            0,0,1,0);
    Matx34d P1 = cv::Matx34d(1,0,0,50,
                             0,1,0,0,
                             0,0,1,0);


// calculates the motion map which basically calculates the optical
flow in the image.
int sp = CalculateMotionMap(Iinp->img, Iinp1->img, imgpts1, imgpts2,
pt1, pt2, imgpts1_good, imgpts2_good, match);


// Finds the fundamental and the essential matrix . Here the values P
and P1 are the camera matrices . This value may be stored by the user
to track motion.
FindCameraMatrices( K, Kinv, distortion_coeff, imgpts1, imgpts2,
imgpts1_good, imgpts2_good, P, P1, match, pointcloud);
```

```
// Convert the triangulated points into point cloud which is
compatible with the point Cloud library . this library provides the
visualisation necessary .
for (unsigned int i = 0 ; i < pointcloud.size(); i++){
    cout<<pointcloud[i].pt.x<<" , "<<pointcloud[i].pt.y<<" , "<<
    pointcloud[i].pt.z<<endl;
}
int size1 = pointcloud.size();
cout<<size1<<endl;
point *temppoint = new point[size1];
temppoint = Recoverpoint( pointcloud, temppoint);
for (unsigned int i = 0 ; i < pointcloud.size(); i++){
    cout<<temppoint[i].P.x<<" , "<<temppoint[i].P.y<<" , "<<temppoint[
    i].P.z<<endl;
}

// Declare a PCL recognised point cloud
pcl::PointCloud<pcl::PointXYZ>::Ptr point_display(new pcl::PointCloud<
pcl::PointXYZ> ());

point_display = convertintopointcloud(temppoint, point_display,
pointcloud.size());

// Display the visualization
visualization(point_display);
```

```
// Calcualte the polygonal mesh to demonstate the surface normals
pcl::PolygonMesh triangles;

triangles = pointcloudmesh(point_display);

// // namedWindow("Contours", CV_WINDOW_AUTOSIZE );

// // imshow("Contours", Iinp1->img);

// // waitKey(0);



// Delete all the memory references.
delete temppoint;
delete Iinp;
delete Iinp1;
K.release();
Kinv.release();
cam_matrix.release();
distortion_coeff.release();
return 0;

}
```

Key Points about the Library

- The header file declaration.h must be included in the code to use the library. There are some prequisite requirements for running the library. The library can be executed outside of the host computer with a webcam camera or using a stored set of images, The only issue, that the user would have to careful with, would be the installation of OpenCV and PCL library.

- The code snippet written above has the ability to input images from the camera and to triangulate its points for 3D. There are a few limitations of the library, that were observed during the testing and experimentation. The calculation of an accurate Fundamental matrix and essential matrix is only feasible when there are more than 100 point features detected in the image. It was observed that if few features are used then the Essential matrix ends up with a zero determinant which is bad for triangulation process.

- It was also observed that using the wireframe is extremely difficult to get more than 100 feature matches. Even though the function could work with a wireframe, it could not generate a dense point cloud with it. Hence it became necessary to use the gray scale images to calculate the camera matrices.

- Another limitation is the size of the input images. The hardware became very overloaded when a picture from a 12 megapixel camera is processed. It almost took half an hour to calculate all the camera matrices. To overcome this problem a resize function with variable size was introduced.

- Camera Calibration is very important in the system. Without the correct camera matrix it is not possible to find the required triangulation and the third coordinate.

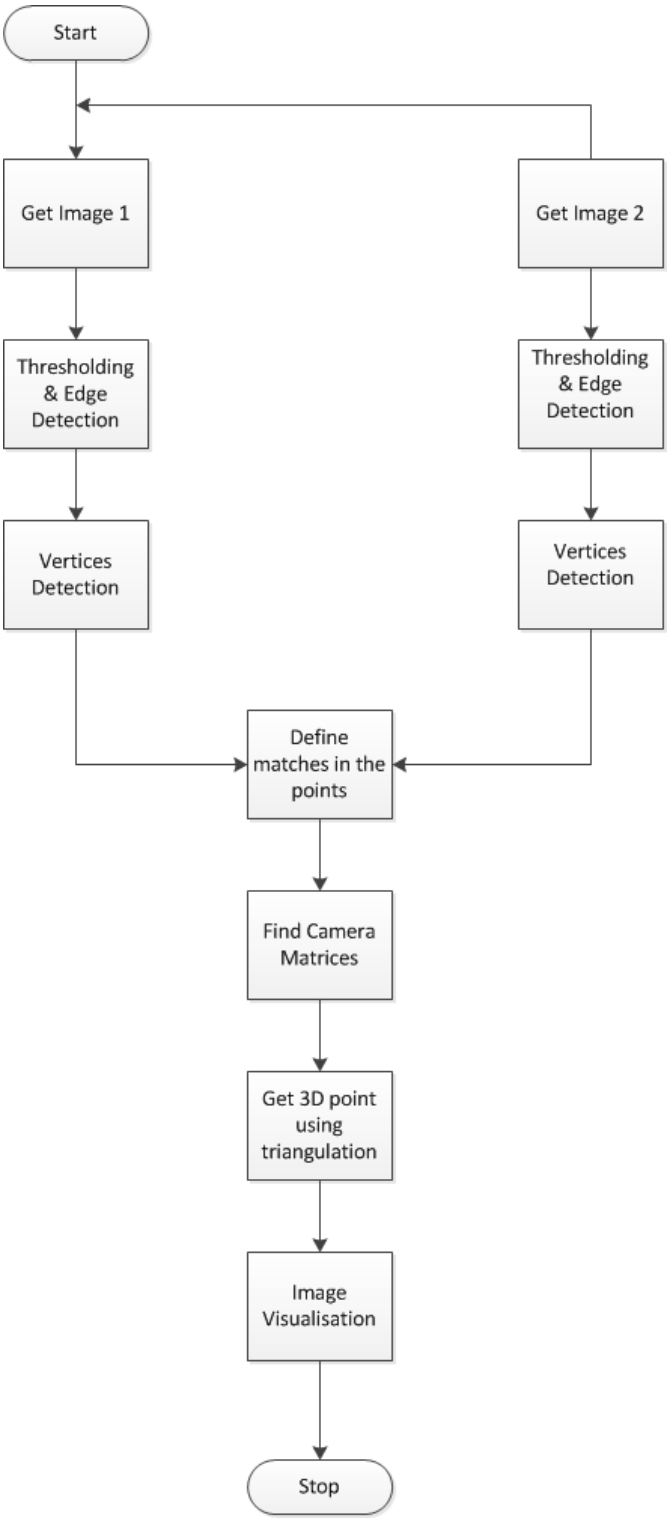

Figure 7.1. Algorithm of the System

BIBLIOGRAPHY

[1] Pravin Dhake. A real time operating system based test-bed for autonomous vehicle navigation. 2007.

[2] Robert S Woodley and Levent Acar. A testbed system for nonlinear or intelligent control. In *American Control Conference, 1999. Proceedings of the 1999*, volume 5, pages 3441–3445. IEEE, 1999.

[3] Aaeon. Aaeon pfm 945c user manual. 2005.

[4] Aaeon. Sensoray model 911 user manual. 2005.

[5] Connect Tech. Connect tech pci-104 to pc/104 adapter user manual. 2012.

[6] Paolo Mantegazza, EL Dozio, and S Papacharalambous. Rtai: Real time application interface. *Linux Journal*, 2000(72es):10, 2000.

[7] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. " O'Reilly Media, Inc.", 2008.

[8] Lijun Ding and Ardeshir Goshtasby. On the canny edge detector. *Pattern Recognition*, 34(3):721–725, 2001.

[9] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011.

[10] Pravin Dhake. A real time operating system based test-bed for autonomous vehicle navigation. 2007.

[11] Changchang Wu. Visualsfm: A visual structure from motion system. 2011.

[12] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustmentâĂŤa modern synthesis. In *Vision algorithms: theory and practice*, pages 298–372. Springer, 2000.

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

[14] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer vision and image understanding*, 110(3):346–359, 2008.

[15] Richard Hartley and Andrew Zisserman. *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[16] Berthold KP Horn. Tsai camera calibration method revisited. *Online: http://people. csail. mit. edu/bkph/articles/Tsai_Revisited. pdf*, 2000.

[17] Philip HS Torr and David W Murray. The development and comparison of robust methods for estimating the fundamental matrix. *International journal of computer vision*, 24(3):271–300, 1997.

[18] Richard I Hartley. A linear method for reconstruction from lines and points. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, pages 882–887. IEEE, 1995.

VITA

Krishnan Raghavan was born on the planet Earth in a small town named Kumbakonam, Tamilnadu, India in the year 1990. He received his primary education in Rajasthan, India. In July of 2008, He moved to Mumbai, Maharashtra for his bachelors degree. In 2012, hee received his Bachelors degree in Instrumentation Engineering from Vivekanand Education Society Institute of Technology that is affiliated with University of Mumbai. During these four years, he served as the Editor of International Society of Automation and volunteered for several other technical and non technical activities on campus.

After receiving his bachelors degree, he moved to U.S.A., where he joined the Missouri University of Science and Technology, Rolla as a graduate student in the Electrical and Computer Engineering Department to pursue a Master of Science degree with the Thesis option. He received his Masters Degree in Computer Engineering in the December 2014.