Integration of multiple vision systems and toolbox development

Rohit Vijay Bapat

INTEGRATION OF MULTIPLE VISION SYSTEMS AND TOOLBOX DEVELOPMENT

by

ROHIT VIJAY BAPAT

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

2012

Approved by

Ming C. Leu, Advisor
Fue-wen Frank Liou
Steven M. Corns

ABSTRACT

Depending on the required coverage, multiple cameras with different fields of view, positions and orientations can be employed to form a motion tracking system. Correctly and efficiently designing and setting up a multi-camera vision system presents a technical challenge. This thesis describes the development and application of a toolbox that can help the user to design a multi-camera vision system. Using the parameters of cameras, including their positions and orientations, the toolbox can calculate the volume covered by the system and generate its visualization for a given tracking area. The cameras can be repositioned and reoriented using toolbox to generate the visualization of the volume covered. Finally, this thesis describes how to practically implement and achieve a proper multi-camera setup.

This thesis describes the integration of multiple cameras for vision system development based on Svoboda's and Horn's algorithms. Also, Dijkstra's algorithm is implemented to estimate the tracking error between the master vision system and any of the slave vision systems. The toolbox is evaluated by comparing the calculated and actual covered volumes of a multi-camera system. The toolbox also is evaluated for its error estimation. The multi-camera vision system design is implemented using the developed toolbox for a virtual fastening operation of an aircraft fuselage in a computer-automated virtual environment (CAVE).

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Ming C. Leu, for his advice, guidance, and encouragement throughout my graduate studies and particularly during this research. I would also like to express my deepest gratitude to Dr. Steven M. Corns for his invaluable contributions and suggestions during the entire duration of this project.

I would also like to thank Dr. Frank Liou for his time and effort in serving as a committee member and reviewing this thesis. Special thanks go to my colleagues Dr. Wenjuan Zhu, Sajeev Chirayil and Chinmay Daphlapurkar for their valuable advice and also my friends who have been of wonderful support.

Last, but not least, I am very thankful to my parents for their continuous love and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The camera-based motion tracking system is an important aspect of human interaction with virtual reality (VR) applications. Different VR applications require different sizes of tracking areas, depending upon which the user employs different numbers of cameras to form a multi-camera vision system. The user places the cameras at various positions and with various orientations to attain the desired coverage area. Designing a multi-camera vision system setup is a challenging task. The system should be mathematically and scientifically correct and also practically efficient. The development and implementation of a novel software toolbox that will help users design multi-camera vision systems is described in this thesis, which is intended for user to design multi-camera vision systems.

With this toolbox, the user will be able to design and mathematically evaluate the volume covered by the designed multi-camera vision system setup prior to the actual setup. The toolbox is designed to provide flexibility in the number of cameras used, with variable fields of view (FOVs) placed at random positions and orientations. The user can also change the positions and orientations of the cameras and re-evaluate the system's setup.

Another aspect of designing the multi-camera vision system setup is calibrating the cameras together. Several calibration methods have been suggested and implemented in past research. A camera calibration toolbox is available in MATLAB, as referred to in [10]. Implementing this toolbox involves integrating calibration methods suggested by Zhang [7] and Svoboda [5]. The first part of the calibration method is the calibration of a stereo vision system for intrinsic parameters of cameras based on the algorithm suggested by Zhang. The second part of the method is the calibration of cameras for extrinsic parameters using the algorithm suggested by Svoboda [5]. Based on intrinsic and extrinsic parameters, a transformation matrix is determined for the multi-camera vision system. One very important condition of this method of calibrating multiple cameras is that at least some common overlap between the FOVs of all cameras must be achieved. For larger tracking areas, more cameras are employed. It is not always feasible to achieve a common overlap between the FOVs of all cameras. The practical way to handle this issue is to group the cameras whose FOVs overlap. This results in the formation of more than one multi-camera vision system, each with a defined coordinate system for motion tracking. To define a homogeneous coordinate system for the entire tracking area, it is important to generate a mathematical relationship between several coordinate systems. This thesis explains the solution to this issue, i.e., integrating the vision systems using Horn’s algorithm.

This thesis contains a detailed description of the development of the toolbox, as well as the implementation of Horn’s algorithm for integrating multi-camera vision systems. Section 2 contains a brief summary of the background of the research and VR applications, explaining the history of VR applications, the computer-automated virtual environment (CAVE), the software and hardware used in the CAVE, etc. The principles of the multi-camera vision system, its calibration using Svoboda’s algorithm and its error estimation technique is described in Section 3. Section 4 discusses the development of the multi-camera vision system setup toolbox in detail. This discussion includes a description of the modules in the toolbox, the architecture of the toolbox, the principles used in the MATLAB module and the visualization pipeline of Visualization Toolkit (VTK). Section

5 includes a detailed explanation and implementation of Horn’s algorithm for integrating the multi-camera vision systems and the error estimation of the vision systems. It also includes a description of the shortest path finding methodology using Dijkstra’s algorithm. The results of the toolbox are evaluated by comparing the calculated dimensions of the total area covered with the actual volume covered by the multi-camera vision system. The toolbox’s error estimation also is evaluated. The VR applications based on the motion tracking system and the CAVE are briefly explained in Section 6, which also explains the application of Wiimotes as the cameras and the simulation of the virtual fastening operation of the fuselage.

1.1. REVIEW OF MULTI-CAMERA SETUP METHODS

Because multi-camera setup is a complex process, some background study is required before determining which methodology to use. Some recent research has discussed similar issues regarding achieving the correct multi-camera setup.

Cerfontaine et al. [1] described a multi-camera setup optimization method for optical tracking. The approach uses camera parameters, such as initial position and orientation and FOV, to find the correct camera placement. The method requires the dimensions of the given volume as input. Then, depending upon the camera parameters, the algorithm will determine the positions visible to any particular camera, and such positions will be collected together. Next, the positions that are visible to either of the two cameras are collected together. These points will define the volume within the given volume that can be used for tracking purposes.

Svoboda et al. [5], while describing multi-camera self-calibration for virtual environments, explained the configuration of the multi-camera setup. The cameras were placed so as to achieve a common overlap between their FOVs. However, they did does not describe the setup of the cameras, i.e., their positioning and orientation. The basic issue is to determine the common overlap and to achieve the maximum tracking area.

Zürl [24] explained the multi-camera setup for A.R.Ttrack®, which is a professional tracking system. The methodology behind placing the cameras is mainly depends on the occlusion. The markers are placed at specific positions inside the given tracking area; the cameras are placed such that the markers are visible to the cameras. The number of cameras and initial positions and orientations are set manually and depend on the experience of the user. After setting up the multi-camera system, if sufficient volume is not covered, the cameras are moved and reoriented manually. This method is user specific and time consuming.

Another professional tracking system, widely used for motion tracking in VR applications, is VICON® [27]. The method for arranging multiple cameras in VICON is to set them up in a semicircular manner such that the cameras point to the tracking area. In this method, the cameras are placed and oriented manually. The number of cameras, as well as their placements and orientations, depend on personal experience. The system does not come with a software tool that can calculate the volume covered by the system. In this method, a faulty multi-camera setup will take time to reposition and re-orient.

Considering that the cameras in a professional tracking system are placed and oriented manually, the necessity arises for a software tool that can evaluate the multi-camera setup by calculating the volume covered by the cameras and comparing it with the desired volume coverage. This thesis proposes and describes the development of a software toolbox that can serve this purpose. The toolbox also is designed to generate the visualization of given volume and covered volume so that the user can visualize the multi-camera system setup and evaluate it through the software toolkit.

1.2. ERROR ESTIMATION TECHNIQUES

Error estimation of a multi-camera motion tracking system is important. Much research has been and continues to be conducted regarding this issue. Horn et al. [25] described an algorithm for a closed-form solution of absolute orientation using unit quaternions. The paper explains how to determine the relationship between two Cartesian coordinate systems. The inputs for this method are the coordinates of common points in two Cartesian coordinate systems. This method is particularly useful for a multi-camera setup in which all cameras cannot form one vision system. So, several vision systems are formed from the cameras, and then they are integrated using Horn’s algorithm. As the calculations depend on the generated coordinates of the points, an error might be introduced while forming a closed-form solution. However, the paper does not explain how to handle this error.

Dijkstra developed an algorithm [23] to find the error between a master vision system and slave system. For a system in which several vision systems exist, the user defines one of the systems as a master system and the others as slave systems. The algorithm calculates the shortest path between any two vision systems depending on the input errors between the vision systems, which are manually calculated. The shortest path indicates the least error between any two vision systems. Hence, it is very useful for applications in which several multi-camera vision systems exist.

Bellman and Ford [26] also developed an algorithm for shortest path generation between any two vision systems. The algorithm is similar to Dijkstra’s algorithm, but the primary difference is that it only takes care of error between immediate vision systems. In general, the algorithm is employed when the errors between vision systems are negative. The algorithm takes more CPU time than Dijkstra’s algorithm.

Freeman et al. [17], in proved that calibration errors differ from system to system and also depend on the intrinsic and extrinsic parameters and scene geometry. The authors present an algorithm for predicting the statistics of marker tracking error in real-time. The implemented method for finding the error is to calculate the average and standard deviations.

Pentenrieder et al. [18] described several ways to predict the errors for different calibration methods and professional tracking systems. Their paper describes a way of finding the error by comparing the actual distance between two markers and the distance calculated from the coordinates of two markers. These errors are determined at several positions inside the tracking area. Finally, the error of multi-camera vision systems is calculated either by finding the average or root mean square of these errors.

Considering investigations into finding the error of multi-camera vision systems, this thesis describes the methodology in which several vision systems are integrated by implementing Horn’s algorithm. The error between two consecutive vision systems is measured manually using the method explained by Freeman et al. [17]. The system explained in this thesis consists of more than two multi-camera vision systems, in which one is the master vision system and the others are slave vision systems. Hence, there is a need to find the shortest path. Amongst the algorithms described above, Dijkstra’s algorithm is chosen because it is faster than the Bellman-Ford algorithm and because the errors are positive; hence, there is no need to use the Bellman-Ford algorithm. In this thesis, these algorithms are practically implemented and tested.

2. BACKGROUND

Author used the motion capture system based on multiple wiimote tracking systems for virtual reality applications. The placement and orientation of the Wiimotes was based on the area to be tracked. Author attempted to implement the previously-developed wiimote vision system setup toolkit, which assumes that all Wiimotes are placed at equal distances from each other and have the same orientation. Also, the toolkit provides results when the Wiimotes are placed at a height of 7m from the ground.

The practical implementation of a multi-camera tracking system requires the flexibility of choosing random positions and orientations for each camera. The placement and orientation of Wiimotes is based on the personal experience of the user.Author required a technical and scientific reasoning behind specific placements and orientations of the cameras; for this purpose, author is proposing to develop a software toolbox using MATLAB and VTK. The user will choose the type of camera, provide dimensions of the area to be tracked and specify the position and orientation of the camera in three-dimensional (3D) space. The calculations for the area covered by the tracking system will be carried out using MATLAB. This toolbox will be able to give a graphical 3D visualization of the tracking system camera setup and the area covered by the tracking system. A graphical interface will then be developed using VTK in order to provide a 3D visualization of the complete system to the user.

The toolbox will eliminate the need to have personal experience in order to position and orient the cameras of the tracking system. A module implementing Dijkstra’s algorithm will be developed in MATLAB as part of the toolbox. If the user wants to use multiple vision systems for position tracking, this module will provide the shortest path between the master vision system and each of the other vision systems depending on the distance measurement error between adjacent vision systems. Ultimately, the user will be able to implement the multi-camera tracking system for VR applications and then analyze the tracking results provided by the toolkit.

2.1. VIRTUAL REALITY

2.1.1. History of Virtual Reality. The term *virtual reality* sprouted in the 1960s. It was the time when new age cinematography and entertainment devices began to develop. The concept of virtual reality was stormy and sensational at that time. Due to new technologies in entertainment, people began imagining the concept of a human being present in a world that feels real but that actually is not. The idea of virtual reality has been around since 1965, when Ivan Sutherland expressed his ideas of creating virtual or imaginary worlds. The device by which total immersion was attempted for virtual reality was developed by Morton Heilig. Between 1960-1962, Heilig created a multi-sensory simulator called Sensorama. A prerecorded film in color and stereo was augmented by binaural sound, scent, wind and vibration experiences. This was the first approach to creating a virtual reality system, and though it had all the features of such an environment, it was not interactive [30]. Numerous definitions of virtual reality (VR) exist that depend on the context of its application. Virtual reality commonly is referred to as a computer-generated environment that offers the viewer a convincing illusion and an intense feeling of immersion in an artificial world that exists only in the computer. Virtual reality thus often is referred to as *immersion technology*.

Virtual reality (VR) and virtual environments (VE) are used interchangeably in the computer community. These terms are the most popular and most often used, but there are many others, including *synthetic experience*, *virtual worlds*, a*rtificial worlds* and *artificial reality*. Some definitions of VR include:

1) "Real-time interactive graphics with three-dimensional models, combined with a display technology that gives the user the immersion in the model world and direct manipulation." [30]
2) "The illusion of participation in a synthetic environment rather than external observation of such an environment. VR relies on a three-dimensional, stereoscopic head-tracker display, hand/body tracking and binaural sound. VR is an immersive, multi-sensory experience." [30]
3) "Computer simulations that use 3D graphics and devices such as the DataGlove to allow the user to interact with the simulation." [31]
4) "Virtual reality refers to immersive, interactive, multi-sensory, viewer-centered, three dimensional computer generated environments and the combination of technologies required to build these environments." [33]

Since the early 1960s, there have been many changes in the technologies used for immersion. The evolution of computers and electronics in the late 1980s boosted the experiments people conducted in an attempt to perfect virtual reality applications and technologies. The developments in electronics and sensors helped early researchers to develop means for interaction between humans and virtual environments [32].

2.1.2. Interaction with Virtual Environments. Virtual reality systems are evaluated primarily based on the extent to which the user can be immersed in and interact with it. VR requires more resources than do standard desktop systems. VR requires additional input and output hardware devices and special drivers for enhanced user interaction. However, extra hardware alone will not create an immersive VR system. The most important parts of the human-computer-human interaction loop fundamental to every immersive system are the input and output devices. The user is equipped with a head-mounted display, tracker and an optional manipulation device (e.g., three-dimensional mouse, data glove, etc.). As the human performs actions such as walking, rotating the head (i.e., changing the point of view), and describing data, his/her behavior is fed to the computer from the input devices. The computer processes the information in real-time and generates appropriate feedback that is passed back to the user by means of output displays. In later stages, many researches was conducted to develop low-cost interaction systems.

One of the most important developments has been low-cost motion tracking systems. The development of systems sometimes is inspired by gaming technologies (Nintendo Wiimotes, Xbox Kinect, etc.). Basically, the input devices carry the information about the state of the user to the system in real-time. The output devices make changes in the virtual environment in accordance with the inputs to the system. These changes in the virtual environment cause immersion. Hence, correct choices of input and output devices facilitate the development of good virtual reality applications [32]. VR technologies can be applied to study processes in which:

- the working environment is hazardous (chemical, nuclear reactors)

- the process setup is time consuming and complex (mechanical assemblies)
- the setup is expensive (fuselage assemblies, medical operations)
- delicate operations are performed (medical operations)

Some of the most popular input devices are CyberGlove™, DataSuit, Motion Tracker, touch sensors, haptic devices (Phantom Device™) and Wands. The most familiar output devices are head-mounted display (HMD), cathode ray tube (CRT) projectors, and liquid crystal display (LCD) projectors. The schematic representation is shown in Figure 2.1. Some auxiliary devices that facilitate the immersion of humans are stereo glasses, either shutter glasses or polarized glasses. 3D visualization is the most important component of VR applications because it facilitates the real life visualization of objects in the scene.

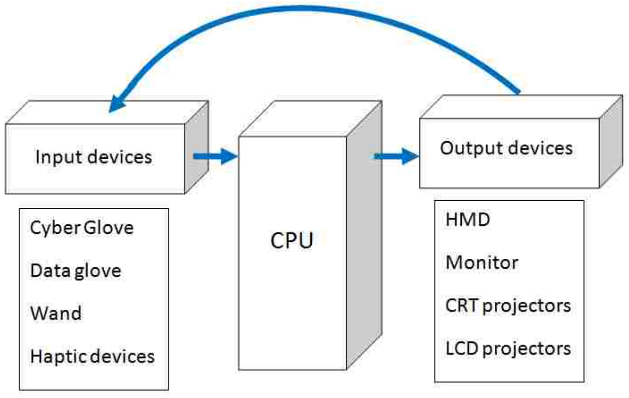


Figure 2.1. Interaction of Input and Output Devices with Computer System

2.2. AVAILABLE VIRTUAL REALITY SOFTWARE

Apart from the input and output devices, which are generally electronic and electrical equipment, one of the most important parts of virtual reality applications is the computer. Computers help in developing the desktop kind of virtual reality. These days, almost all virtual reality applications are developed on computers. As computers have continued to evolve since the 1990s in terms of processing speed, memory and graphics rendering capacities, more and more realistic applications have been developed. The basis of these computer applications is software, which facilitates application development.

Many types of software and software toolkits have been developed for creating virtual reality applications, some of which are commercially developed by commercial industries and some of which are *open-source*. Open-source software is most popular amongst researchers. Many early researchers developed their own open-source VR software. Some examples of VR software include:

Commercial software:

- 3DVIA Virtools *by* Dassault Systems
- Vizrd *by* WorldViz (www.worldviz.com)
- Quest3D (http://www.vrealities.com/quest3d.html)
- Syzygy *by* Illinois Simulation Lab
- EON Technologies
- CAVElib (http://www.mechdyne.com/cavelib.aspx)

Open-source software:

- VRJuggler *by* VRAC , Iowa State University

- OpenGL
- OpenSceneGraph (www.openscenegraph.org)
- VRML – Virtual reality modeling language
- VRPN – Virtual reality peripheral networks

This software is developed basically by coding, which is done in various languages. Among them, C++, C#, JAVA and Python are the most important computer development languages. For commercial software, the coding is done in specialized scripts developed for particular software programs. They have a specific structure of routines, classes and processes. The user must use these routines to create real-time interactive virtual environments. Sometimes, the routines of two different toolkits are combined in a code to exploit their peculiar advantages. One example is the combination of VRJuggler and OpenGL, which will be explained in detail in the applications section.

2.3. COMPUTER-AUTOMATED VIRTUAL ENVIRONMENT (CAVE)

A CAVE is a virtual reality and scientific visualization system. Instead of using an HMD, it projects stereoscopic images on the walls of a room (user must wear LCD shutter glasses). A standard CAVE consists of four surfaces, which include three back-projected walls and a front-projected floor. The projections on the walls of the CAVE are monitored by computers. The virtual environment is divided so as to project on separate walls. Each computer is connected to either a CRT or LCD projector. The scenes rendered on the walls, which depict the same virtual scene, are stereo images created at a high frame rate. The synchronization between the processing computers is very important in order to attain a realistic synchronization of projected images. Ideally, the computers form a ‘cluster’ in which one computer is the *master* system, and the others are *slave* systems. The main rendering pipeline in followed by the master system, and the interacting devices are also connected to this system. The most popular input devices used in a CAVE are wands, cyber gloves and marker-based motion tracking systems. A CAVE is basically used for the visualization of virtual environments, which consist primarily of CFD visualizations, automobile components, and ergonomic analyses [33][34].

CAVE CONFIGURATION: The CAVE available at Missouri University of Science and Technology (Missouri S&T) has the following configuration:

Dimensions: 3m X 3m X 3m (Height X Width X Depth)

Number of walls: 4 (Front, Left, Right and Floor)

Number of computers in cluster: 4

Computer system: Windows 7 ™ (64 bit)

Display projectors: CRT projectors

Number of projectors: 4

Display resolution: 1600x900

Stereo display type: Active stereo display

Motion tracking system: Wiimote and Firefly camera tracking system

Frequency of image flipping: 85 Hz

Figure 2.2 depicts a schematic representation of the CAVE at Missouri S&T. Four CAVE computers form a computer cluster, within which the master computer governs the display on the front wall. All the computers are connected to individual CRT displays. The projectors also are interconnected with each other in order to attain synchronization.

The images displayed on the walls are in stereo mode. The type of stereo is an active stereo, in which the images for the left and right eye are flipped one after another at a certain frequency. The shutter glasses are used, in which shutters for the left and right eye are opened and closed in sync with the frequency of the stereo, thus creating a 3D effect.

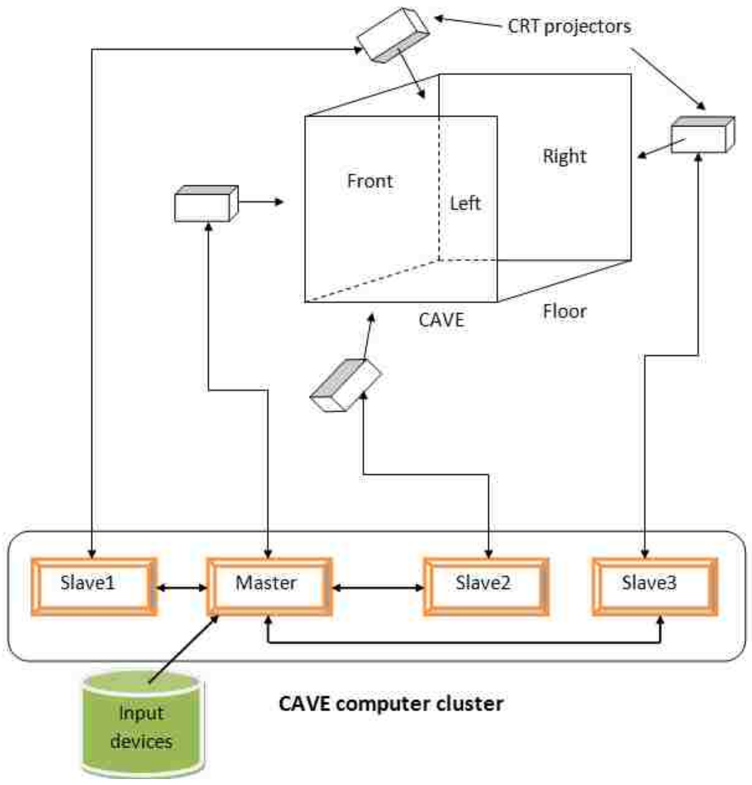


Figure 2.2. Schematic Diagram of CAVE Setup

2.3.1. CAVE Hardware. Any virtual reality application consists of hardware and software devices working together. For the CAVE at Missouri S&T, various types of hardware are installed together. By classic definition, this hardware consists of the structure of the CAVE, walls, computer systems, projectors and shutter glasses. The most important hardware devices in a CAVE are the display projectors, which, in CAVEs, always are CRT displays. CRT displays are based on conventional television technology and offer relatively good image quality with a high resolution (up to 1600x1280), sharp view and big contrast. Their disadvantages are their excessive weight and power consumption. They also generate strong, high-frequency, magnetic fields that may be hazardous to the user's eyes.

A 3D display functions on the basis of stereo vision. The emitters are also in sync with the projectors shown in Figure 2.3. As shown in Figure 2.4, system contains infrared emitter for corresponding frequency of image flipping, receptor on the shutter glasses. Frequency of image flipping (85Hz) synchronized with the shutter glasses. The opening and closing of the shutters of crystal shutter glasses shown in Figure 2.5 is synchronized with the frequency of the stereo display. The infrared emitter sends the synchronizing signal to the receptor of shutter glasses.

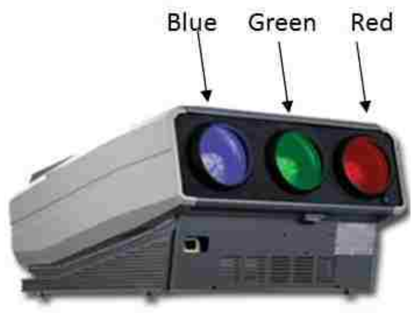


Figure 2.3. CRT Projector for Stereo Image Projector

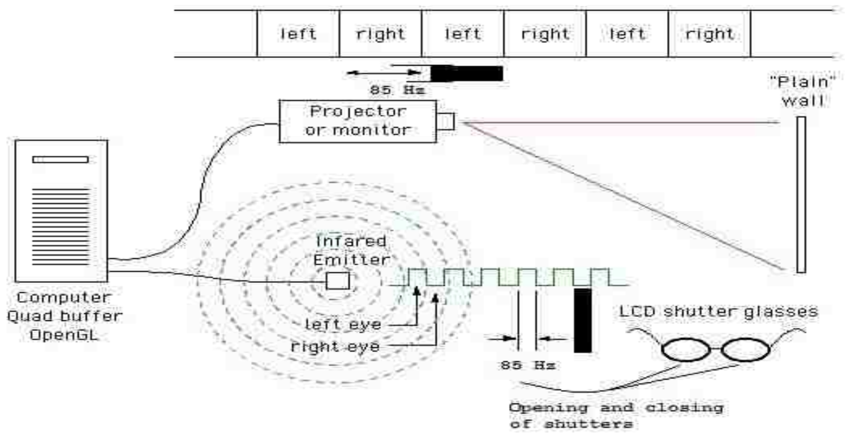


Figure 2.4. Schematic Diagram of Shutter Glasses

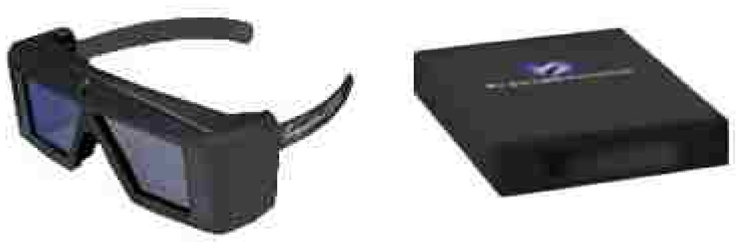


Figure 2.5. Shutter Glasses and Infrared Emitter

2.3.2. CAVE Software. Computers serve as the backbone of all kinds of new age VR applications. Whenever computers are considered, the first aspect that comes to mind is software. Apart from the hardware setup for the display and synchronization, everything is software based. As discussed previously, there exist various software programs for VR applications. To design the application in the CAVE,experimented with different software, such as OpenSceneGraph, Visualization Toolkit (VTK), OpenGL, VRJuggler, finally settling on VRJuggler + OpenGL. In the following sections, these two software programs are explained, along with their advantages.

2.3.2.1 VRJuggler. VRJuggler is an object-oriented software system for the development and execution of virtual reality applications. To achieve hardware independence, the VRJuggler architecture is based on a microkernel, which uses a set of managers, each one dedicated to specific tasks. The microkernel's main responsibility is to serve as the mediator for the managers, which involves the following tasks:

- Sustain the interactive performance of the system and the applications
- Coordinate interactions among the managers and manage the communication between the managers and the applications
- Maintain synchronization of the system components
- Handle runtime reconfiguration of the VR system
- Direct the execution of multiple applications

The programming for VR requires more than just knowledge of a given programming language. VRJuggler takes advantage of many programming design patterns and advanced concepts to maximize its power, flexibility, and extensibility. A good background in mathematics is helpful for performing the myriad transformations that must be applied to 3D geometry. The kernel-based structure helps in designing the middleware application for VR applications. VRJuggler is situated between the hardware and the renderer.

The kernels in VRJuggler form routines to interact with the computer system on the hardware level. VRJuggler uses the rendering pipelines of OpenGL and also has routines to use the rendering of OpenSceneGraph, VTK, etc. VRJuggler applications do not have a main() function, but further explanation is required. While it is true that user *applications* do not have a main() function because they are objects, there must still be a main() somewhere that starts the system because the operating system uses main() as the starting point for all applications. In typical VRJuggler applications, there is a main(), but it only starts the VRJuggler kernel and gives the kernel the application to run. It then waits for the kernel to shut down before exiting. The basic structure of the kernel model for the main() function is as follows [34][35]:

```
#include <vrj/Kernel/Kernel.h>
#include <simpleApp.h>
   int main(int argc, char* argv[])
  {
      vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
      simpleApp* app     = new simpleApp();        // Create the app object
      kernel->loadConfigFile(...);           // Configure the kernel
      kernel->start();                       // Start the kernel thread
      kernel->setApplication(app);           // Give application to kernel
      kernel->waitForKernelStop();           // Block until kernel stops
      return 0;
 }
```

Another important aspect of VRJuggler after setting up the kernel for application is sending the display properties to the computer. The information about the display is stored in a *configuration file* (*.jconf). This file is written in an XML coding routine. The configuration file for a particular system is unique depending upon the frequency of image flipping of the stereo, size of projection, and input devices. In the configuration file,   define the origin of the virtual scene. To implement VRJuggler for the CAVE application,created four configuration files for four computers. Hence, a different configuration file exists for each wall of the CAVE.

2.3.2.2 OpenGL. OpenGL is a software interface to graphics hardware. It is designed as a hardware-independent interface to be used for many different hardware platforms. OpenGL programs can also work across a network (client-server paradigm), even if the client and server are different kinds of computers. OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, the user must work through whatever windowing system controls the particular hardware being used. Similarly, OpenGL does not provide high-level commands for describing models of 3D objects. Such commands might allow the user to specify relatively complicated shapes, such as automobiles, body parts, airplanes, or molecules. With OpenGL, the user must build the desired model from a small set of *geometric primitives* - points, lines, and polygons. OpenGL is a state machine, i.e., various states (or modes) remain in effect until the user changes them. OpenGL Pipeline contains a series of processing stages in order. Two sets of graphical information, vertex-based data and pixel-based data, are processed through the pipeline, combined together, and then written into the frame buffer. Figure 2.6 shows the rendering pipeline of OpenGL [36].

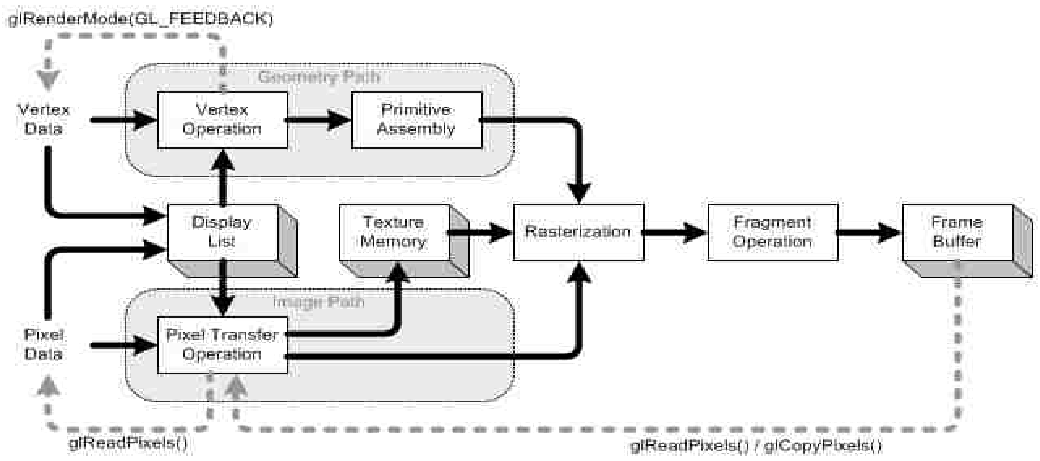


Figure 2.6. Architecture of Rendering Pipeline of OpenGL

Another important toolkit used along with OpenGL is the OpenGL Utility Toolkit (GLUT). This utility toolkit is developed to render basic geometric objects, such as curves, NURBS, and freeform surfaces. GLUT is very important to the texturing of the object. Texturing is analogous to wrapping a picture or an image on a surface. Another important use of OpenGL and GLUT is to transform the objects. OpenGL and GLUT use real-time transformations such as rotation, translation and scaling. The main() function of OpenGL is in an inherent loop. The tracking system is connected to the OpenGL part of the code for real-time simulations.

2.4. THE VISUALIZATION TOOLKIT

The Visualization Toolkit (VTK) is an open-source, portable (Windows, WinTel/Unix), object-oriented software system for 3D computer graphics, visualization, and image processing. Implemented in C++, VTK also supports Tcl, Python, and Java language bindings, permitting complex applications, rapid application prototyping, and simple scripts. Although VTK does not provide any user interface components, it can be integrated with existing widget sets, such as Tk or X/Motif [37].

VTK provides a variety of data representations, including unorganized point sets, polygonal data, images, volumes, and structured, rectilinear, and unstructured grids. VTK comes with readers/importers and writers/exporters to exchange data with other applications. Hundreds of data processing filters are available to operate on these data, ranging from image convolution to Delaunay triangulation. VTK's rendering model supports 2D, polygonal, volumetric, and texture-based approaches that can be used in any combination.

VTK consists of two major pieces: a compiled core (implemented in C++) and an automatically-generated interpreted layer.

C++ core**:** Data structures, algorithms, and time-critical system functions are implemented in the C++ core. Common design patterns, such as object factories and virtual functions, ensure portability and extensibility. Because VTK is independent of any graphical user interface (GUI), it does not depend on the windowing system. Hooks into the window ID and event loop allow developers to plug VTK into their own applications.

Interpreted layer**:** While the compiled core provides speed and efficiency, the interpreted layer offers flexibility and extensibility. GUI prototyping tools, such as Tcl/Tk, Python/Tk, and Java AWT, permit the rapid building of professional applications. These popular programming languages come with other packages, such as Python's numerical library, NumPy. The visualization pipeline of VTK can be depicted as Figure 2.7:

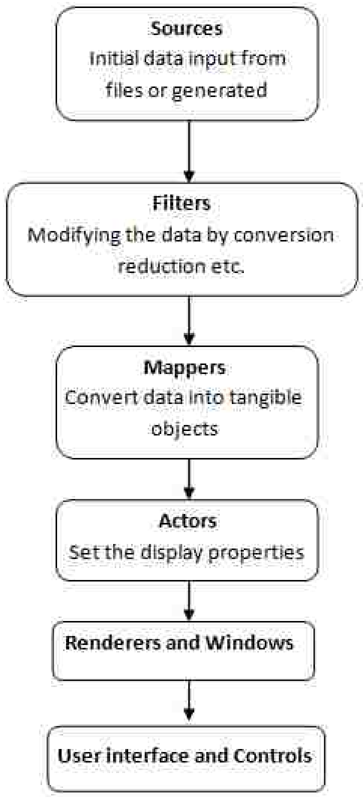

Sources
Initial data input from files or generated
Filters
Modifying the data by conversion reduction etc.
Mappers
Convert data into tangible objects
Actors
Set the display properties
Renderers and Windows
User interface and Controls

Figure 2.7. Visualization Pipeline of VTK

The visualization pipeline of VTK:

- Sources: Sources are quite simply the source of data flowing through the visualization pipeline. There exist two basic types of sources: readers, which read data out of files in a wide variety of formats, and independent sources, which generate a data flow based on input parameters (e.g., a cone source, which generates information describing a cone, given its radius and height). In general, any VTK component that does not receive a flow of data from some other VTK component can be considered a source.

- Filters: Filters are VTK components that receive data from other components, modify the data in some way, and then deliver the modified data as output to be used by other components. Filters may extract some portion of a large data set, subsample data sets to a coarser resolution, interpolate data sets to a finer resolution, merge multiple inputs into a combined output, split compound inputs into component parts, or produce a wide variety of other transformations. User-written procedures also can function as filters.
- Mappers: Mappers are VTK components that receive data from other components (usually filters, but sometimes directly from sources) and "map" the data to some sort of a physical manifestation that can be rendered by the rendering engine.
- Actors: Actors are VTK components that allow the appearance properties of the physical manifestations of the data as rendered onto the screen to be adjusted and controlled. Some of the properties typically controlled via actors are transparency and color mapping. The term "actor" comes from analogy with the stage. The actor is a physical representation of the data "standing" on the "stage" (appearing in the rendering window), who’s appearance can be modified through lighting, makeup, costumes, etc.
- Renderers and Windows: Renderers and windows represent the end of the VTK pipeline, which users actually see on the screen. In practice, there generally is not much that the user must do with renderers and windows, with a few notable exceptions:
    - All actors must be added to a rendering window before they can appear on the screen. Therefore, the renderer usually is created immediately in a

VTK program, even though it comes at the end of the data flow pipeline. Then, each actor can be added to the renderer as that particular pipeline section is completed.

  - VTK components normally do not generate their output until requested to do so. Typically, this is accomplished by requesting that the rendering window render its results. This will cause the renderer to issue an update ( ) request to all of its inputs, which will in turn issue update ( ) requests to all of their inputs, and so on back down the pipeline to the sources. If any parameters in the pipeline change (e.g., in response to user input through the user interface), then the rendering window must be asked to re-render before the effects of the parameter adjustment can be seen.
  - Interactors, which allow users to grab and rotate the rendered data, typically are added along with the renderers and windows.

- User Interface and Controls: Actually, the user interface and controls are not part of the visualization pipeline. In spite of this, these two components play very important roles in the usability of VTK for rendering the real-time objects along with human interaction. All the commands for real-time interaction are placed within the rendering loop of VTK. VTK supports input from mouse, keyboard, etc. Some toolkits have been developed for VTK that provide support for other input devices, such as wands.

3. MOTION CAPTURE SYSTEMS FOR VIRTUAL REALITY

The tracking devices are the main components of the VR system they interact with the system's processing unit, which relays to the system the orientation of the user's point of view. In systems that allow users to roam around within a physical space, the user's location, as well as his direction and speed, can be detected with the help of motion trackers. Various types of motion tracking systems are utilized in VR systems, including the following:

- Six degrees of freedom can be detected (6-DOF).
- Orientation consists of an object's yaw, roll and pitch.
- These are the objects' positions and orientations within the x-y-z coordinates of a space.

All tracking systems consist of a device that is capable of generating a signal, from the sensor. The device also has a control unit, which is involved in processing the signal and sending information to the CPU. Some systems ask the user to attach the sensor to the user or the user's equipment. In this case, the user must position the signal emitters at certain levels in the nearby environment. Differences can be noticed easily in some systems in which the emitters are worn by the users and covered by sensors, attached to the environment. The signals emitted from emitters to different sensors can take various shapes, including electromagnetic, optical, mechanical and acoustic signals.

The various types of tracking devices have the following various merits and demerits:

- Electromagnetic tracking systems: These calculate magnetic fields generated by bypassing an electric current simultaneously through three coiled wires. These wires are arranged perpendicularly to one another. These small structure acts as an electromagnet. The system's sensors calculate how its magnetic field creates an impact on the other coils. The measurement shows the orientation and direction of the emitter. Efficient electromagnetic tracking systems demonstrate excellent responsiveness and low latency levels. The drawback is that whatever can create a magnetic field also can come between the signals, which are sent to the sensors.

- Acoustic tracking systems: These tracking systems sense and produce ultrasonic sound waves to identify the orientation and position of a target. They calculate the time taken for the ultrasonic sound to travel to a sensor. The sensors usually are kept stable in the environment. The user puts on ultrasonic emitters. However, calculating the target's orientation and position depends on the time required for sound to reach the sensors. Acoustic tracking systems contain many faults. Sound travels quite slowly, so the updating of a target's position is naturally slow. The system's efficiency can be affected by the environment as the sound's speed through air often changes depending on the humidity, temperature or barometric pressure found in the environment.

- Optical tracking devices: These devices use light to calculate a target's orientation and position. The signal emitter typically includes a group of infrared LEDs. The sensors consist of only cameras, which can understand the infrared light that has been emitted. The LEDs illuminate in a fashion known as sequential pulses. The pulsed signals are recorded by the camera, and then the information is sent to the system's processing unit, which can extrapolate data. This will estimate the target's position and orientation. The upload rate of optical systems is quite fast, which reduces the tenancy issue. The demerits of the system are that the line of sight between an LED and a camera can be obscured, which interferes with the tracking process. Infrared radiation and ambient light also can make the system useless.

- Mechanical tracking systems: These tracking systems depend on a physical link between a fixed reference point and the target. One of the many examples of a mechanical tracking system is located in the VR field, called a BOOM display. A BOOM display, an HMD, is attached on the rear of a mechanical arm consisting of two points of articulation. Position and orientation detection is accomplished through the arm. The update rate is quite high in mechanical tracking systems, but the demerit is that they limit the user's range of motion.

3.1. TWO-CAMERA VISION SYSTEM

Motion tracking is a prime aspect of camera-based vision systems. Many approaches and methods have been developed to calibrate cameras precisely, including methods developed by Zhang [7], Tsai [6], Svoboda et al. [5], Heikkila and Silven [8]. Previous research has shown that Zhang's method of stereo vision calibration is the most accurate for calibrating intrinsic parameters, while Svoboda's method is efficient for calibrating extrinsic parameters [9]. In the calibration method described by Vader et al. [9], the best features of the two methods discussed above are combined to achieve maximum accuracy. Multi-camera vision systems are calibrated according to the same method. Svoboda's algorithm provides an efficient way to calibrate more than two cameras together under the condition that some common overlap of FOVs of all cameras exists [5].

Pinhole camera model: The basis of calibration is to determine the 3D coordinates of a point from the 2D coordinates obtained from the camera projection plane. To study the calibration process, it is very important to understand the structure of the camera. In a pinhole camera, as shown in Figure 3.1, points are projected on an image plane through perspective projection.

Figure 3.1 depicts a camera with the center of projection O and the principal axis parallel to the Z axis. The image plane is at focal point of the camera; hence, it is at a distance of the focal length 'f', away from the center of projection O. A 3D point P = (X, Y, Z) is imaged on the camera's image plane at coordinate P_c = (u, v).

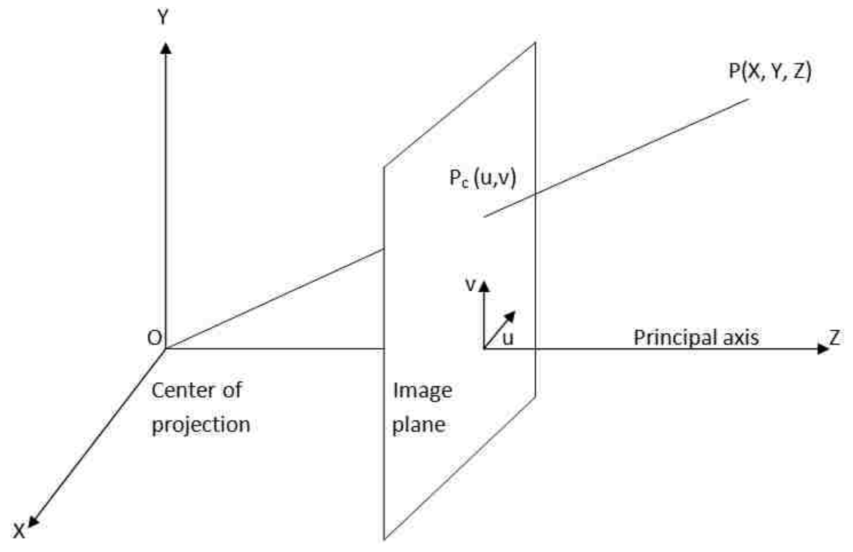


Figure 3.1. Principle of Perspective Projection of a Point on the Camera's Image Plane

First, find the camera calibration matrix C, which maps P in the 3D space to P_c in the 2D image plane. Find P_c using similar triangles, as in:

$$\frac{f}{Z} = \frac{u}{X} = \frac{v}{Y} \tag{1}$$

which yields:

$$u = \frac{fX}{Z}$$

$$v = \frac{fY}{Z} \tag{2}$$

Using homogeneous coordinates for P_c, the equations above can be written as:

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{3}$$

Equation (3) generates the coordinates of point $P_c = (u, v, w) = \left(\frac{fX}{Z}, \frac{fY}{Z}, 1\right)$.

Now, if the coordinates of the image plane's origin do not coincide with the point at which the Z-axis intersects the image plane, P_c will need to be translated with respect to the desired origin. Let the translation be defined by (t_u, t_v), where t_u lies along the u-axis, and t_v lies along the v-axis. Now, a new u and v for P_c is given as:

$$u = \frac{f\mathrm{X}}{\mathrm{Z}} + t_u$$

$$v = \frac{f\mathrm{Y}}{\mathrm{Z}} + t_v \qquad (4)$$

Substituting this new translation in Equation (4) to define the coordinates of point P_c yields a new transformation matrix. The coordinates of point P_c are defined using the following equation:

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} f & 0 & t_u \\ 0 & f & t_v \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \mathrm{X} \\ \mathrm{Y} \\ \mathrm{Z} \end{pmatrix} \qquad (5)$$

Equation (5) defines the coordinates in some unit, such as millimeters, inches, etc. However, for cameras, the unit is the pixel. The camera's resolution defines the number of pixels covered in the image plane. The standard format for expressing resolution is pixels/inch. Let us assume rectangle pixels with a resolution of mu and mv pixels/inch in the u and v directions, respectively. Therefore, to measure P_c in pixels, its u and v coordinates should be multiplied by m_u and m_v, respectively. Thus:

$$u = m_u * \frac{f\mathrm{X}}{\mathrm{Z}} + m_u * t_u$$

$$v = m_v * \frac{f\mathrm{Y}}{\mathrm{Z}} + m_v * t_v \qquad (6)$$

Equations (6) can be expressed in matrix form as follows:

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} m_u * f & 0 & m_u * t_u \\ 0 & m_v * f & m_v * t_v \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} \mathrm{X} \\ \mathrm{Y} \\ \mathrm{Z} \end{pmatrix} = \begin{pmatrix} a_x & 0 & u_0 \\ 0 & a_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} * \mathrm{P} = \mathrm{KP} \tag{7}$$

Note that **K** depends only upon the camera's intrinsic parameters, such as its focal length and principal axis, and thus defines these parameters. Thus, K, a 3x3 matrix, is called the *intrinsic parameters* matrix.

Now, if the camera's center of projection is not at (0 0 0) and is oriented in an arbitrary fashion (not necessarily on the Z axis perpendicular to the image plane), then it must be rotated and translated to make the camera's coordinate system coincide with the configuration in Figure 3.1. Let the camera's translation to the origin of the XYZ coordinate be given by **T** (Tx, Ty, Tz). Let the rotation applied to make the principal axis coincide with the Z axis be given by a 3x3 rotation matrix **R**. Then, the matrix formed by applying first the translation and then the rotation is given by the 3x4 matrix E:

$$\mathrm{E} = (\mathrm{R} \mid \mathrm{RT}) \tag{8}$$

Matrix **E** is called the *extrinsic parameter* matrix. So, the complete camera transformation now can be represented as:

$$\mathrm{K}(\mathrm{R} \mid \mathrm{RT}) = (\mathrm{KR} \mid \mathrm{KRT}) = \mathrm{KR}\,(\mathrm{I} \mid \mathrm{T}) \tag{9}$$

Hence, P_c, the projection of P, is given by:

$$\mathrm{P_c} = \mathrm{KR}\,(\mathrm{I} \mid \mathrm{T})\,\mathrm{P} = \mathrm{CP} \tag{10}$$

C is a 3x4 matrix usually called the *complete camera calibration* matrix. Note that because C is 3x4, P must be in 4D homogeneous coordinates, and P_c derived by CP will be in 3D homogeneous coordinates.

3.1.1. Principle of a Two-Camera Stereo Vision System. Tracking necessitates the shift from one camera to multiple cameras. The calibration procedure of a pinhole camera must be modified when using more than one camera. The basic structure of a vision system is a stereo system, i.e., using two cameras at a known distance. Hence, the geometry of a stereo system must be defined. Figure 3.2 shows the geometry of a stereo vision system.

Two cameras with their image planes and geometry of base line and epipolar line are shown in Figure 3.2 The basic parameters are O_l, the left camera's center (i.e., the center of projection for the left camera), and O_r, the right camera's center (i.e., the center of projection for the right camera). E_l is the left epipoint, and E_r is right epipoint. P_l is the projection of point P on the left image plane, and P_r is the projection of point P on the right image plane. The line joining the points E_l - P_l and E_r - P_r is called the *epipolar line*. The plane passing through the centers of projection and the point in the scene is called the *epipolar plane*. For the stereo vision system, the relationship between the two camera coordinate systems also must be calibrated.

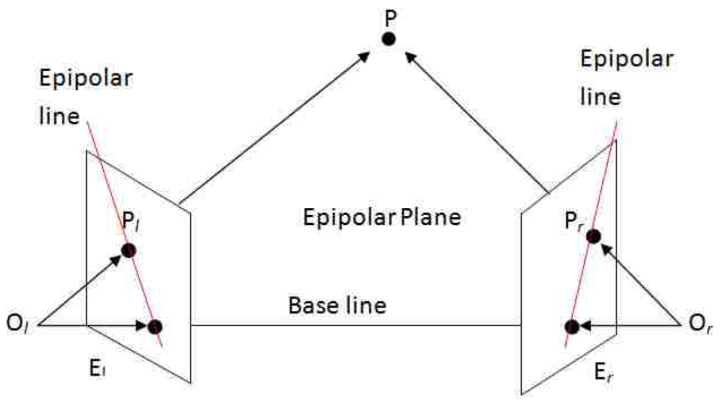


Figure 3.2. Principle of a Stereo Vision System

For a point P in 3D space, its two coordinates, P_L and P_R, in the left and right camera coordinate systems, have the following relationship where, R_s is the rotation matrix, and T_s is the translation vector between the two coordinate frames of the stereo system.

$$P_R = R_s * P_L + T_s \tag{11}$$

3.1.2. Coverage of the Stereo Vision System. Calculating the coverage area is very important in motion tracking systems. Studying the coverage of the stereo system is imperative because the total available area for motion tracking depends on the total common coverage area. The basic common coverage of a stereo system can be calculated using the camera's pyramidal FOV. The basic parameters of pyramidal FOV are known to the users. Consider two cameras in a stereo system at a distance b from each other, with half major angle (α) and half minor angle (β) of the pyramid and a pyramid height h as shown in Figure 3.3.

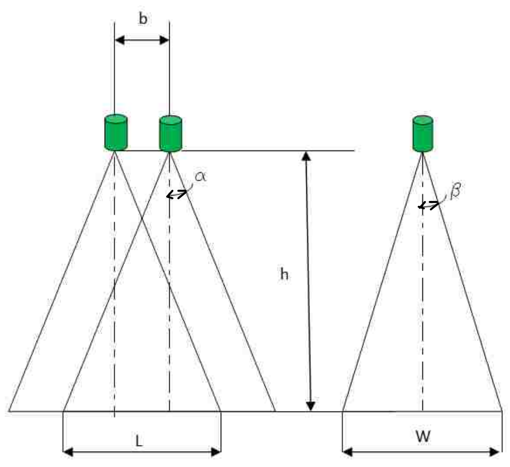


Figure 3.3. Coverage Area of the Stereo System

Now knowing the parameters of pyramidal FOV shown in Figure 3.3, the mathematical expressions for the coverage of stereo systems can be defined, i.e., express **L** and **W** in terms of those parameters, as:

$$L = 2h*\tan(\alpha) - b$$

And

$$sW = 2h*\tan(\beta) \tag{12}$$

The values of L and W are generated considering the wiimote as the camera. For the wiimote, $\alpha = 20.5^o$, $\beta = 15^o$, h = 7m and b = 10cm (0.1m), then yielding L = 5.1m and W = 3.75m. So, this particular stereo system's area of coverage is 5.1m X 3.75m. Using the same equations (12), the coverage of any stereo system can be determined. In the toolbox, a set of visible points is used to determine the coverage area because the cameras are not oriented in same direction. However, this method is very important for standard coverage calculations for stereo vision systems.

3.2. MULTI-CAMERA VISION SYSTEM

For wide-area motion tracking purposes, it is necessary to shift from a stereo system to a multi-camera vision system. Some theories have been developed to calibrate multi-camera vision systems [5][6][10]. Previous research in this area has shown that the calibration method suggested by Svoboda is the most effective for the cameras used in the experiments [9] [5]. The method offers a way to calibrate more than two cameras simultaneously, forming a single vision system for motion tracking. It is evident that more cameras provide a larger coverage area. Hence, while considering the development of a new technique for wide-area motion tracking system implementation, it is important to use this algorithm for defining and calibrating multiple vision systems. The only necessary condition for implementing this algorithm is for the cameras' FOVs to overlap during calibration. Then, for motion tracking a point must be visible to at least two cameras simultaneously at any given moment, for comparatively accurate position tracking [9] [1].

Principle of Multi-Camera Vision System Implementing Svoboda's Algorithm

The multi-camera vision system is based on the multi-camera self-calibration technique suggested by Svoboda [5]. The algorithm is briefly described below

Consider **m** cameras and **n** object points; it is assumed that the pinhole camera model is valid. The points X_j are projected to the image points u_{ij}:

$$X_j = [X_j,\ Y_j,\ Z_j,\ 1]^T,\ j = 1, \ldots, n \qquad (13)$$

$$l_{ij} \begin{pmatrix} u_{ij} \\ v_{ij} \\ 1 \end{pmatrix} = l_{ij}\, u_{ij} = P_i\, X_j, \qquad l_{ij} \text{ v } R^{+},\ i = 1, \ldots, m \qquad (14)$$

For m points, this equation can be expressed as following where each P_i is a 3x4 matrix for ith camera that contains 11 camera parameters.

$$\begin{bmatrix} \lambda_{11} \begin{bmatrix} u_1^1 \\ v_1^1 \\ 1 \end{bmatrix} \ldots\ldots \lambda_{1n} \begin{bmatrix} u_n^1 \\ v_n^1 \\ 1 \end{bmatrix} \\ \ldots\ldots \\ \lambda_{m1} \begin{bmatrix} u_1^m \\ v_1^m \\ 1 \end{bmatrix} \ldots\ldots \lambda_{mn} \begin{bmatrix} u_n^m \\ v_n^m \\ 1 \end{bmatrix} \end{bmatrix} = [P_1 \ldots\ldots P_m]^T_{3mx4} * [X_1 \quad \ldots. \quad X_n]_{4Xn} \qquad (15)$$

These parameters depend on the six DOF that describe the camera's position and orientation and on the five intrinsic parameters. The u_{ij} are the observed pixel coordinates. The goal of the calibration is to estimate the scales λ_{ij} and the camera projection matrices P_i shown as Figure 3.4. The wrong collected calibration points can be detected by the calibration analysis specified in the calibration process described in RANSAC [10]. Hence, the coordinates of these wrongly collected points can be estimated by the method described by Svoboda et al. [5]. All the outputs ($P_i X_j$) from the equation can be put into one matrix, W_s, as:

$$W_s = \widehat{P}\widehat{X} \tag{16}$$

where W_s is called a *scaled measurement matrix.*

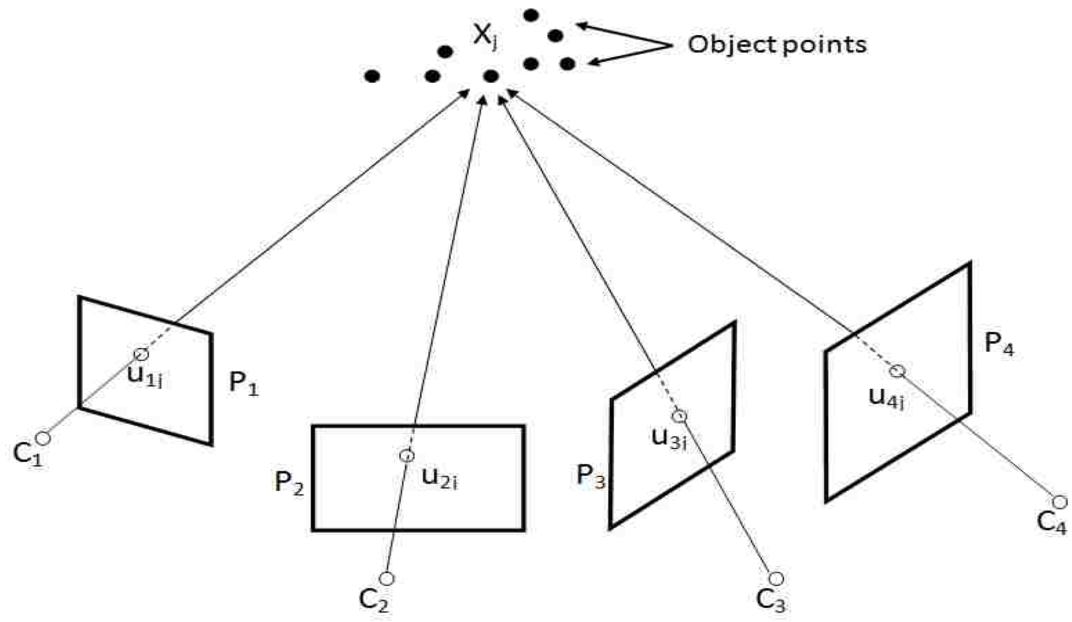


Figure 3.4. Schematic Structure of 4-Camera Setup Implementing Svoboda's Algorithm

Integration of multi-camera vision systems: Setting up a multi-camera vision system for a wide area is a classical issue. Extensive work has been done in this area, but mainly for surveillance purposes [19][20][21][22]. A key method used in previous research was to apply position and motion tracking systems for tracking people based on blob detection using image processing techniques through regular surveillance cameras [22]. The basics of calibrating the cameras are the same, i.e., based on the pinhole camera model and calibration processes [5][6][7]. Wide-area position tracking is implemented by calibrating the cameras separately or calibrating a specific camera vision system for a specific area. For this purpose, researchers primarily have used Zhang's and Svoboda's algorithms, whose implementations have been discussed. The surveillance camera tracking system primarily is an image-based tracking system. The main consideration for this kind of tracking systems is that the background objects remain in the same position. Hence, the tracking is implemented depending upon the movement of objects detected with a fixed background. Movement is detected using image processing applications. The main objective of detecting these blobs is not to position the objects in some world coordinate system but to form a topography of persons or objects [21][22]. In past research, the need had arisen to find an inexpensive method of position and motion tracking. Therefore, author has implemented Wiimotes as cameras for tracking purposes, which is inexpensive compared to professional tracking systems.

Based on the objective of wide-area tracking by surveillance cameras,define the main difference between optical sensor position tracking systems and wide-area surveillance tracking as that the former system is deployed for absolute position tracking in a world coordinate system. The primary advantage of an optical tracking system is that it can determine the absolute position of an object in a world coordinate system even with changing backgrounds. This objective is very helpful as it allows this tracking system to be implemented at any place and for any process. Hence, if a method can be defined for integrating and calibrating a large number of cameras for position tracking, an optical tracking system for wide-area position tracking can be used.

For the purpose of forming a vision system, implementing Svoboda’s algorithm for calibration provides an efficient way to calibrate multiple camera vision systems. The most important aspect of this calibration method is that there should be some common overlap between the FOVs of all the cameras [5] [1]. Using an optical tracking system for position tracking requires either a common overlap between FOVs or some way of integrating several vision systems to form a single vision system. Here, author has proposed and implemented a method for integrating multiple vision systems that involves applying Horn’s algorithm for finding closed-form solutions for absolute orientations based on quaternions, as stated in [9] [25].

Implementing Horn’s algorithm**:** To integrate multi-camera vision systems, a method of finding the absolute orientation between Cartesian coordinate systems defined for particular vision systems is implemented. The primary condition for implementing Horn’s algorithm is that the Cartesian coordinate systems are to be defined for all the vision systems. The previous section in which the implementation of Svoboda’s algorithm was explained also contained an explanation of how a coordinate system is defined for a vision system. For the condition in which no common overlap exists between the cameras’ FOVs, try to form some kind of mathematical relationship between the Cartesian coordinate systems of vision systems. For the transformation matrices generated, the transformation matrix must be found in order to transform the coordinates of a point in one Cartesian coordinate system to the other Cartesian coordinate systems. This transformation matrix again is a combination of a rotation matrix and a translation matrix. A method for generating this transformation matrix from the coordinates of three or more points in two different coordinate systems of two different vision systems is described by Horn [25]. In this paper, a closed-form solution for absolute orientation is generated using quaternions. Implementing this solution requires coordinates of at least three points in two Cartesian coordinate systems. Hence, the absolute orientation gives a rotation matrix to align one Cartesian coordinate system to other Cartesian coordinate systems. This method solves the equations iteratively. The aim of this method is to minimize the transformation's residual error [25]. The transformation between two Cartesian coordinate systems can be thought of as the result of a rigid-body motion and thus can be decomposed into a rotation and a translation. There are three degrees of freedom for translation, and the rotation and scaling factors each add one degree of freedom. Hence, there are seven degrees of freedom in all. Three points each have X, Y and Z coordinates in both Cartesian coordinate systems. Therefore, there are nine equations and seven parameters to determine from these equations.

Figure 3.5 explains the basis of this algorithm's implementation. Two Cartesian coordinate systems, '*Left*' and '*Right,*' have been defined. The axes of these systems are X_l, Y_l, Z_l and X_r, Y_r, Z_r, respectively. There are four points visible to both the coordinate systems. Let the coordinates of the points be defined as $P_{l,1}$, $P_{l,2}$, $P_{l,3}$, …, $P_{l,m}$ in the *Left* Cartesian coordinate system and $P_{r,1}$, $P_{r,2}$, $P_{r,3}$, …, $P_{r,m}$ in the *Right* Cartesian coordinate system for 'm' points.

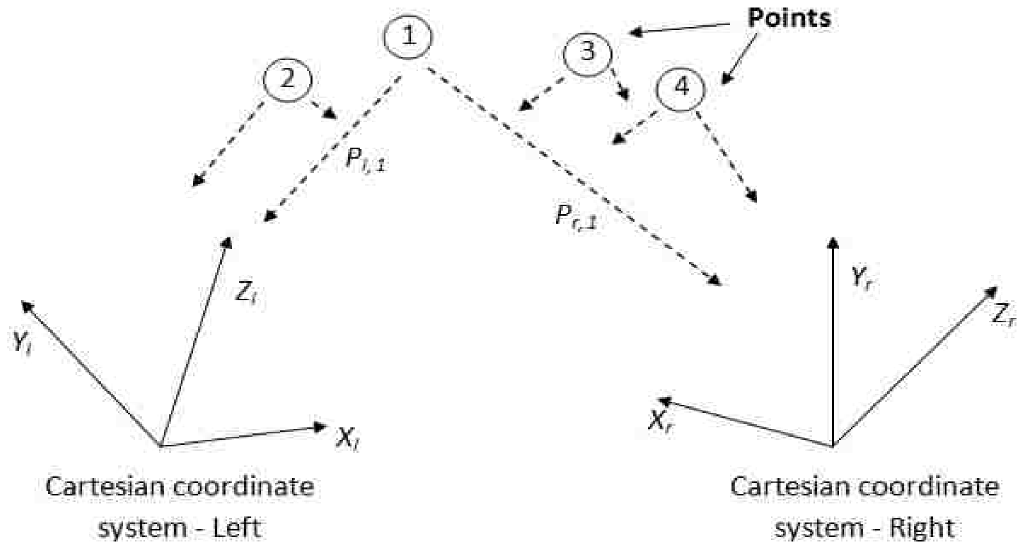

Points
1
2
3
4
$P_{l,1}$
$P_{r,1}$
Z_l
Y_l
X_l
Y_r
Z_r
X_r
Cartesian coordinate system - Left
Cartesian coordinate system - Right

Figure 3.5. The Coordinates of a Number of Points is Measured in Two Different Coordinate Systems

Consider:

$$x_l = r_{l,2} - r_{l,1} \tag{17}$$

Then:

$$\hat{x}_l = x_l / \| x_l \| \tag{18}$$

Equation (18) defines the unit vector in the direction of the new X axis in the *Left* coordinate system. Now, let:

$$y_l = (P_{l,3} - P_{l,1}) - [(P_{l,3} - P_{l,1}).\ \hat{x}_l]\ \hat{x}_l \tag{19}$$

Here, three points define a triad in the *Left* coordinate system. Similarly, a second triad can be constructed in the *Right* coordinate system. The required coordinate transformation can be estimated by finding the transformation that maps one triad onto the other as shown in Figure 3.6.

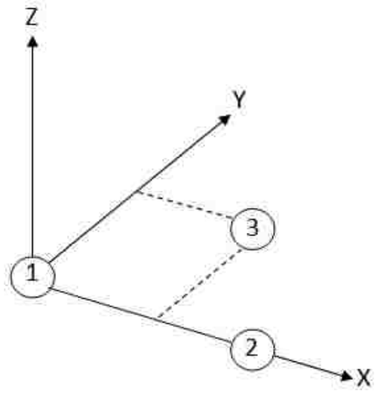


Figure 3.6. Three Points Define a Triad

The unit vector along the new y axis can be defined as:

$$\hat{y}_l = y_l / \| y_l \| \quad (20)$$

For the *Left* coordinate system, the third unit vector along the new z axis can be defined as:

$$\hat{z}_l = \hat{x}_l \text{ X } \hat{y}_l \quad (21)$$

In the same way, the unit vectors for the *Right* coordinate system can be defined as $\hat{x}_r$, $\hat{y}_r$, $\hat{z}_r$. Using these column vectors, M_l and M_r can be defined as:

$$M_l = | \hat{x}_l \ \hat{y}_l \ \hat{z}_l | \text{ and } M_r = | \hat{x}_r \ \hat{y}_r \ \hat{z}_r | \quad (22)$$

$$M_l^T r_l \quad (23)$$

Multiplication by M_r then maps these into the right-hand coordinate system yields the components of the vector r_l along the axes of the constructed triad.

$$r_r = M_r M_l^T r_l \quad (24)$$

$$R = M_r M_l^T \quad (25)$$

Hence, the total rotation matrix, i.e., the sought after rotation, is given by. The result is orthonormal because M_r and M_l are orthonormal by construction. Equations (25) constitute a closed-form solution for finding the rotation, given three points. Now, the translation must be found. For *n* points, the coordinates in the *Left* and *Right* coordinate systems are { $P_{l,\,i}$ } and { $P_{r,\,3}$ }, where i ranges from 1 to *n*. The transformation of the coordinates from the *Left* to the *Right* coordinate system takes the form in which s is the scale factor and r_0 is the translational offset:

$$P_r = sR(P_l) + r_0 \tag{26}$$

Here, the rotation is linear, and length is preserved so that where, $||P||^2 = P.\ P$ is the square of the length of vector P.

$$||R(P_l)||^2 = ||P_l||^2 \tag{27}$$

Here, because the data is erroneous, it will not be possible to find a scale factor, a translation, and a rotation that satisfy the transformation equation above for each point. There will be some difference in terms of the actual calculation; this difference is called *residual error* e_i:

$$e_i = P_r - sR(P_l) - r_0 \tag{28}$$

The algorithm tries to minimize this residual error:

$$\sum_{i=1}^{n} ||\ e_i\ ||^2 \tag{29}$$

The centroids of the coordinate systems are defined as:

$$\bar{P}_l = \frac{1}{n} \sum_{i=1}^{n} P_{l,i} \quad \text{and} \quad \bar{P}_r = \frac{1}{n} \sum_{i=1}^{n} P_{r,i} \tag{30}$$

$$P'_{l,i} = P_{l,i} - \bar{P}_l \text{ and } P'_{r,i} = P_{r,i} - \bar{P}_r \tag{31}$$

$$\sum_{i=1}^{n} P_{l,i} = 0 \text{ and } \sum_{i=1}^{n} P_{r,i} = 0 \tag{32}$$

The error in terms of new coordinates is written as:

$$e'_i = P'_r - sR(P'_l) - r'_0 \qquad (33)$$

$$r'_0 = r_0 - \overline{P}_r + sR(\overline{P}_l) \qquad (34)$$

The sum of squares of errors becomes:

$$\sum_{i=1}^{n} \| P'_r - sR(P'_l) - r'_0 \|^2 \qquad (35)$$

In Equation (35), the middle term becomes zero, as explained earlier in Equation (32). This error is minimized when $r'_0 = 0$. The equation then reduces to:

$$r_0 = \overline{P}_r - sR(\overline{P}_l) \qquad (36)$$

The main inference drawn from this equation is that *the translation is just the difference between the right centroid and the scaled and rotated left centroid.* Because $r'_0 = 0$, the total error in Equation (36) is minimized to:

$$\sum_{i=1}^{n} \| P'_{r,i} - sR(P'_{l,i}) \|^2 \qquad (37)$$

The next step is to estimate the scale factor s. Expanding the total error in Equation (37) yields:

$$\sum_{i=1}^{n} \| P'_{r,i}\|^2 - 2s\sum_{i=1}^{n} P'_{r,i} \cdot R(P'_{l,i}) + s^2\sum_{i=1}^{n} \|P'_{l,I}\|^2 \qquad (38)$$

Equation (38) is of the form:

$$S_r - 2sD + s^2S_l \qquad (39)$$

where $\boldsymbol{S_r}$ and $\mathbf{S_l}$ are the sums of the squares of the measurement vectors (relative to their centroids), $\boldsymbol{D}$ is the sum of the dot products of corresponding coordinates in the right system with the rotated coordinates in the left system. Completing the squares in terms of s yields:

$$(s\sqrt{S_l} - D/\sqrt{S_l})^2 + (S_rS_l - D^2)/S_l \qquad (40)$$

Using this equation, the error can be minimized with regard to the scale factor s when the first term of Equation (40) is zero, i.e., s = D/S$_l$, and then the expression for scale factor s is found as:

$$s = \sum_{i=1}^{n} r'_{r,\,i} \,.\, R(r'_{r,i}) \,/\, \sum_{i=1}^{n} ||r'_{r,\,i}||^2 \tag{41}$$

3.3. ACCURACY MEASUREMENT OF MOTION CAPTURE SYSTEMS

Another important aspect of vision systems is the accuracy of their motion tracking. The traditional method of determining the system's accuracy is to compare the distance between two markers, e.g., two LEDs at a known fixed point. This method is explained and demonstrated in various papers [17] [18]. Basically, accuracy is measured as the difference between the calculated distance and the known distance. Let **D1** be the known fixed distance between two IR LEDs and **D2** be the calculated distance between two points generated from the motion tracking system. The calculated error d can be plotted in MS Excel, and the average error of the particular multi-camera vision system can be determined. There are two methods of representing the error, either by finding the average d_{avg} or by finding the root mean square (RMS) value d_{rms}:

$$D2 = \sqrt{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2} \tag{42}$$

In which d_{avg} = average (D2 – D1) mm, where d is the calculated error in mm:

$$d_{rms} = \frac{\sqrt{\sum D2^2}}{N}, \text{ N = Number of observations} \tag{43}$$

In some cases, researchers find the percentage deviation of the calculated distance to the actual distance.

4. MULTI-CAMERA VISION SYSTEM SETUP TOOLBOX

The aim of the multi-camera setup is to track a moving object within a certain volume without any data loss. For this purpose, the position of an object must be determined by first determining the position of an infrared LED. The method described in [10] is used to calibrate the vision system. This means that all positions within this volume must be visible by at least two cameras at all times. To achieve this objective, it is essential to place the cameras at correct positions and with correct orientations. The problem is how to evaluate the positions and orientations of these cameras such that the total coverage is attained. One method by which to determine the area covered by a multi-camera vision system is to manually find all the points visible to the tracking system to determine the total area out of a given volume to be tracked. This method is time consuming, extensively inaccurate and very difficult to validate. This thesis suggests a mathematical method to determine all points inside a given volume that are visible to at least two cameras simultaneously at any given time. This is a mandatory condition for accurate position tracking [5].

4.1. TOOLBOX ARCHITECTURE

The system designed for multi-camera setup can accept any random position and orientation of a camera. Research conducted in the past had some constraints. The first of which was that all cameras in the simulation have fixed orientations. Those systems could not select the orientation and position of the cameras. The second constraint involved choice in selecting the camera. A previous system was built for Wiimotes only [9]. It was essential to design a system that could incorporate any type of camera. The key points of the proposed system are that it has the ability to accommodate the input of a random number of any types of cameras with random positions and orientations.

4.1.1. Modules in the Toolbox. The multi-camera vision system setup toolbox has two modules: 1) the MATLAB module, and 2) The Visualization Toolkit (VTK) module. The MATLAB module specifically performs the mathematical functions of the system, and the VTK module performs the visualization functions of the system. A Figure 4.1 briefly describes the flowchart of the whole toolbox.

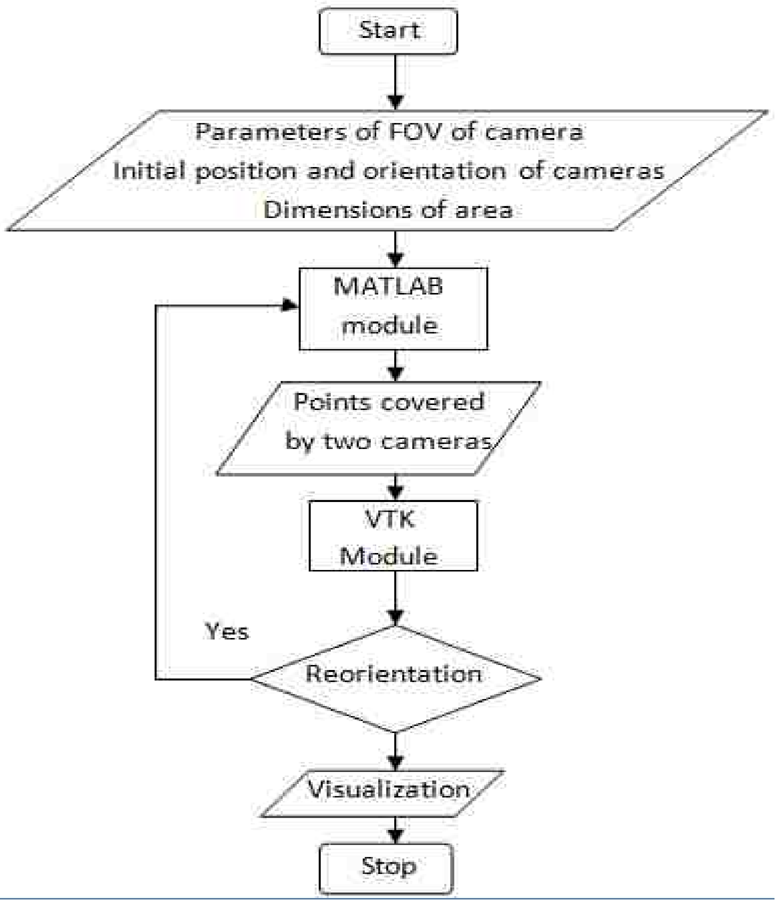

Start
Parameters of FOV of camera
Initial position and orientation of cameras
Dimensions of area
MATLAB module
Points covered by two cameras
VTK Module
Yes
Reorientation
Visualization
Stop

Figure 4.1. Flowchart for Multi-Camera Vision System Setup Toolbox

4.1.2. Flowchart for Data Flow and Output for MATLAB. The MATLAB module contains the functions and codes for calculating the visibility of the points based on the pre-set parameters of cameras. The output of the MATLAB module is the set of points visible to the multi-camera vision system. The total visible points are written in a data file (.dat) along with the initial parameters of cameras, i.e., positions, orientations and angles of FOVs. Figure 4.2 shows the flowchart for data flow in the MATLAB module.

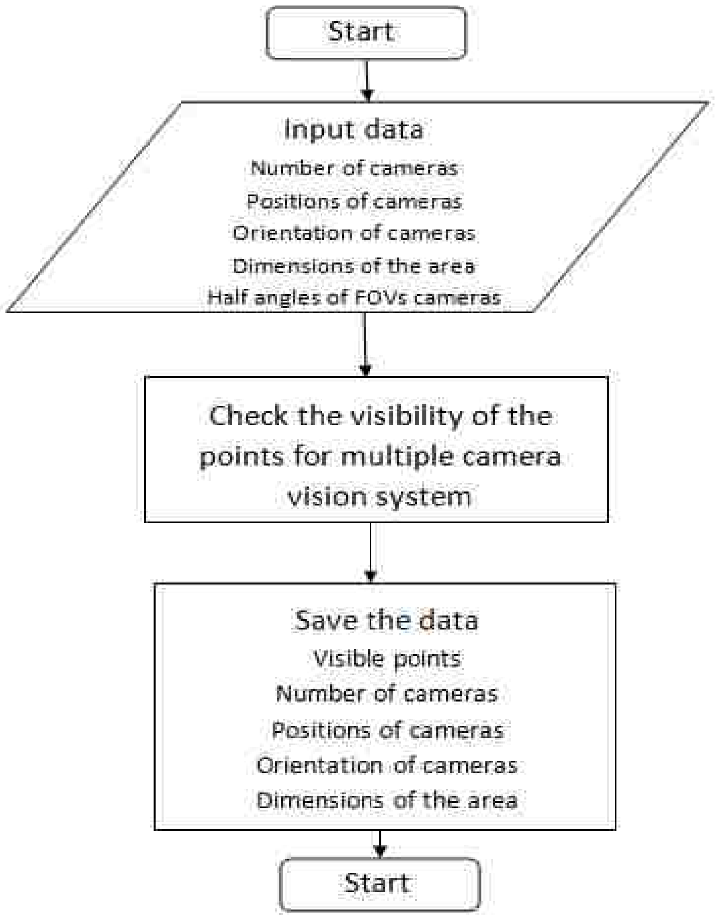


Figure 4.2. Flowchart for Data Flow and Output for MATLAB

4.1.3. Flowchart for Data Flow and Output for VTK. The main function of the VTK module of the toolbox is to visualize the whole system. The input is provided to the VTK code through a data file in which the visible points are saved as shown in Figure 4.3. Based on the FOV parameters and the initial positions and orientations of the cameras, the entire system is visualized in a window.

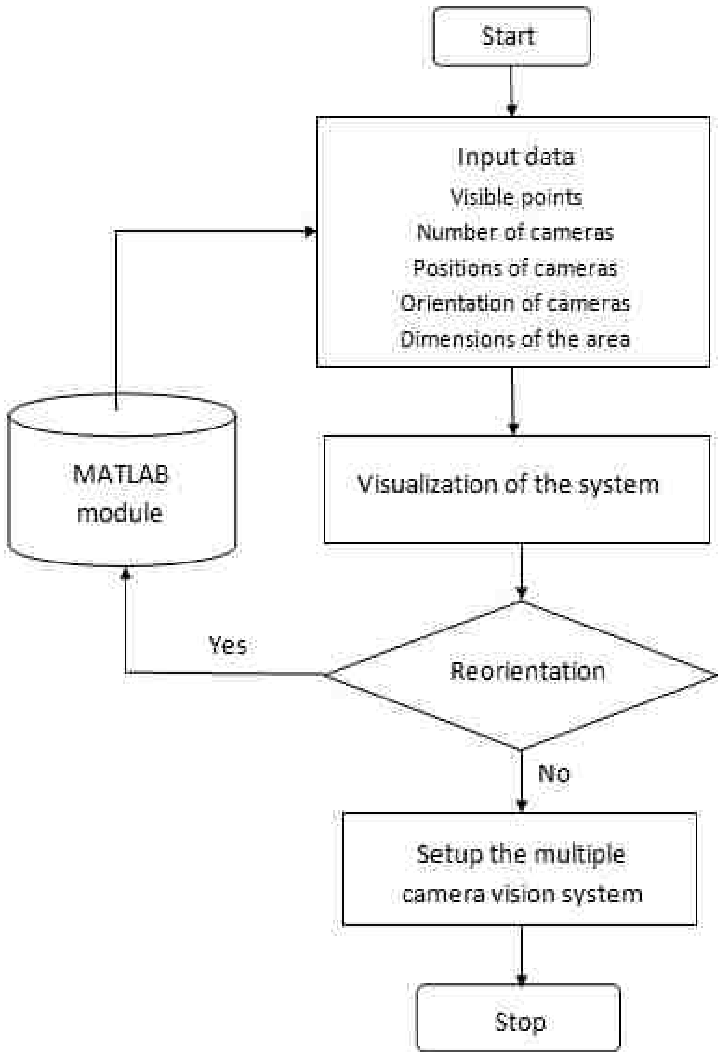


Figure 4.3. Flowchart for Data Flow and Output for VTK

4.2. MATLAB MODULE

4.2.1. Matlab Module Inputs. The user must provide specific inputs to the toolbox in order to use it. These inputs are basic, simple and can be determined easily. The toolbox is designed for ease of use. The units used are mm for distance and degrees for angles. The basic inputs for the toolbox are as follows:

- Type of camera: The type of camera is defined by its FOV, which is pyramidal. Hence, the inputs are:
    - Half angles of the pyramidal FOV
    - Height of the pyramidal FOV
- Dimensions of the area to be tracked: The user must input the dimensions of the area to be tracked, i.e., the total available area. The dimensions along the X, Y and Z directions (length x height x width) are provided in mm.
- Position of cameras in the XYZ coordinate system: The XYZ coordinates of each camera are provided by the user as the input for the toolkit. XYZ coordinates are provided in mm.
- Orientation of cameras as the direction vector: The toolkit is designed for any number of cameras placed anywhere and with any orientation. The center axis of the pyramidal FOV is aligned with this direction vector.

4.2.2. Visibility Calculations in MATLAB. This thesis has explained how the visibility check is applied for the points inside the given volume. Previous research has revealed that the FOV of a camera is always considered pyramidal. Hence, it becomes very important to understand the geometry of pyramid in detail. In an earlier section, the geometry of a pyramid was explained in terms of vector geometry as well as classical geometry. The next step is to check the visibility. Finding the points visible inside a pyramid is a classic problem. Although a pyramid can be defined mathematically, the rectangular base of the pyramid makes this a difficult task. Various researchers and mathematicians have explained different methods by which to find the visibility of a point inside a given geometry [13][14]. Author has attempted to use the method of defining a cross-sectional plane on the center axis of the pyramid at a distance from the apex of the pyramid equal to the absolute distance between the point to be checked and the apex. Then, using the algorithm described by Burke [13], author has designed a code to determine if the point lies inside the given pyramid. The first step in this method is to describe the FOV of the particular camera, which is pyramidal with a rectangular base. Only three parameters are needed to describe a pyramid, the pyramid's half major angle (α), half minor angle (β) and height (h) as shown in Figure 4.4. Depending upon the defined coordinate system, the FOV of the camera can be determined.

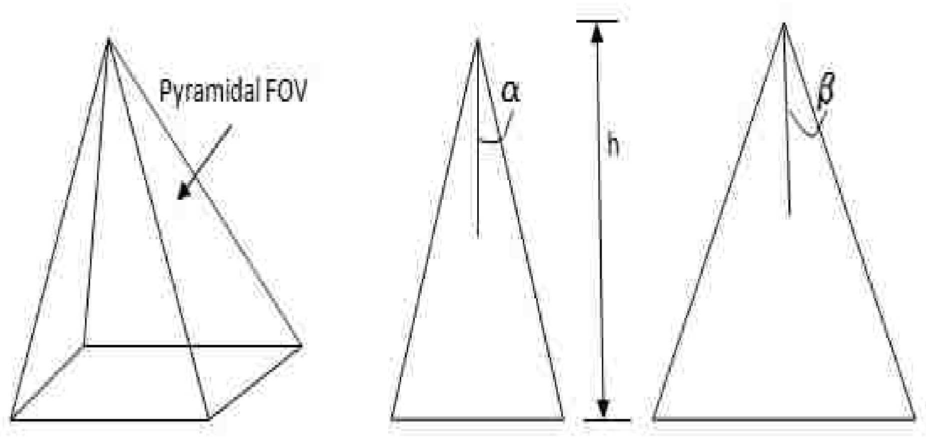


Figure 4.4. Basic Geometry and Parameters of the Pyramidal FOV of the Camera

Based on this information, the pyramidal FOV can be defined in parametric form. For this purpose,begin with the basic definition of a line in parametric form. For a line segment defined by two points, **A** and **B,** in a 3D coordinate system, the unit vector **u** that defines the direction of the line and a scalar parameter **t** that ranges from 0 to 1 starting from point A (t = 0) to B (t = 1) can be determined. The parametric representation of the line helps in determining any point **C** between A and B as shown in Figure 4.5, if the value of parameter t is known.

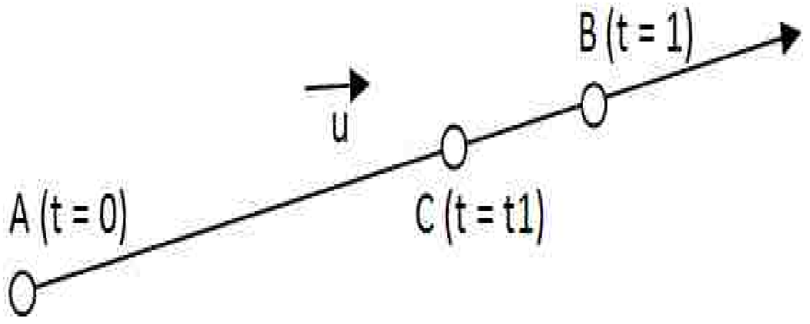


Figure 4.5. Parametric Representation of a Line

$$C = A + t1*u \quad \text{-> } \textit{Parametric equation of the line} \tag{44}$$

Information regarding the angles, the height of the pyramid and the parametric representation of a line will be used to define a pyramid and its rectangular base. The coordinate system is defined by the user, and the height of a pyramid is the maximum length of the camera's visibility (e.g., the height for a wiimote is 7m). The user-defined coordinate system is defined by X, Y and Z. The new coordinate system defined after random orientation is X_n, Y_n and Z_n. The pyramid's base is rectangular, as is the cross section of a pyramid by a plane parallel to the base. This rectangular, cross-sectional pyramid is defined by four points, P_1, P_2, P_3 and P_4. The apex of the pyramid is **O,** and center point of the pyramid's base is **P.** Initially, the pyramid is placed in such a way that its center axis aligns with the Y axis, and O is at its origin. The unit vector on which the center of the rectangular base lies is [0 1 0]'. Depending on the unit direction vector, the rotation matrix **R**(3x3) is formed. After rotation, the center of the rectangular base is transformed to point P. The newly transformed center axis is defined by vector **OP.** Using this vector, the unit vector along OP can be defined, thus allowing the detection of any point on the center axis using the parametric representation of the line, as in Equation (44). While defining the placement of the camera, the axis of the pyramidal FOV is aligned to the direction vector, which is used to form a rotation matrix **R**. Then, using the coordinates of the camera, the translation matrix **T** (3x1) can be defined. Combining the rotation and translation matrices allows the transformation matrix **TR** (3x3) for a given camera to be defined. This transformation matrix will be used to define the placement of a camera and hence a description of the pyramidal FOV. By applying this transformation on the pyramidal FOV of the camera along with the parameters of FOV, i.e., half major angle and half minor angles, the position and geometry of the FOV are precisely defined. Figure 4.6 shows an oriented FOV along with its parameters. The rectangle at the bottom of the FOV shows a cross section of the pyramidal FOV by a plane parallel to its base.

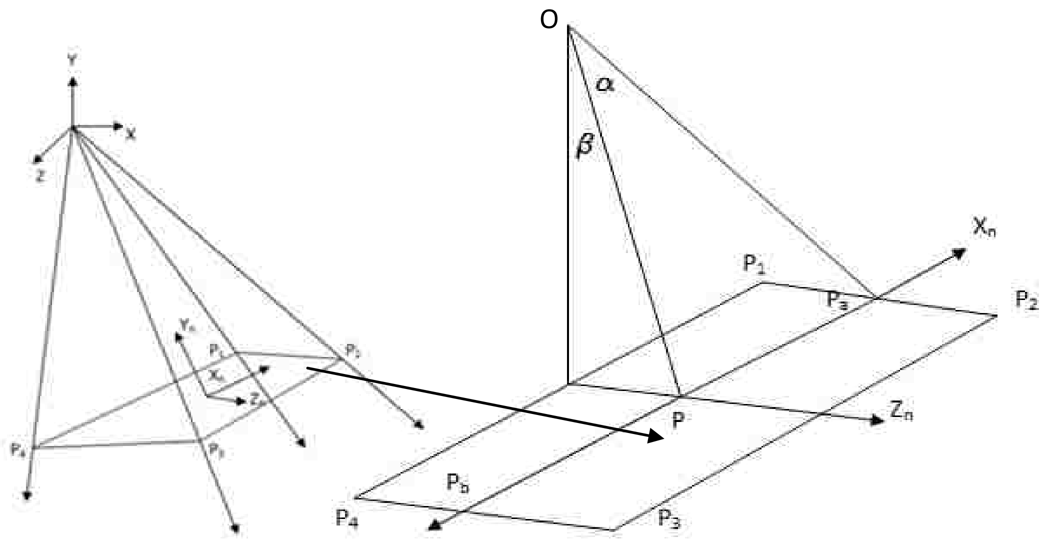


Figure 4.6. Oriented FOV and Cross-Section Plane

Visibility of a point: The effectiveness of a camera-based tracking system has a direct relationship with the visibility of a point for cameras. According to calibration methods defined by Svoboda [5], a point in a given volume should be visible to two cameras at any given time for efficient position tracking. To define a mathematical method for the issue of visibility, author has defined several points in the given volume. Points are defined on the X, Y and Z axes at specific distances. The dimensions of the given volume are dimX, dimY and dimZ. The points inside this given volume are defined using the nested loops in MATLAB.

After performing these loops, each point Pt inside the given volume is defined. Now, after defining the points, their visibility must be checked for each camera. In other words, the visibility of a point is checked by determining whether that point lies inside the pyramidal FOV of the camera. Here, use the description of the pyramidal FOV. Based on the description in Figure 4.7, a rectangular cross sectional area is defined using points P_1, P_2, P_3 and P_4.

```
for i = 0 : d : dimX  -> along X axis
    for j = 0 : d : dimY -> along Y axis
        for k = 0 : d : dimZ -> along Z axis
        Point Pt = [i j k];
        end;
    end;
end;
```

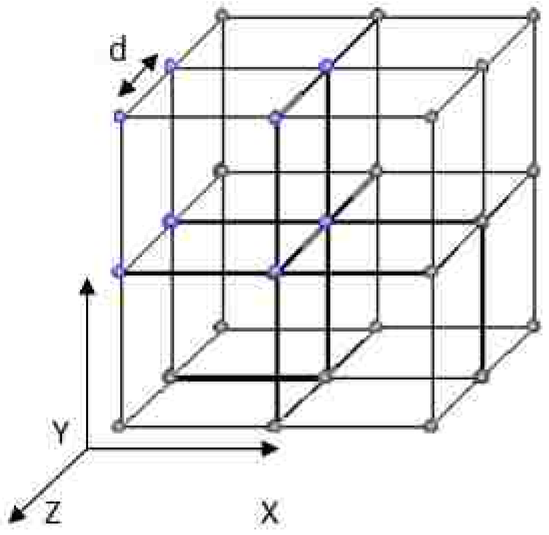


Figure 4.7. Points Inside a Given Volume in X, Y and Z Axis Direction

The next step is to determine if a point lies inside the camera's FOV. It is possible to define the points P_1, P_2, P_3 and P_4 using the center of the rectangular cross section P and the half major angle and half minor angle. Referring to Figure 4.6, the new coordinate system using transformation is defined:

$$X_n = TR*X\ ,\ \ Y_n = TR*Y \text{ and } Z_n = TR*Z \tag{45}$$

To determine the parallel distance of any given point Pt in the given volume from the pyramid's apex O, a 3D line from point Pt perpendicular to line OP must first be drawn. The basic mathematics behind drawing this line are inspired from the method of finding the perpendicular distance between a point and a line [11]. For this purpose, a perpendicular line is drawn from any point Pt to line OP, as shown in Figure 4.8. The point of intersection is point P. This new coordinate system has point P as its origin. Then, points P_a and P_b along the $+X_n$ and $-X_n$ axis, respectively, are defined, and these points lie on the edges of the rectangular cross section.

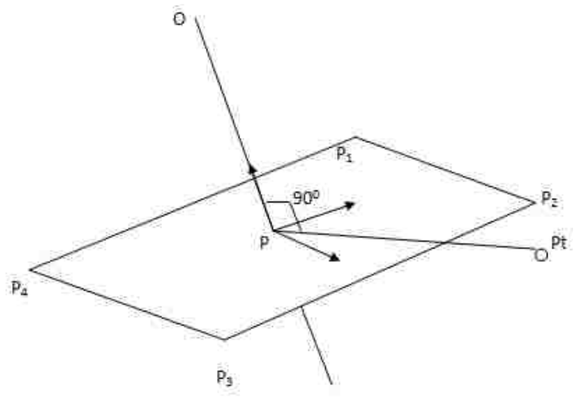


Figure 4.8. Cross-Section Plane Generated at Point of Intersection P

Using parametric representation, points P_a and P_b are defined as where m = the unit vector along vector OP.

t = dot(OP,OPt)./norm(AB) -> *determine scalar t to locate P on line OP* (46)

P = O + t.*m -> *locate P on line OP* (47)

To define the vertices of the rectangular cross section, the two points P_a and P_b must first be defined, as discussed earlier and shown in Figure 4.6. These points will be helpful in defining the vertices of the rectangle. To define these two points along the X_n and Z_n axes, respectively, the scalar parameters t1 and t2 along X_n and Z_n first will need to be determined. Given half major angle and half minor angle b of the pyramidal FOV:

t1 = t.*tan((a).*pi./180)

t2 = t.*tan((b).*pi./180)

$P_a = P + t_1*X_n$

$P_b = P - t_1*X_n$ (48)

where X_n and Z_n are unit vectors defining the new coordinate system. At this point, some very useful information has been obtained about the coordinates of points P_a and P_b and the unit vectors X_n, Y_n and Z_n defining the coordinate system of the pyramidal FOV. The coordinates of vertices P_1 and P_2 can be determined using P_a as the base point and t2 as the scalar parameter, locating P_1 along the $+Z_n$ and P_2 along the $-Z_n$ axis. Similarly, the coordinates of points P_3 and P_4 along $+Z_n$ and $-Z_n$ can be determined using point P_b as the base point. The following MATLAB process demonstrates the procedures for defining the vertices of a rectangular cross section:

P1 = Pa + t1.*Zn

P2 = Pa – t1.*Zn

P3 = Pb – t1.*Zn

P4 = Pb + t1.*Zn (49)

Here, the coordinates of the vertices in P1, P2, P3 and P4 are stored.

This process is performed for each point Pt in the given volume to determine if it lies inside this rectangular area. Simply put, a cross section at the point of intersection of the center line of the pyramid and the perpendicular line drawn from any random point Pt is defined. The plane of the cross section is perpendicular to the center line. Hence, the point Pt and the cross-section plane lie in the same infinite plane. Therefore, whether that point lies inside that rectangular cross section can be determined mathematically.

A different algorithm is implemented to determine if the point lies inside the given rectangular area. To proceed further, the right-hand rule of the cross product must be recalled [12]. Considering the two vectors U and V in a plane, the order of vectors in the cross product determines the direction of the cross product vector W, as shown in Figure 4.9.

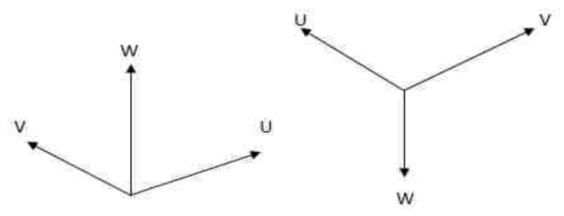


Figure 4.9. Directions of Cross Product Vector W for Different Orders of U & V

The next step is to solve the problem of determining if a point lies in the interior of a polygon. Many approaches have been used to solve this problem, such as ray-casting, convex hull and determining on which side of a given line the point lies [13][14].

Amongst these methods described by Foley [13] and Burke [14], this study employs the latter. A rectangular cross section is a convex polygon. The methods described in [13] and [14] work with 2D coordinate systems, but no methods are described for 3D systems. In this study, the logic behind the algorithm is extended, and a new algorithm a for 3D coordinate system is developed. This new algorithm combines vector mathematics and cross-product rules.

Algorithm statement: If the polygon is convex, then one can consider the polygon as a "path" from the first vertex. A point lies in the interior of these polygons if it is always on the same side of all the line segments making up the path. See Figure 4.10.

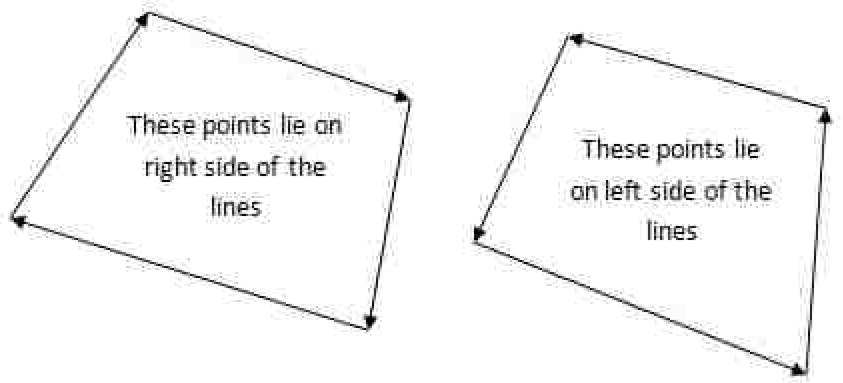

These points lie on right side of the lines
These points lie on left side of the lines

Figure 4.10. Points Lying Inside a Convex Polygon Depending on Clockwise & Counterclockwise Order

Execution in 2D: Given a line segment between P_0 (x_0,y_0) and P_1 (x_1,y_1), another point P (x,y) has the following relationship to the line segment:

Compute:

$$\text{CProduct} = (y - y_0)(x_1 - x_0) - (x - x_0)(y_1 - y_0) \tag{50}$$

CProduct is nothing but the cross product of PP_0 and PP_1

$$\text{CProduct} = PP_0 \times PP_1 \tag{51}$$

If the value of cross product CProduct is less than 0, then P lies to the right of the line segment. If Cproduct is greater than 0, then P lies to the left. If CProduct is equal to 0, then P lies on the line segment.

Consider a coordinate system such that one axis lies towards the right-hand side of the screen and one axis lies towards the upper side of the screen. Depending upon the order of vectors in the cross product, the direction of the output vector is either toward the viewer or away from the viewer. Consider a line of reference defined by the points A_1 and A_2 and B_1 and B_2 on either side of the line, as shown in Figure 4.11.

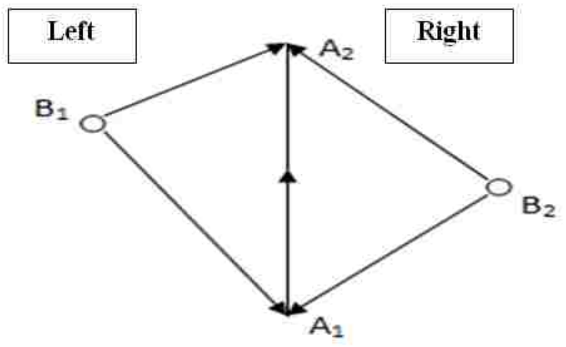


Figure 4.11. Points Lying on Left and Right Side of the Line Depending on the Direction of Vector A1A2

Draw vectors joining points B_1 and B_2 and A_1 and A_2. The next step is to find the cross product between vectors B_1A_1 and B_1A_2 and between vectors B_2A_1 and B_2A_2. Considering the right-hand rule of the cross product, the direction of the product vector is either towards or away from the viewer. If the direction of the product vector is towards the viewer, the point lies on the *left* side of the line, and if the direction of the product vector is away from the viewer, then the point lies on the *right* side of the line. Using this information, it is determined that point B_1 lies on the left side and B_2 lies on right side of line A_1A_2 [13]. Earlier, the vertices of the rectangular cross section were determined. The counterclockwise (CCW) order of vertices P_1 -> P_4 -> P_3 -> P_2 is considered to define the directions of the lines drawn joining these vertices in CCW order. The lines defined in this order are:

```
The Pseudo Code:
If
point Pt lies on left side of line $P_1P_4$
and
point Pt lies on left side of line $P_4P_3$
and
point Pt lies on left side of line $P_3P_2$
and
point Pt lies on left side of line $P_2P_1$
Then, point Pt lies inside rectangular cross section defined by $P_1$ -> $P_4$ -> $P_3$ -> $P_2$.
```

P_1P_4, P_4P_3, P_3P_2 and P_2P_1: In a 3D space, if the given point Pt lies on the left side of each line, looking in the direction of the vector, then that point lies inside the rectangular cross section on the same plane. Figure 4.12 shows two cases of the location of point Pt, one inside and one outside of the rectangular cross section.

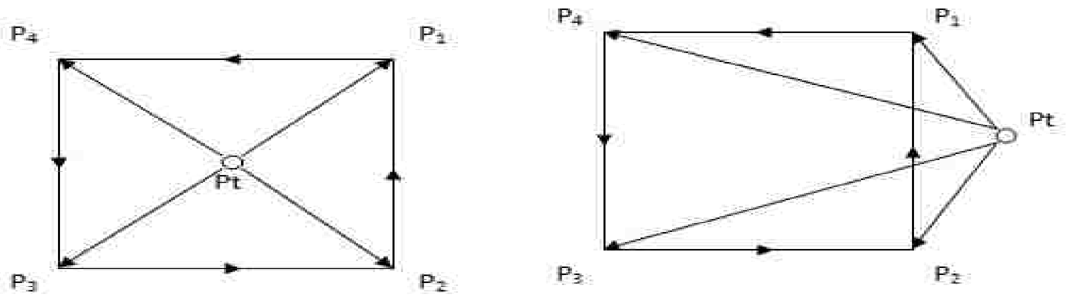


Figure 4.12. Point Lying Inside and Outside a Given Rectangular Cross Section Where Vectors are Considered in Counterclockwise Order

To find the visibility of any point inside the given pyramidal FOV of a camera, the same process was followed for each point for all cameras. All such points visible to cameras in a matrix Cam (mx3), where m is the number of points, were collected. The points inside the FOV of a particular camera are shown in Figure 4.13.

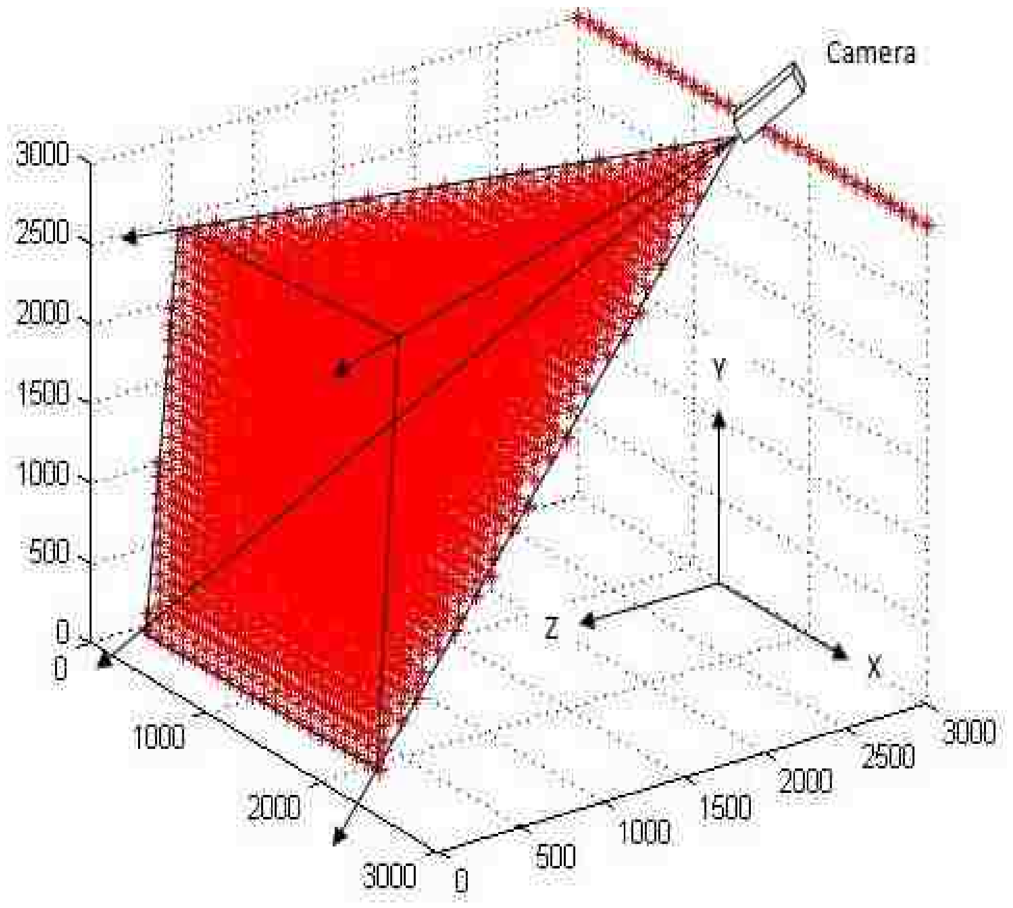


Figure 4.13. Points Visible to a Single Camera Inside the Given Volume

Mathematically, if the points visible to one camera can be found, all the points in the given volume visible to all the individual cameras can be found. The user is capable of defining the random number of cameras to be used. Hence, the system is designed to accommodate a random number of cameras. Figure 4.14 shows the points visible to three cameras.

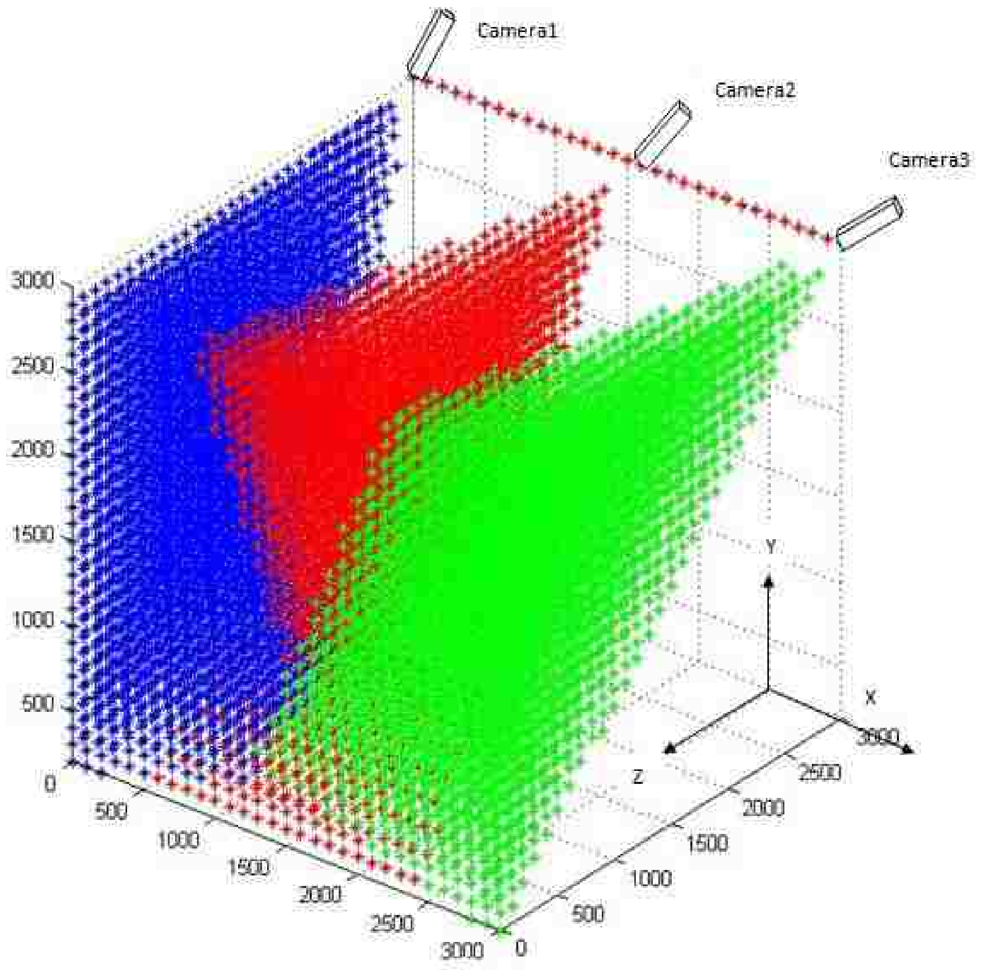


Figure 4.14. Points Visible to Multi-Camera Setup

Figure 4.14 shows that some areas covered by points are visible to any two cameras. The next objective is to collect all the points visible to two cameras at any given moment. Having this matrix Cam computed allows a scoring or evaluation function to be specified taking the matrix as input [1]. The matrix Cam contains all the points covered by all the cameras. Hence, if any point is visible to two cameras, it will be stored twice, and so on. Therefore, the number of appearances of a point in the matrix Cam will reveal the number of cameras to which that point is visible. The most efficient method is to find all the points that appear two or more times in the input matrix Cam, $k \geq 2$, where k is the number of appearances, also called a score. According to the method of calibration implemented, i.e., Svoboda's algorithm [5], the position of a point can be tracked if that point is visible to two cameras.

A setup achieving an increased number of traceable positions for this evaluation method offers better coverage of the specified volume of interest. However, if heavy occlusion problems are encountered during the tracking process, it is desirable to increase the number *k* of cameras required for a position to be classified as: traceable [1].

Pseudo code:

```
for i = 1 : m -> loop to consider each point in the matrix
   k = 1 -> initial counter
       for j = i : m
            if
            i^th point is equal to j^th point of matrix Cam
            k = k + 1
            Add point to matrix CommonTwoCam
            end
       end
   end
```

The matrix CommonTwoCam contains all the points visible to two or more cameras. The points saved in matrix CommonTwoCam provide a clearer picture of the total area visible to the vision system for position tracking. Figure 4.15 shows the points visible to two cameras in the system, represented by black-colored points. The total points visible to all the cameras are represented by different colors.

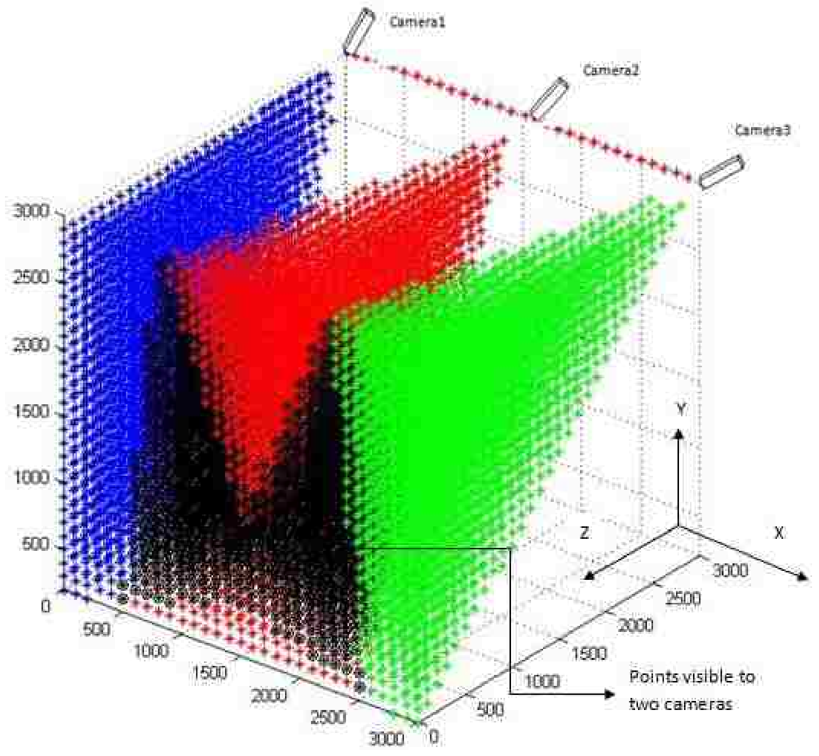


Figure 4.15. Points Visible to Any Two Cameras Simultaneously Shown in Black

In a similar manner, author has designed the function CommonThreeCam.m, which will identify all the points visible to three cameras at any given moment inside the given volume. These points are directed towards the input for the VTK module of the toolbox through a data file (.dat).

4.2.3 MATLAB Module Output. Previous sections have included an explanation of how to find the points visible to two or more cameras at any given time. The output of the MATLAB module consists of the following (which is also the data passed to the VTK module):

1) Type of cameras (parameters of the FOV of the cameras)

2) Number of cameras
3) Position of cameras
4) Orientation of cameras
5) Coordinates of the points visible to two or more cameras

4.3. SYSTEM SETUP VISUALIZATION IN THE VTK MODULE

The sole purpose of the VTK module is to create a visualization of the system's setup along with the area covered by the multi-camera vision system.

4.3.1. VTK Module Inputs. As explained in the toolbox flowchart, the output of the MATLAB module is directed to the VTK module. Hence, the inputs for the VTK module are the same as the outputs of the MATLAB module, which are:

1) Type of cameras (parameters of the FOV of the cameras)
2) Number of cameras
3) Position of cameras
4) Orientation of cameras
5) Coordinates of the points visible to two or more cameras

The coordinates of the points are read from a data file. The cameras are shown by their FOVs, i.e., pyramids and gray boxes at their positions along with their numbers. The total available area to be tracked is indicated by the red lines forming a wireframe, and the volume covered by the multiple tracking system is shown by a solid white color. A sample visualization of the system is shown in Figure 4.16. To reduce ambiguity and clarify the visualization, the camera models in the visualization are given numbers so that the user will be able to identify the camera.

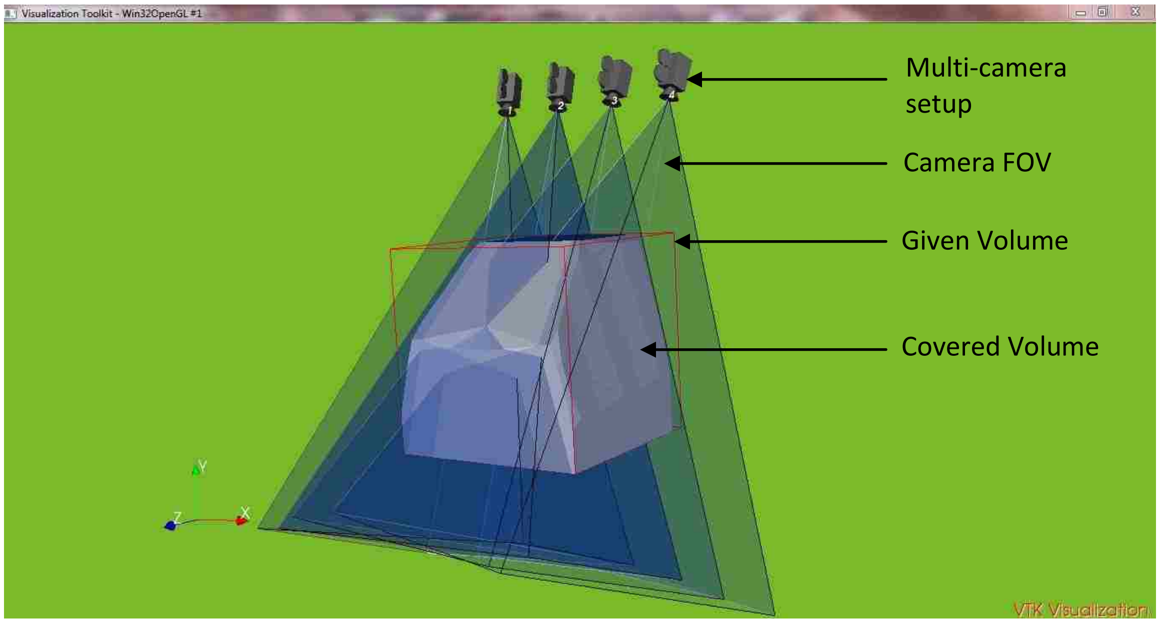


Figure 4.16. Visualization of the Area Covered by the Multi-Camera Vision System Shown in Gray and the Given Volume Enclosed by Red Lines

4.3.2 Reorientation of the Cameras. The ability to reorient cameras is an exclusive feature of the multi-camera vision system setup toolbox. The toolbox is designed for anyone who wants to use a multi-camera vision system. Hence, it is probable that the setup of cameras will be incorrect in terms of the area covered. The toolbox serves its purpose by reducing the time required for practical setup and again changing the orientation. Each change in camera orientation requires another calibration.

The toolbox aims to reduce this time by using pure mathematical and computer graphics methods.

Consider a scenario in which the user wants to change the orientation of the cameras. The user designed multi-camera setup has three cameras initially. The user will need to check the area covered shown by solid white color in the visualization by the vision system with the initially oriented cameras. The VTK visualization is shown by Figure 4.17.

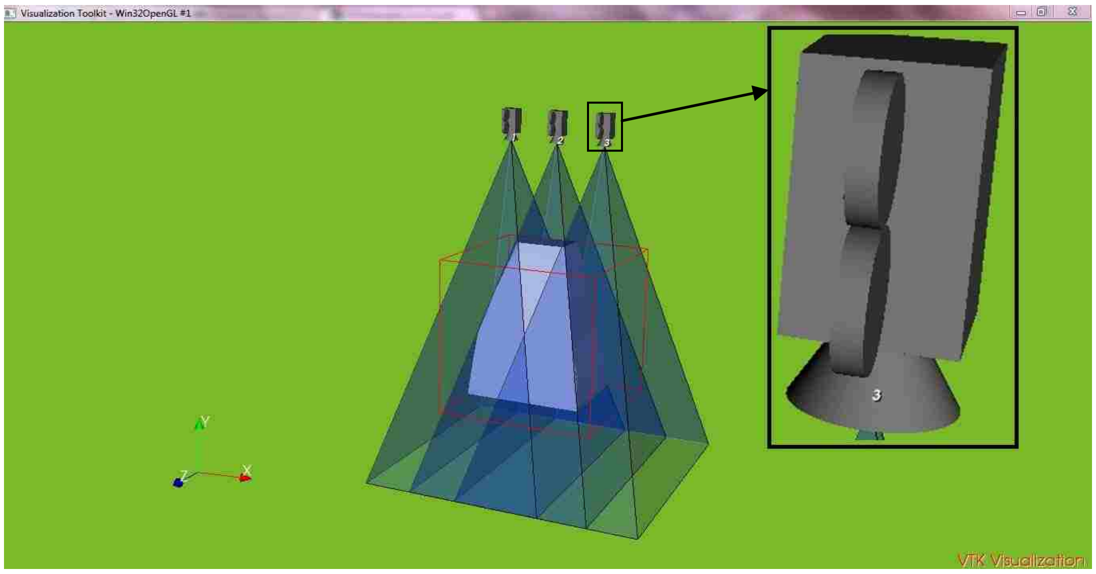


Figure 4.17. Visualization of the Area Covered by the Multi-Camera Vision System consisting of Three Cameras along with their Numbers

Now suppose the volume covered by the multi-camera setup is not enough, the user will have to reorient the cameras. The user also adds another camera to the multi-camera setup design as shown in Figure 4.18. Hence, the user will have to go through the same procedure in the MATLAB module again until the system is visualized in VTK. This process is demonstrated in the flow chart for the data flow of the toolbox in Figure 4.18.

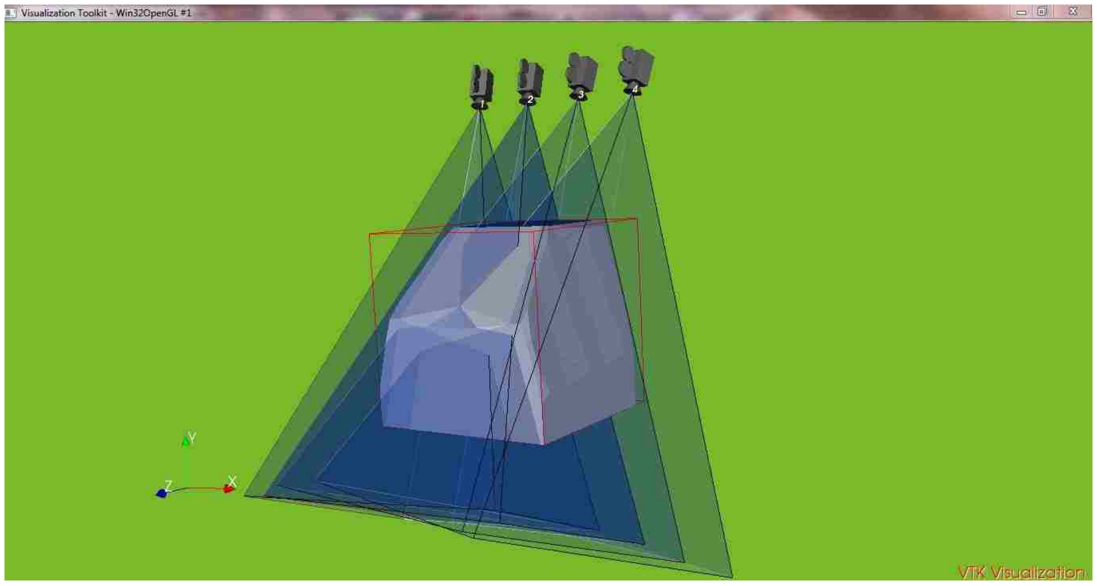


Figure 4.18. Visualization of the Area Covered by the Reoriented Multi-Camera Vision System consisting of Four Cameras along with their Numbers

As shown in the Figure 4.18 that the multi-camera setup consisting of four cameras, the cameras are reoriented and larger volume is covered. Finally after observing the visualization the user will confirm the multi-camera setup design depending on the volume covered.

5. ESTIMATION OF POSITION MEASUREMENT ERROR FOR MULTI-CAMERA VISION SYSTEMS

5.1. IMPLEMENTATION OF MULTIPLE VISION SYSTEMS FOR TRACKING

As discussed previously, author has implemented two algorithms, Svoboda's algorithm for calibrating the cameras in a multi-camera vision system with some common overlap between all the FOVs of the cameras and Horn's algorithm for integrating multi-camera vision systems when no common overlap between the FOVs of all the cameras exists.

5.1.1. Implementation of Svoboda's Algorithm. The cameras used to implement Svoboda's calibration are Wiimotes and firefly cameras. The calibration points are collected using infrared (IR) LEDs mounted on a board. To clarify the multi-camera calibration process, this section explains the method for calibrating multiple Wiimotes together to form a separate vision system. This calibration utilizes the MATLAB toolbox [10]. The final output of this calibration is a matrix P defined in Equation (a). Using the code developed in C#, the (u, v) coordinates for all positions for all cameras for multiple points (approximately 1500) were collected for accurate results [9]. For this setup, the Wiimotes were placed in the CAVE, as shown in Figure 5.1.

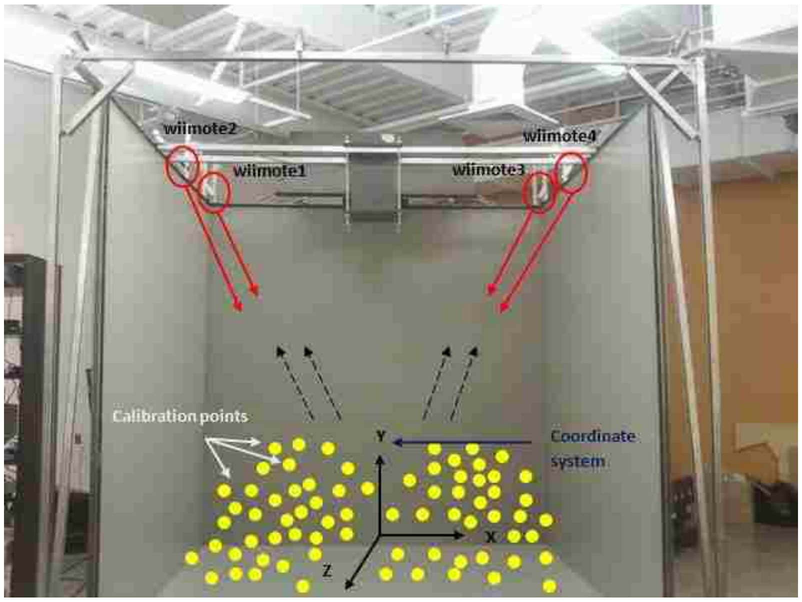


Figure 5.1. The Multi-Camera Vision System Setup for Svoboda's Algorithm Implementation and the Collection of Calibration Points inside the CAVE

The points collected in the entire vision system are shown in Figure 5.1. The calibration generates a transformation matrix, i.e., calibration P matrix (3m X 4), where m is the number of cameras. After generating the P matrix, the vision system is ready for position tracking. The sample P matrix for four cameras is:

-488.73205	1323.41760	-435.21397	1185.53350	
-526.40715	-488.82855	-1245.01160	-93.40065	3 X 4 transformation matrix for camera1
0.64987	0.36752	-0.66528	1.17224	
-992.48446	967.30407	-482.19519	1714.78630	
-383.32724	-808.56491	-1122.23490	21.61061	3 X 4 transformation matrix for camera2
0.40030	0.47282	-0.78498	0.79788	
-1151.31010	-879.13303	-429.73457	1138.95770	
698.38082	-519.75748	-1163.99980	-808.85998	3 X 4 transformation matrix for camera3
-0.52349	0.43104	-0.73496	1.78634	
368.32671	-1428.02520	-282.01197	1237.97110	
777.78668	329.28422	-1215.46320	-907.34692	3 X 4 transformation matrix for camera4
-0.63062	-0.27065	-0.72736	1.91762	

An important factor in setting up and calibrating a vision system is defining the coordinate system for the vision system. The tracking system program developed in C# [9] provides a way of accomplishing this task. For this purpose, a **T**-shaped marker plate on which four IR LEDs are placed forming a T shape is used as shown in Figure 5.2.

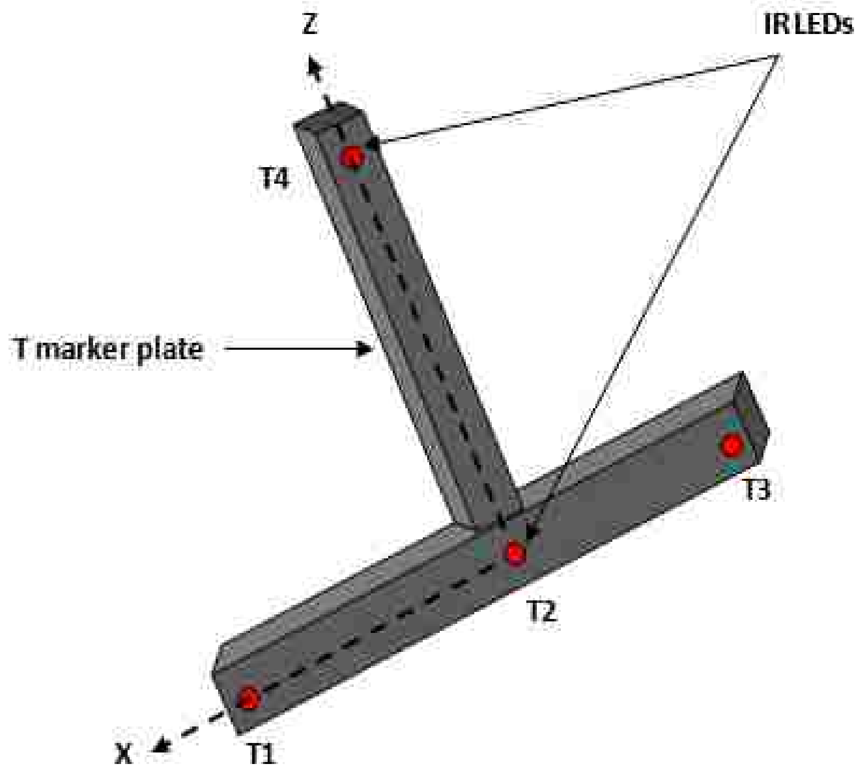


Figure 5.2. T-Shaped Marker Plate with IR LEDs Mounted

The markers define the coordinate axes. The tracking system program finds the coordinates of the four points T1, T2, T3, and T4. The coordinates of points T1-T2-T3 define one axis, and the line joining the points T2-T4 defines the second axis. Using the unit vectors of these axes, the third axis is defined using the cross product. Therefore, the coordinate system is defined with the three axes X, Y and Z, along with the point T2 as the origin of the coordinate system. Defining the coordinate system and the origin provides the rotation matrix R_{vision}, the translation matrix T_{vision} and the scaling factor s_{vision} for all the points visible to the vision system. Hence, define the coordinates of points as:

$$P_{vision} = s_{vision} * R_{vision} * P + T_{vision} \qquad (52)$$

5.1.2. Implementation of Horn's Algorithm. According to the algorithm explained by Horn [25], the coordinates of at least three points are collected in two Cartesian coordinate systems. The rotation matrix, translation matrix and scale factor are calculated using an iterative method of solving the equations, as explained in [29]. Hence, more equations will lead to a more accurate result. In implementing Horn's algorithm for integrating vision systems, as many data points as possible have been collected. These points are the ones visible in both vision systems simultaneously. This will ensure the collection of the coordinates of these points in both of the Cartesian coordinate systems.

For practical implementation, consider two vision systems as shown in Figure 5.3. The *Left* and *Right* vision systems have two different Cartesian coordinate systems. In the following practical implementation, a total of 450 points have been collected. Data points are saved in two matrices, LeftPoints (3m X 3) and RightPoints (3m X 3), where m is the number of points. These two matrices are inputs for the integration function in MATLAB.

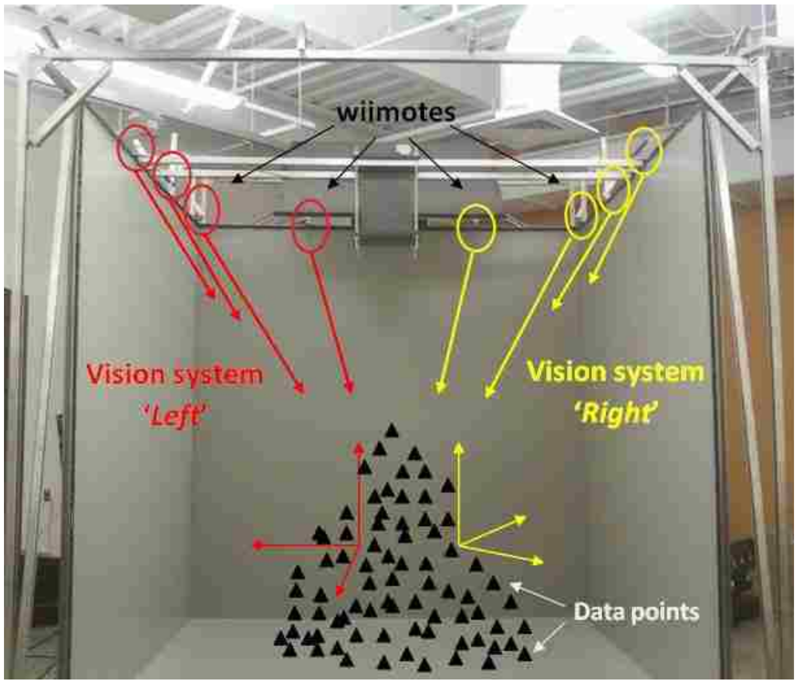


Figure 5.3. System Setup for Integration of Two Multi-Camera Vision Systems Using Horn's Algorithm and Collection of Data Points Visible to both Vision Systems

Two vision systems are calibrated using a combination of Zhang's and Svoboda's algorithms [9]. The calibration method discussed earlier enables the Cartesian coordinate system of each vision system to be defined. Hence, the rotation matrix $\mathbf{Rot_L}$ and $\mathbf{Rot_R}$ and the translation matrix $\mathbf{Tr_L}$ and $\mathbf{Tr_R}$ of the Left and Right vision systems, respectively, are defined.For the current setup:

Rot_L =

0.6353	-0.76923	-0.06799
-0.27627	-0.14418	-0.950201
0.7211	0.622482	-0.304126

Tr_L =

-223.5761353	-315.2733301	769.7844046

Rot_R=

-0.94821	0.317633	-0.0027
-0.05	-0.15766	-0.98623
-0.31368	-0.93501	0.165377

Tr_R =

-258.912	-264.754	669.8854

In the MATLAB module of the toolbox, a function is designed for finding the absolute orientation of a Cartesian coordinate system. Using this function rotation matrix, a translation matrix is generated for determining the absolute orientation between two Cartesian coordinate systems. In this calculation, the *Left* Cartesian coordinate system is the source coordinate system, and the *Right* Cartesian coordinate system is the target coordinate system. Hence, a transformation to transform the coordinates of the *Right* Cartesian coordinate system to the *Left* coordinate system is found. This transformation is given as shown in Figure 5.4:

$$P_{Left} = s_{Horn}.Rot_{Horn}(P_{Right}) + T_{Horn} \quad (53)$$

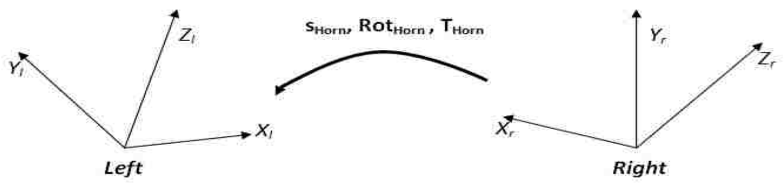


Figure 5.4. Transformation of the Coordinates of the Points from the Right Cartesian Coordinate System to the Left Cartesian Coordinate System

where P_{Left} (3mx3) and P_{Right} (3mx3) are coordinates of the points in the *Left* and *Right* Cartesian coordinate systems, respectively, and m is the number of points. Rot_{Horn} (3x3) is the rotation matrix, T_{Horn}(1x3) is the translation matrix and s_{Horn} is the scale factor.

$s_{Horn} = 1$

$Rot_{Horn} =$

-0.93442	0.204515	-0.29162
0.111229	0.945329	0.306563
0.338373	0.254021	-0.90608

$T_{Horn} =$

39.1057	112.641	62.75604

5.2. ERROR ESTIMATION OF THE TRACKING SYSTEM

The key factor that decides the usability of any position tracking system is its accuracy. Some research has been conducted on the same topic [17][18]. Author has presented the mathematical equations and methods used to estimate the maximum position measurement error of the multi-camera vision system. To clarify the evaluation of the implementations of these algorithms, author has reffered to several papers, research notes and technical reports.

5.2.1. Error Estimation for Svoboda's Algorithm. The errors are calculated for a particular setup of the multi-camera vision system shown in Figure 5.1. The important aspect of error estimation is careful calibration of the vision system. The cameras in the vision system are calibrated exactly according to the procedure described in previous research [9][5]. While implementing the C# tracking code, it is essential for the user to collect the right set of points. For two IR LEDs, the program shows blobs of two different colors. It is critical that, while collecting the data, the colors of these blobs are the same for all the cameras. Only then will the data be accurate. If the colors of the blobs are different for different cameras, then the program might generate random coordinates. In general, the user must diligently address the order of the IR LEDs for all the cameras. Previous research [17][18] does not mention a particular least number of observations for estimating the error of a vision system. Error is calculated for a total of 327 positions. The error in position tracking is calculated as shown in Equations (42) and (43). The observations are plotted in MS Excel.

The average error calculated for this particular vision system equals 3.01mm, and the RMS error equals 2.78mm shown by plot in Figure 5.5.

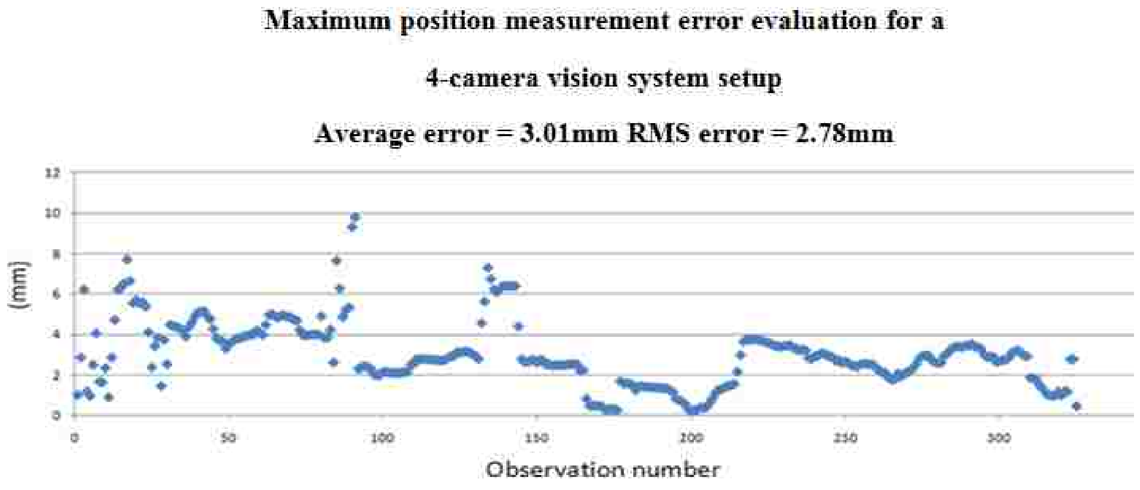


Figure 5.5. Maximum Position Measurement Error Evaluation Plot for Multi-Camera Vision System Calibrated Using Svoboda’s Algorithm

5.2.2. Error Estimation for Horn’s Algorithm. The error estimation methodology used for Horn’s algorithm is similar to that used for Svoboda’s algorithm. The main difference between the two procedures is that Horn’s algorithm deals with two integrated vision systems. Hence, while using two IR LEDs placed at a known distance from each other, care must be taken that only one LED is visible to only one vision system. Again, for estimating the error for the implemention of Horn’s algorithm, the order of LEDs shown by the color of the blobs must be heeded. To achieve accurate measurements, author has used the method described by Freeman [17], Pentenrieder [18], and Vader [9]. In this method, the user measures the absolute distance. In the study described in this thesis, the coordinates of points at 540 positions were collected. Only one IR LED is visible to one vision system during the collection of these coordinates. Hence, at any given position, two sets of coordinates are collected (one for the Left and one for the Right vision system). Using the scale factor s_{Horn}, Rot_{Horn}, T_{Horn}, the coordinates of the points collected for the Right vision system, i.e., the target coordinate system, are transformed to the Left vision system, i.e., the source coordinate system, as shown in Equation (53). Now, all the coordinates are in only one Cartesian coordinate system (the Left Cartesian coordinate system). Using the formulae for finding the absolute distance, Equation (42), and the RMS value, Equation (43), the error in the collected data points is found. This error is the error in the multi-camera vision system’s integration. The results are plotted in MS Excel, as shown in Figure 5.6.

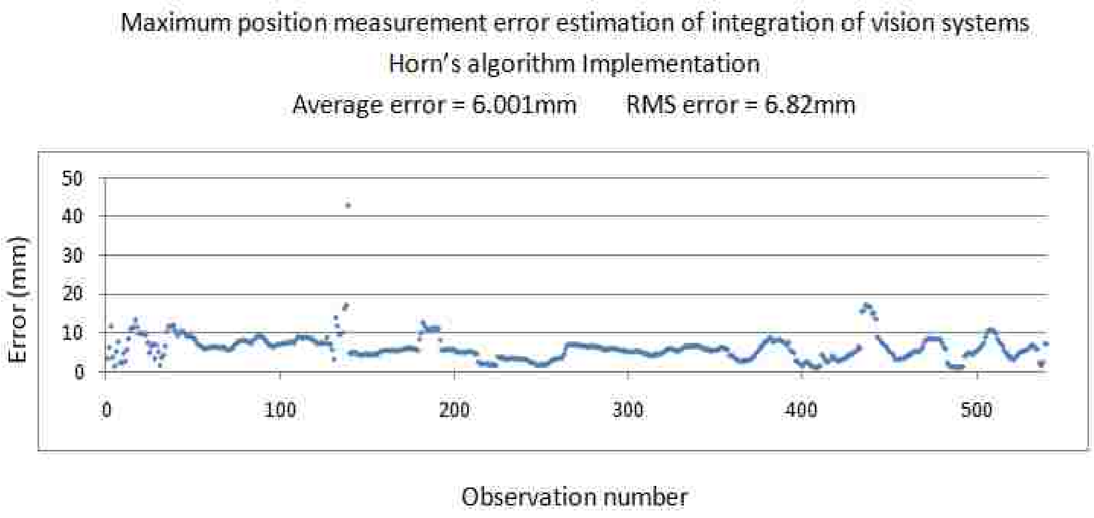


Figure 5.6. Maximum Position Measurement Error Evaluation Plot for Integration of Multi-Camera Vision System Using Horn's Algorithm

5.3. IMPLEMENTATION OF DIJKSTRA'S ALGORITHM

Horn's algorithm is suggested for integrating the vision systems for position tracking in wider areas. This integration also experiences some position-tracking errors between vision systems. Integrating multiple vision systems is necessary for wide-area position tracking. According to the implementation of Horn's algorithm, one of the multi-camera vision systems (the Cartesian coordinate system) is considered the *source*, and the rest are *target* vision systems. Considering that author is integrating multiple vision systems, only one of the vision systems will be the *source* system. This vision system is called the *master system*, and all the target systems are *slave systems*. Now, the task is to transform the coordinates of the points visible to any vision system into the Cartesian coordinate system of the master system. The slave systems may share some common overlapping area. Hence, these vision systems cannot be integrated directly with the master vision system. The scope of the implementation of Horn’s algorithm must be expanded. The slave systems must be integrated by integrating those that share a common overlap. Hence, all the vision systems must be integrated with their adjacent vision systems. This kind of integration will form a grid-like structure of vision systems. The maximum position errors of the integration of these vision systems must be calculated. Figure 5.7 shows the grid of the typical multiple vision system integrations implemented inside the CAVE.

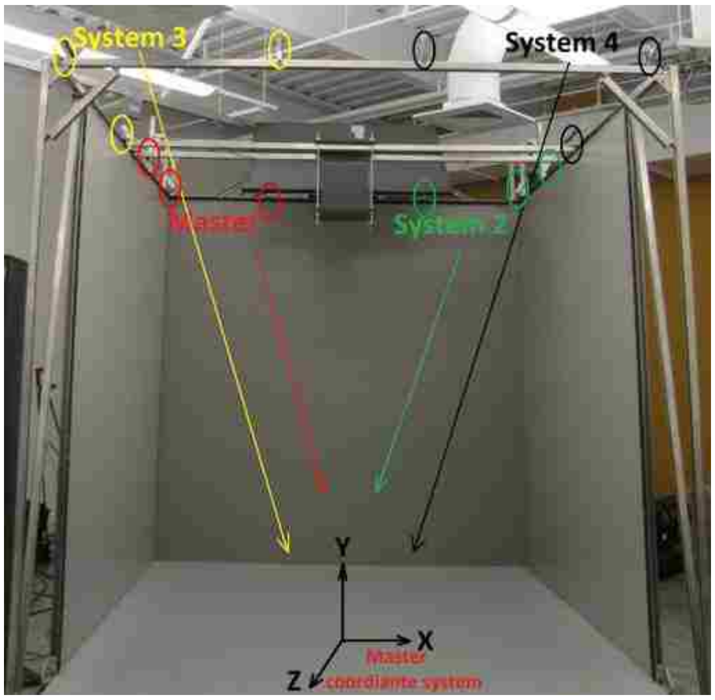


Figure 5.7. Multiple Vision System Setup inside the CAVE and the Adjacent Pairs of Vision Systems Are Integrated using Horn’s Algorithm Implementation

The Dijkstra's algorithm uses the greedy approach to solve the single-source shortest path problem. From the unselected vertices, it repeatedly selects the vertex that is nearest to source s, vertex v, and declares the distance between the two to be the actual shortest distance from s to v. The edges of v, followed by the relevant outgoing edges, are then checked to see if their destination can be reached by v (refer to the diagraph in Figure 5.8).

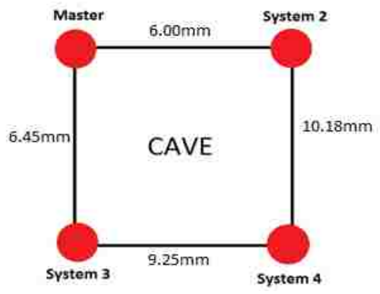


Figure 5.8. Schematic Representation of Multiple Vision System Setup (Top View of CAVE)

Consider the grid along with the maximum position errors as shown in Figure 5.8. The transformation of the coordinates from System2 to Master or from System3 to Master will require only the transformation matrices generated using Horn's algorithm. However, the question of the shortest path arises when considering transforming the coordinates of the points from System4 to Master. Two solutions exist for transforming the coordinates from System4 to Master: Master -> System3 -> System4 Or Master ->System2 -> System4

The correct path, the one that generates the least error, must be chosen from amongst these two possible paths. Based on the maximum position error between all the adjacent vision systems, the algorithm generating the shortest path, described by Dijkstra, is implemented. Dijkstra's algorithm is called the single-source shortest path, as well as the single-source shortest path problem. It computes the length of the shortest path from the source to each of the remaining vertices in the graph. The single-source shortest path problem can be described as follows: Let G = {V, E} be a directed weighted graph with V having a set of vertices. In the special vertex s in V, where s is the source, let EdgeCost(e) be the length of any edge e in E. All the weights in the graph should be non-negative. A directed graph can be defined as an ordered pair: G = (V,E), where V is a set whose elements are called vertices or nodes, and E is a set of ordered pairs of vertices, called directed edges, arcs, or arrows. Directed graphs also are known as digraphs. Dijkstra’s algorithm works by solving the subproblem k, which computes the shortest path from the source to vertices among the k closest vertices to the source. For Dijkstra’s algorithm to work, it should be a directed-weighted graph, and the edges should be non-negative. The actual shortest path cannot be obtained if the edges are negative. At the kth round, there will be a set called Frontier of k vertices that will consist of the vertices closest to the source. The vertices that lie outside Frontier are computed and put into New Frontier. The shortest distance obtained is maintained in sDist[w], which holds the estimate of the distance from s to w. Dijkstra’s algorithm finds the next closest vertex by maintaining the New Frontier vertices in a priority-min queue. The algorithm works by keeping the shortest distance of vertex v from the source in an array sDist. The shortest distance of the source to itself is zero. sDist for all other vertices is set to infinity to indicate that those vertices are not yet processed. After the algorithm finishes processing the vertices, sDist will have the shortest distance of vertex w to s. The two sets, Frontier and New Frontier, are maintained, which facilitates the processing of the algorithm. Frontier has k vertices that are closest to the source and will have already computed the shortest distances to these vertices for the paths restricted up to k vertices. The vertices residing outside of Frontier are put in New Frontier.

```
Pseudo code:
Procedure    Dijkstra (V: set of vertices 1... n {Vertex 1 is the source}
             Adj[1…n] of adjacency lists;
             EdgeCost(u, w): edge – cost functions;)
Var:         sDist[1…n] of path costs from source (vertex 1); {sDist[j] will be
equal to the length of the shortest path to j}
Begin:
Initialize
{Create a virtual set Frontier to store i where sDist[i] is already fully solved}
Create empty Priority Queue New Frontier;
sDist[1]←0; {The distance to the source is zero}
forall vertices w in V – {1} do {no edges have been explored yet}
sDist[w]←∞
end for;
Fill New Frontier with vertices w in V organized by priorities sDist[w];
endInitialize;
```

```
repeat
v←DeleteMin{New Frontier}; {v is the new closest; sDist[v] is already correct}
forall of the neighbors w in Adj[v] do
if sDist[w]>sDist[v] +EdgeCost(v,w) then
sDist[w]←sDist[v] +EdgeCost(v,w)
update w in New Frontier {with new priority sDist[w]}
endif
endfor
until New Frontier is empty
endDijkstra;
```

A practical implementation of Dijkstra’s algorithm in MATLAB is provided in the toolbox; it will allow the user to find the shortest path for the multiple vision systems. The current setup inside the CAVE is:

As Table 5.1 indicates, differences exist between the maximum position measurement errors of pairs of multi-camera vision systems integrated using Horn’s algorithm. Several reasonable possibilities exist for this difference, one being that errors occurring during calibration and integration contribute to larger errors. However, in this study, this reason can be discarded safely because all of the pairs use the same method of calibration and integration.

Table 5.1. Pairs of Multi-Camera Vision Systems Integrated Together Using Horn's Algorithm and Their Maximum Position Measurement Errors

Pair of vision systems	**Average error (mm)**
Master -> System2	6.00
Master-> System3	6.45
System2-> System4	10.18
System3-> System4	9.25

Hence, the shortest path is Master -> System3 -> System4. The maximum position error for the shortest path is 6.45mm + 9.25mm = 15.7mm.

In this way, the user can determine the shortest path between any two vision systems. Dijkstra's algorithm can be used for any number of vision systems. Before implementing this algorithm, the user must possess the data regarding the average maximum position errors between all of the adjacent multi-camera vision systems.

Differences in errors also can be caused by fluctuations that occur while the camera is collecting data. In this study, Wiimotes are used as cameras. The Wiimotes are very sensitive to the intensity of the IR LEDs. It is logical that this intensity decreases when an IR LED is moved away from the camera. Hence,investigated the possible existence of a relationship between the maximum position error and the distance between IR LED markers and the multi-camera vision system. This relationship can be established by plotting the frequency graph of the range of distances vs. the number of data points collected within the given range of distances of the data points from the multi-camera vision system as shown in Figure 5.9. The data is collected for 540 data points i.e. 540 different positions.

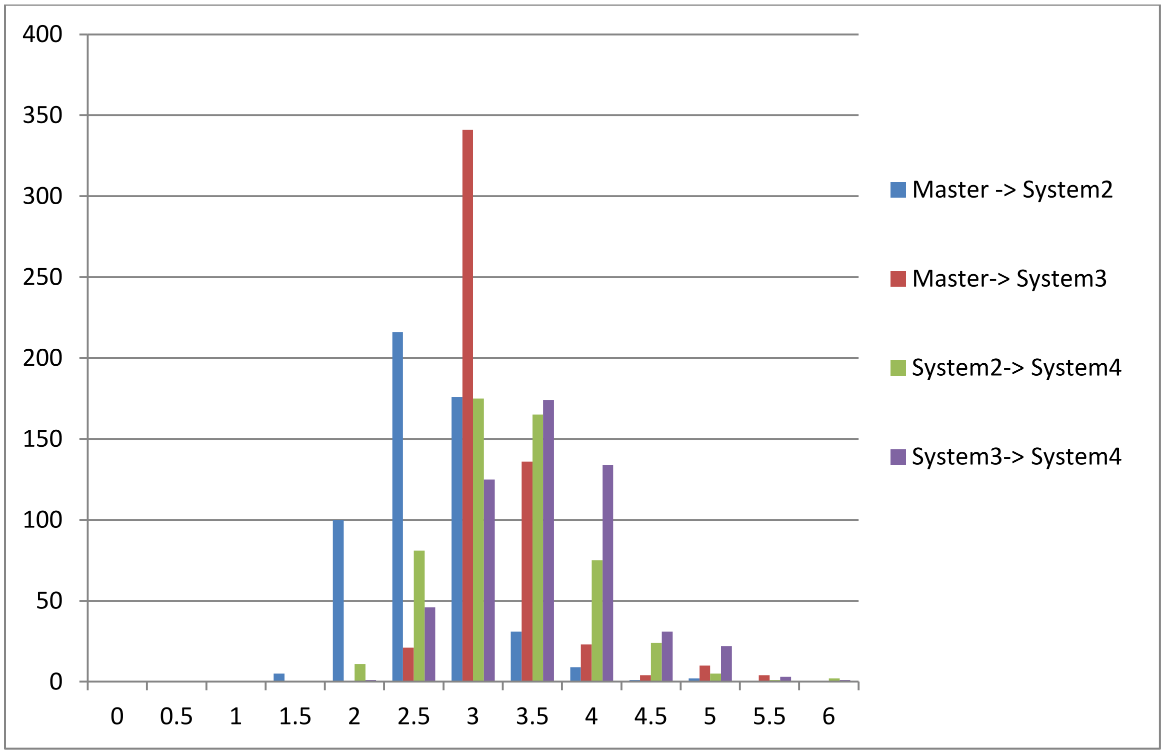


Figure 5.9. The Frequency Plot of the Range of Distances Between the Multi-Camera Vision System and IR LED Marker vs. the Total Number of Data Points Collected Within the Respective Range

The Table 5.2 clearly indicates that for the pair Master->System2, 91.1% of the total points were collected within the range of 2m to 3m from the multi-camera vision system. For the pair Master-> System3, 67% of the total points were collected within the range of 2m to 3m from the multi-camera vision system. However, for the pairs System2-

> System4 and System3-> System4, 81.3% and 85.9% of the total points were collected within the range of 3m to 4.5m from the multi-camera vision systems, respectively. Hence, the statistics support the theory that fluctuations in data collection increase as a marker is moved away from the cameras, causing larger errors in the calibration and integration of the multi-camera vision systems.

Table 5.2. The Range of Distances Between the Multi-Camera Vision System and IR LED Marker vs. the Total Number of Data Points Collected Within the Respective Range

					Range of distances (m)								
System pairs	0	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5	5.5	6
Master -> System2	0	0	0	5	100	216	176	31	9	1	2	0	0
Master-> System3	0	0	0	0	0	21	341	136	23	4	10	4	0
System2-> System4	0	0	0	0	11	81	175	165	75	24	5	1	2
System3-> System4	0	0	0	0	1	46	125	174	134	31	22	3	1

6. TOOLBOX EVALUATION

Author has attempted to evaluate the toolbox based on the results of error estimation obtained by implementing different algorithms for calibrating multi-camera vision systems (Svoboda's algorithm) and integrating multiple vision systems (Horn's algorithm).

6.1. ACCURACY MEASUREMENT

As mentioned previously, Svoboda's algorithm was implemented for calibrating the multi-camera vision systems. The error estimation for all four multi-camera vision systems is plotted, as in Figure 6.1 and shown in Table 6.1. To clarify the estimation and the plot of the data, each vision system has 325 observations. The IR LEDs were placed 165mm from each other. The error is calculated as per Equation (42) and measured in mm. The data points are collected within 1.5 to 3.5 m away from the cameras.

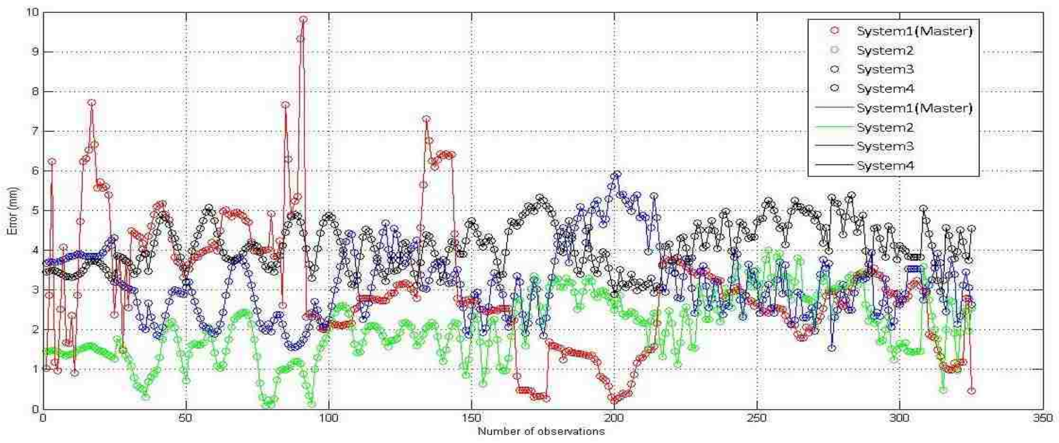


Figure 6.1. Maximum Position Measurement Errors of Four Vision Systems, Each Calibrated Using Svoboda's Algorithm

Table 6.1. Maximum Position Measurement Errors of Individual Multi-Camera Vision Systems and Data Points Collected for Each Observation

Name of the system	**No. of observations**	**Average error (mm)**
Master	325	3.01
System2	325	2.05
System3	325	3.56
System4	325	4.1

The average error of the four vision systems is 3.18mm.

The error estimation for four integrated multi-camera vision systems can be summarized by plotting absolute errors and finding their average. For this error estimation, the same number of observations for each pair of multi-camera vision systems has been collected. There are 540 observations for each pair. The IR LEDs were placed 890mm from each other. The data points are collected within 1.5 to 3.5 m away from cameras as shown in Table 6.2. The average error using Horn's algorithm is 7.97mm.

Table 6.2. Maximum Position Measurement Errors of Pairs of Multi-Camera Vision Systems and Data Points Collected for Each Observation

Pair of systems	**No. of observations**	**Average error (mm)**
Master -> System2	540	6.00
Master-> System3	540	6.45
System2-> System4	540	10.18
System3-> System4	540	9.25

In Figure 6.2, the plots show the maximum position measurement errors of the four pairs of multi-camera vision systems. Sharp red points in the plot can be explained as a result of the instability of the Wiimotes.

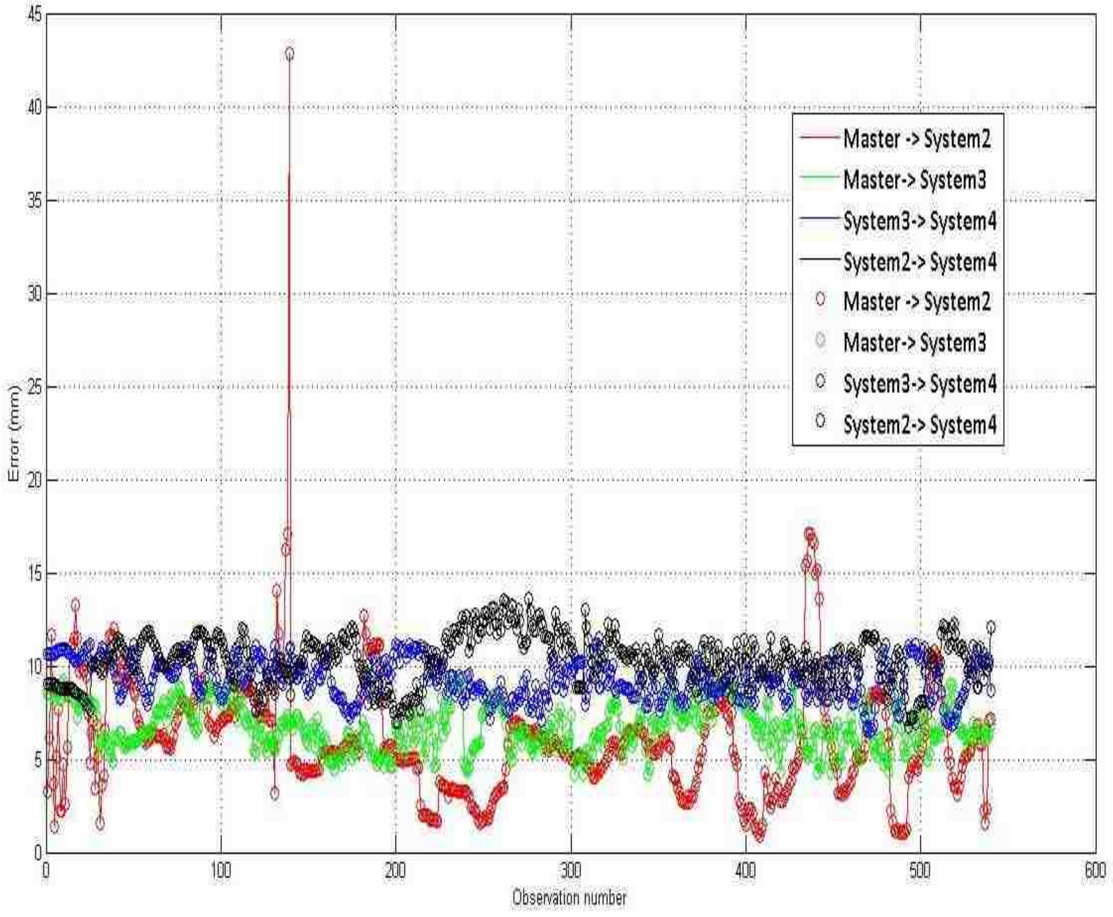


Figure 6.2. Maximum Position Measurement Errors of Four Pairs of Vision Systems, Each Integrated Using Horn's Algorithm

6.2. COMPARISON BETWEEN THE TOOLBOX AND THE ACTUAL IMPLEMENTATION RESULTS

The toolbox is evaluated by comparing the output of the toolbox, i.e. the visualization of the volume covered by the conceptual design of the multi-camera vision system, and the actual volume covered by the practical setup of the multi-camera vision system. For this toolbox evaluation, a stereo camera vision system is placed at a known distance from the origin of the volume to be tracked as shown in Figure 6.3 and 6.4. As mentioned previously, the inputs for the toolbox are:

Type of camera: Wiimote

Minor half angle of FOV: 15^{o} Major half angle of FOV: 20.5^{o}

Maximum visible range (maximum distance from the camera until a marker is visible to the camera): 7000mm

Total number of cameras: 2

Position of the cameras in mm with regard to origin O of the given volume (X, Y, Z): Camera1 (450, 1300, 4900) and Camera2 (550, 1300, 4900)

Volume to be tracked: For this demonstration, a portion of a cubicle is taken as the volume to be tracked.

Total dimensions of this volume: 1000mm X 1750mm X 1800 mm (width X height X depth)

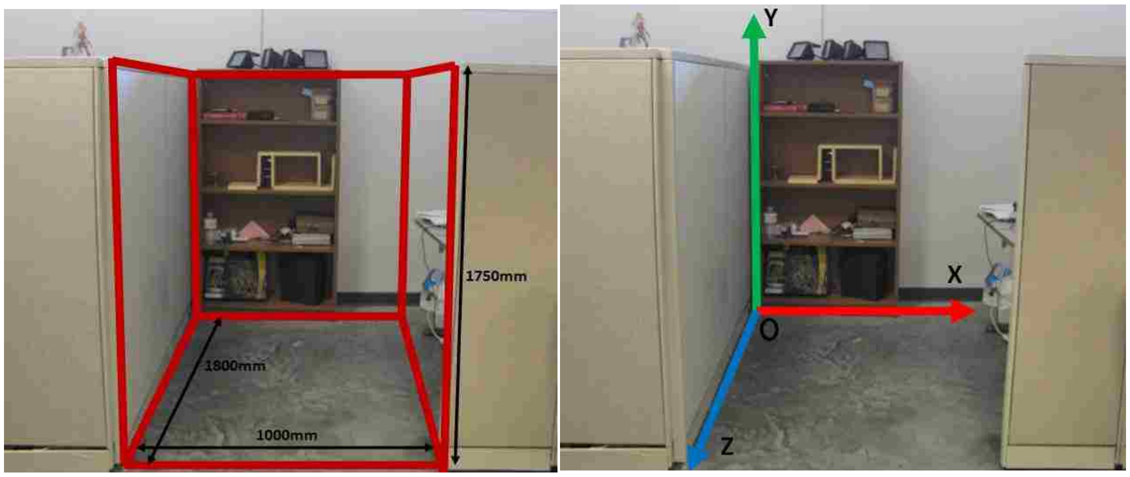


Figure 6.3. The Scenario of the Given Volume, i.e., a Portion of a Cubicle, and the Definition of the Coordinate System and Origin O

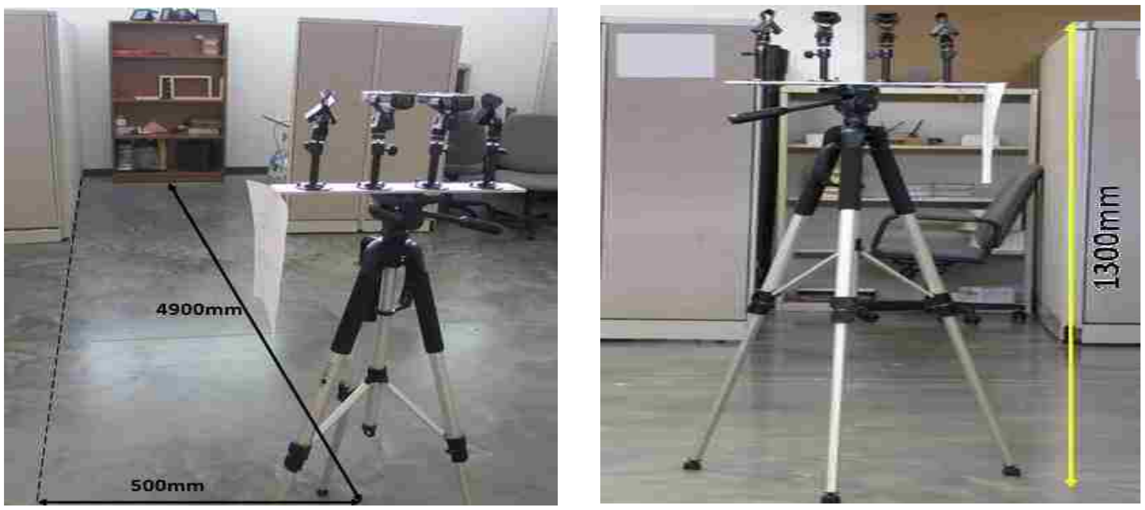
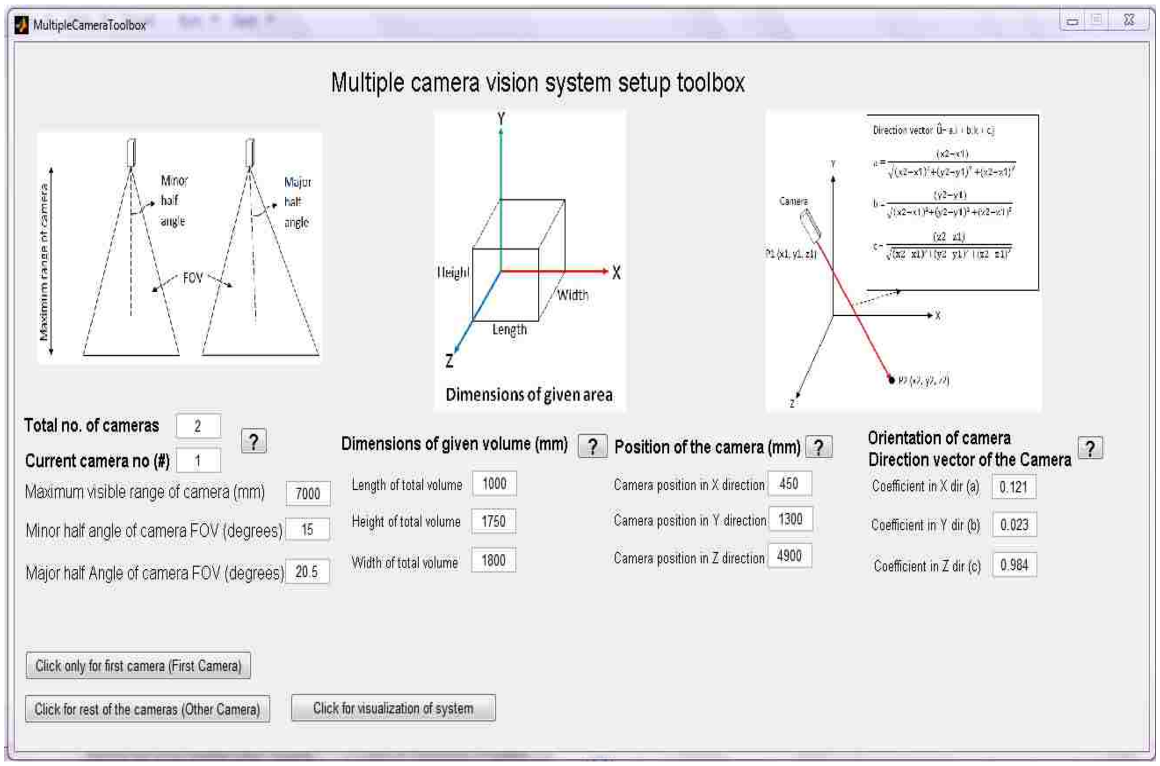


Figure 6.4. Positioning the Stereo Camera System for Tracking

The next step is to input the initial positions and orientations of the cameras. By clicking a few simple and self-explanatory buttons, as shown in Figure 6.5, the user can generate the final output.


MultipleCameraToolbox
Multiple camera vision system setup toolbox
Maximum range of camera
Minor half angle
Major half angle
FOV
Y
X
Z
Height
Width
Length
Dimensions of given area
Camera
P1 (x1, y1, z1)
P2 (x2, y2, z2)
Click only for first camera (First Camera)
Click for rest of the cameras (Other Camera)
Click for visualization of system

Total no. of cameras 2
Current camera no (#) 1
Maximum visible range of camera (mm) 7000
Minor half angle of camera FOV (degrees) 15
Major half Angle of camera FOV (degrees) 20.5

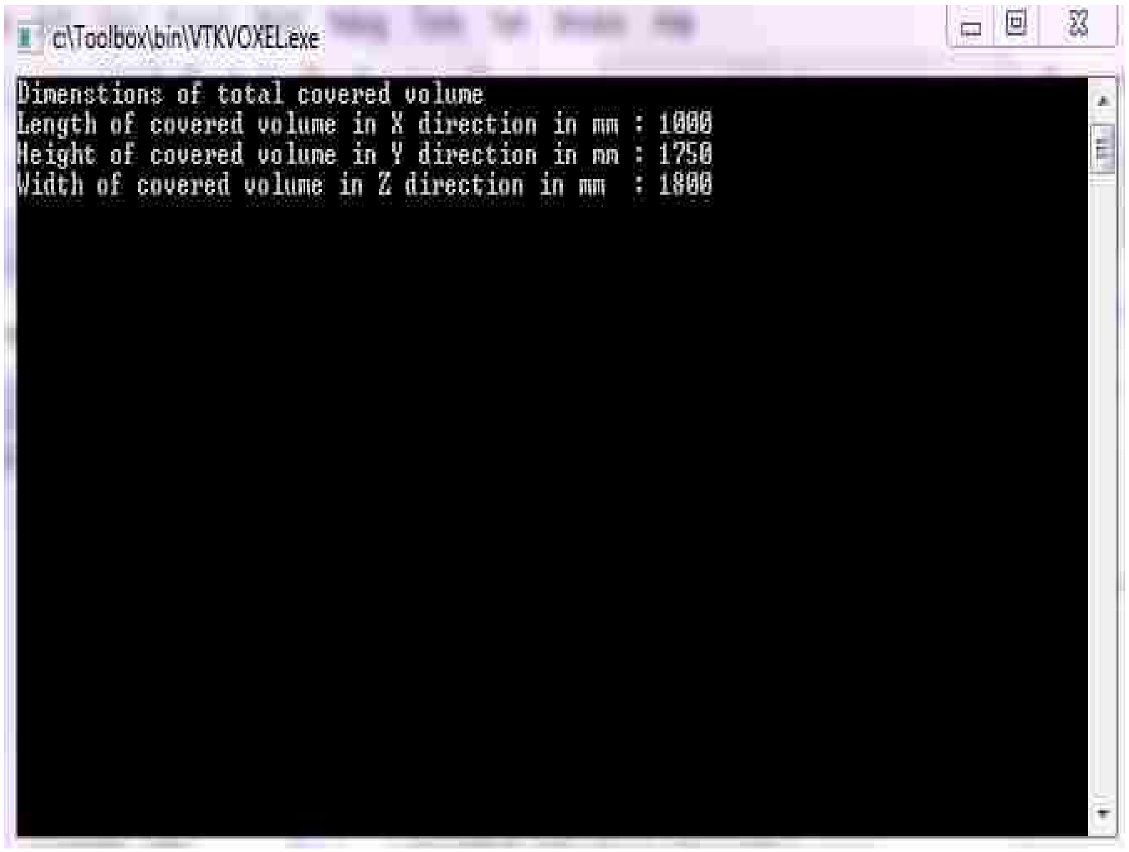
Dimensions of given volume (mm)
Length of total volume 1000
Height of total volume 1750
Width of total volume 1800

Position of the camera (mm)
Camera position in X direction 450
Camera position in Y direction 1300
Camera position in Z direction 4900

Orientation of camera
Direction vector of the Camera
Coefficient in X dir (a) 0.121
Coefficient in Y dir (b) 0.023
Coefficient in Z dir (c) 0.984

Figure 6.5. The Graphical User Interface (GUI) of the Toolbox

The output of the toolbox includes the dimensions of the volume covered and the visualization of the system shown in Figure 6.6.



```
Dimenstions of total covered volume
Length of covered volume in X direction in mm : 1000
Height of covered volume in Y direction in mm : 1750
Width of covered volume in Z direction in mm  : 1800
```

Figure 6.6. Console Output of VTK Indicating Dimensions of the Volume Covered by the Multi-Camera Vision System

According to the output of the toolbox, the system will cover the volume as the dimensions shown in the output are 1000mm X 1750mm X 1800 mm (width X height X depth). The visualization of the VTK module of the toolbox is shown in Figure 6.7.

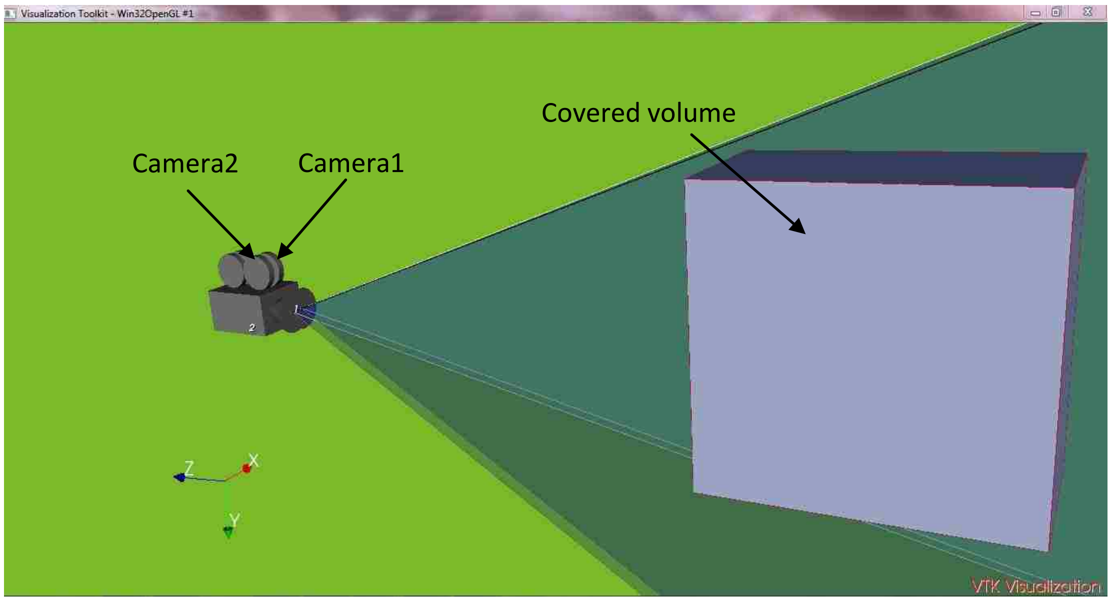


Figure 6.7. VTK Module Output Showing Two Cameras and the Volume Covered by the Two-Camera Vision System

The original volume to be covered is represented by the red wireframe. The white portion shows the volume that will be covered according to the toolbox. Two cameras are shown in the gray color, and their FOVs are shown in the translucent blue color.

The next step is to compare these results with the actual scenario, which is accomplished by checking whether or not an IR LED marker placed in the given volume is visible to the multi-camera vision system as shown in Figure 6.8, 6.9 and 6.10. The visibility of the markers is checked by the wiimote tracking software developed in C#. The visible IR LED marker is represented by a colored blob on the screen. The IR LEDs are placed at the corners of the volume and then at some places within the given volume. The visibility of these IR LEDs will reveal whether the given volume is covered by the multi-camera vision system.

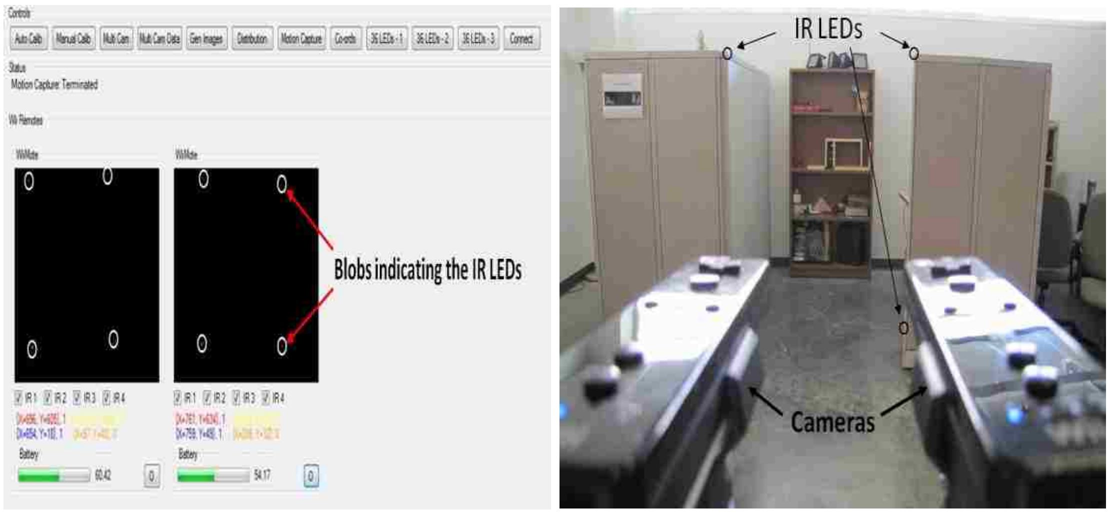


Figure 6.8. Comparison Between the Detection of the IR LEDs and the Actual Setup for LEDs Positioned at the Front Side Corners of the Volume

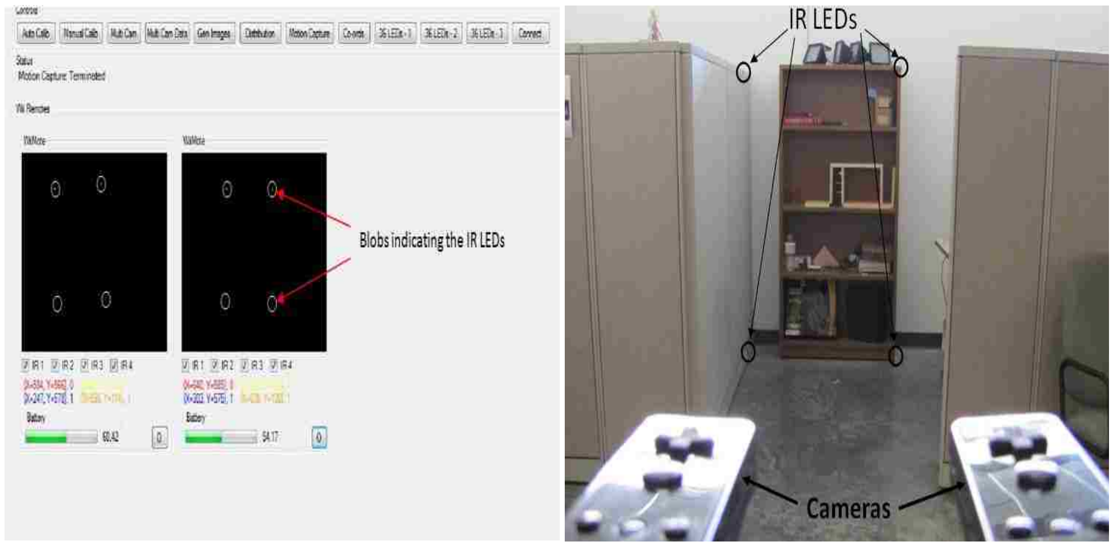


Figure 6.9. Comparison Between the Detection of the IR LEDs and the Actual Setup for LEDs Positioned at the Back Side Corners of the Volume

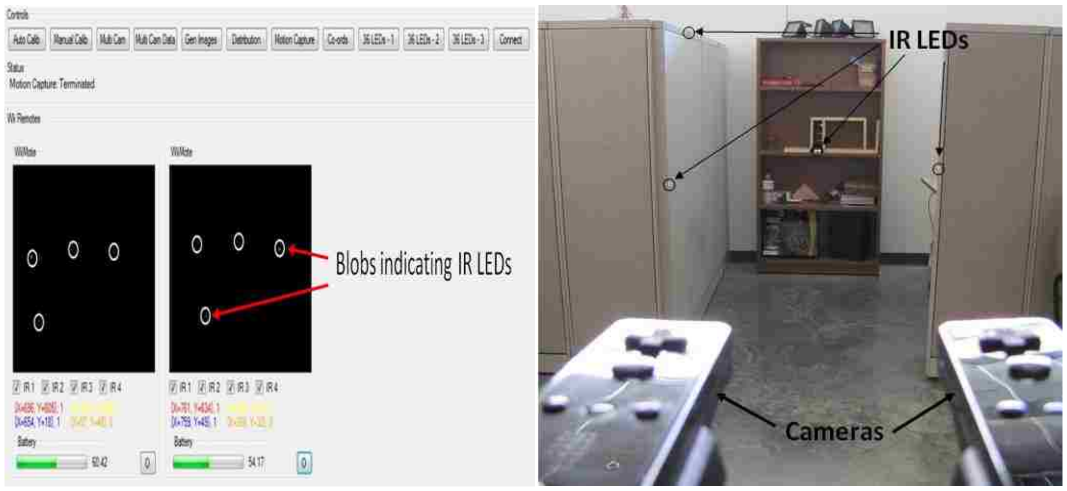


Figure 6.10. Comparison Between the Detection of the IR LEDs and the Actual Setup for LEDs Positioned at Random Places of the Volume

From Figures 6.8, 6.9 and 6.10,   conclude that the given volume is actually covered by the vision system. Hence,   conclude that the toolbox produces high-quality results in a practical implementation.

7. APPLICATIONS

7.1. MULTI-CAMERA VISION SYSTEM SETUP IN THE CAVE

The implementation of the multi-camera vision system utilizing Wiimotes as cameras began with the aim of designing an interactive virtual environment. The multi-camera position tracking system was incorporated through a data file. The basic application of the CAVE is visualization. The starting point for any kind of interactive virtual environment is navigation. The application contains a scenario with simple objects created in OpenGL, such as polygons, cubic blocks, etc.

The navigation process is guided by the camera routine of OpenGL, which has the syntax:

```
gluLookAt(posx, posy, posz, atx, aty, atz, upx, upy, upz);
```

where,

(posx, posy, posz) -> *position of the camera in X Y and Z*

(atx, aty, atz) -> *position where camera is aimed*

(upx, upy, upz) -> *'up' vector*

The position data obtained from the position tracking system is directed to (posx, posy, posz). With this data, navigation is achieved using the multi-camera position tracking system with Wiimotes as cameras explained in Figure 7.1.

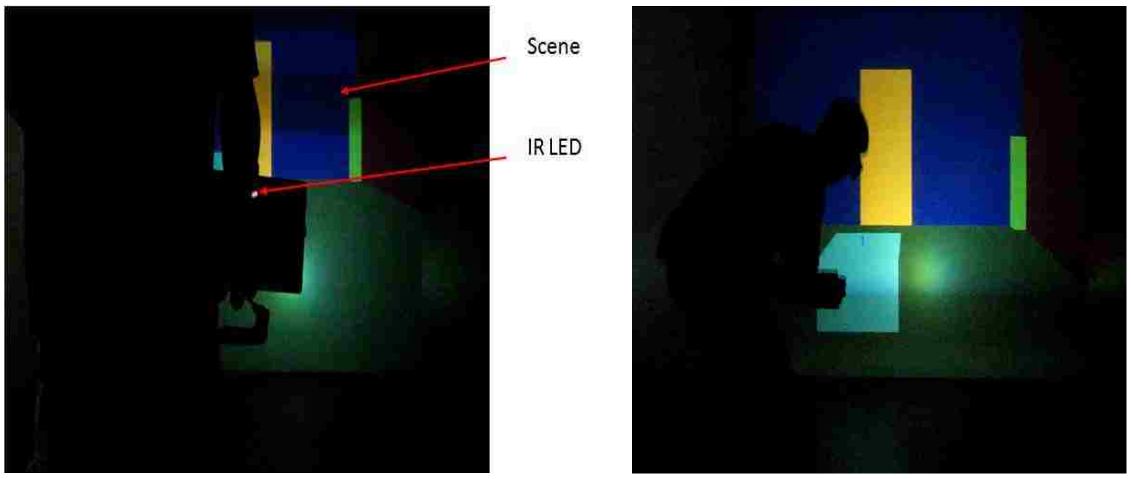
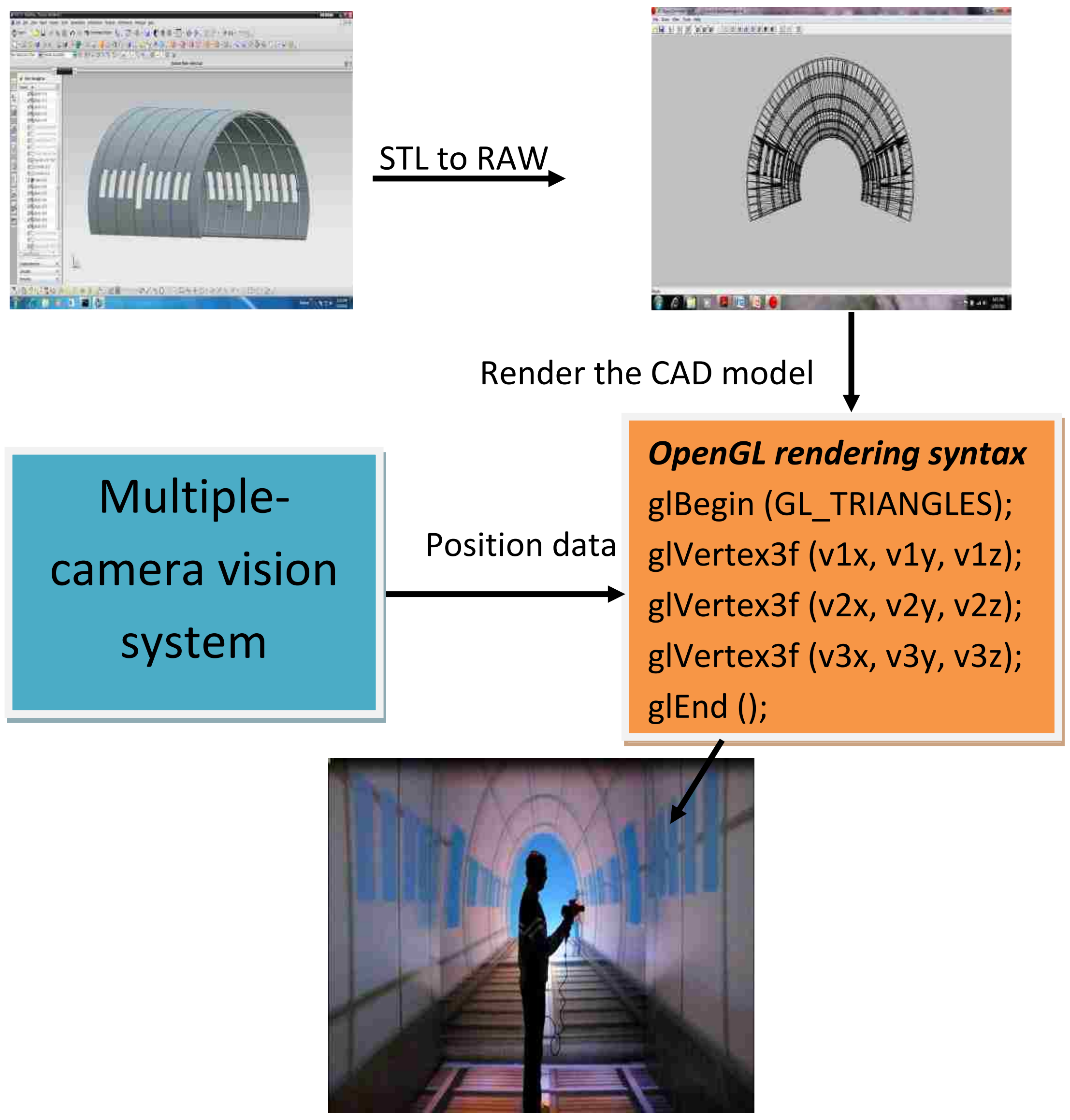


Figure 7.1. Actual Pictures of Navigation Application Inside the CAVE

7.2. VIRTUAL FUSELAGE FASTENING OPERATION

The next step in applying the multi-camera vision systems in the CAVE was to design an application that would simulate a real-life scenario. For this purpose,decided to simulate the fastening operation for a fuselage. The real motivation behind this application was the idea of using the real-time position tracking system for ergonomic analysis of the movement of the worker performing the fastening operation.

The application involves designing the fuselage in the CAD software. The CAD model is rendered using OpenGL and VRJuggler. OpenGL cannot directly render the CAD model, so certain changes must be made to the format in which the model is saved. As mentioned previously, OpenGL can only draw primitive objects, such as lines, rectangular and triangular planes, cubes, spheres, etc. A better option for rendering the CAD model in OpenGL is to use the triangular representation of the CAD model. The vertices of the triangles are obtained by converting the CAD model from the *.STL file format to the *.RAW file format. The RAW file contains only the vertices of the triangles explained in Figure 7.2.



```
glBegin (GL_TRIANGLES);
glVertex3f (v1x, v1y, v1z);
glVertex3f (v2x, v2y, v2z);
glVertex3f (v3x, v3y, v3z);
glEnd ();
```

Figure 7.2. Tree Diagram Showing the Steps in Developing the Virtual Fastening Operation Application Inside the CAVE

The main difference in this application is the movement of an actual object or marker. The marker moves in accordance with the position of the fastening tool. The advantage of this application is that the operator/worker will get the exact location of the fuselage where the fastening operation is being performed. Figure 7.3 shows the rendering of the fuselage and the interaction of the application with the position tracking system. Figure 7.4 shows real images from the virtual fuselage fastening operation inside the CAVE.

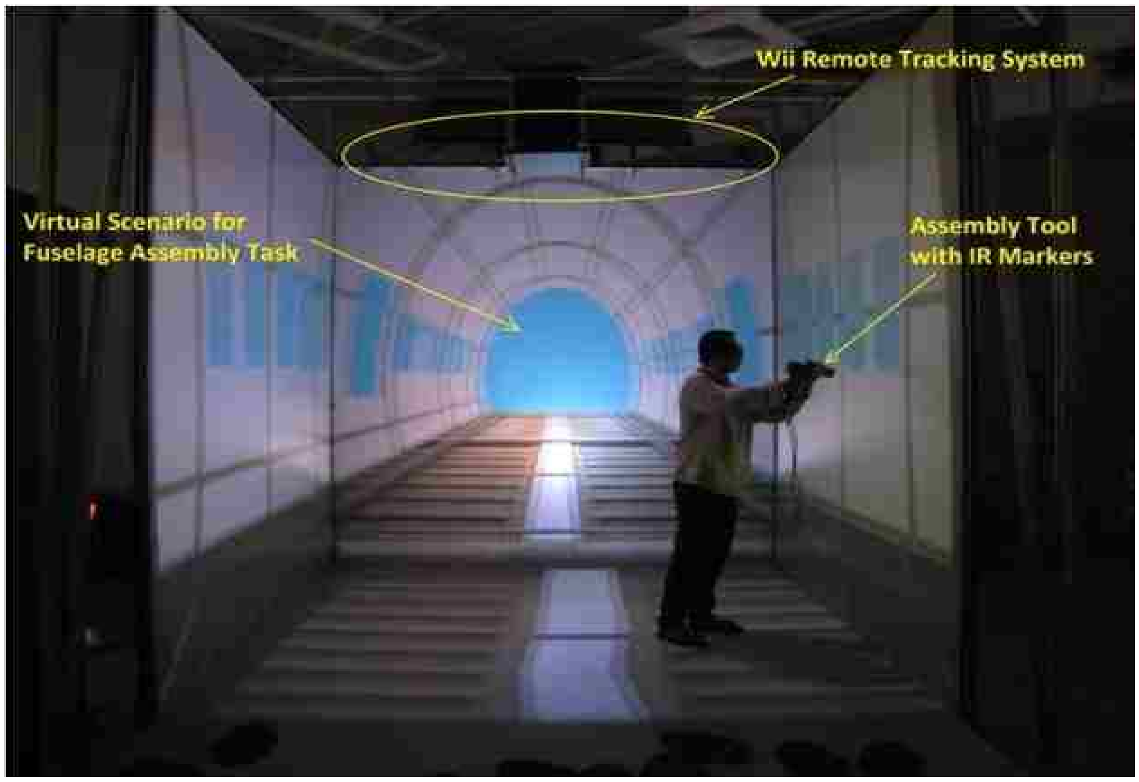

Wii Remote Tracking System
Virtual Scenario for Fuselage Assembly Task
Assembly Tool with IR Markers

Figure 7.3. Picture of the Actual Virtual Fastening Operation of the Fuselage Inside the CAVE 1

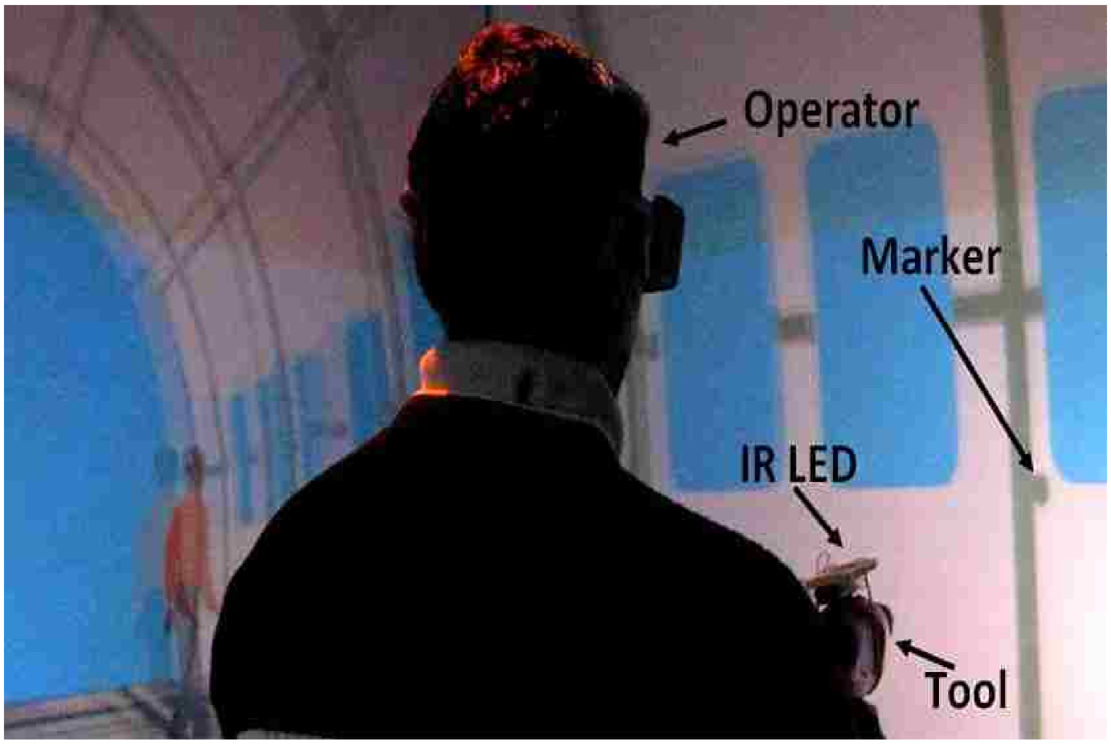


Figure 7.4. Picture of the Actual Virtual Fastening Operation of the Fuselage Inside the CAVE 2

8. CONCLUSIONS

A multi-camera vision system setup toolbox has been developed. The toolbox proves to be very useful for a user to design the setup of a multi-camera vision system. The toolbox comes with a user manual, making it simple to implement. It reduces the time spent placing cameras using the trial and error method. In addition, it serves as a tool to evaluate the design of a multi-camera system setup.

The toolbox was developed using open-source VTK and MATLAB. The basic idea behind this combination is that MATLAB is very efficient and fast with mathematical calculations, while VTK is a versatile toolkit for voxel-based visualization. The developed toolbox provides a GUI to make it easier to use. The toolbox is designed such that the user will be able to generate the visualization with the fewest button clicks. The visualization contains the given volume to be covered, cameras placed at given positions and orientations and the actual volume covered by the multi-camera system. The toolbox allows the user to input any type and number of cameras, which can be placed at any random position and orientation. The user also can input any dimensions of the volume to be covered. The toolbox comes with VTK APIs. With a single click, the toolbox can automatically generate the visualization of the whole vision system along with the volume covered.

The integration of multiple vision systems using Horn's algorithm is also developed and implemented, making it easier to integrate several vision systems for position and motion tracking in a large area. Dijkstra’s algorithm is implemented to estimate the error in integrating any two multi-camera vision systems. The implementation of this algorithm is also developed in MATLAB.

It is demonstrated through practical implementation that the toolbox can design a multi-camera system setup in a CAVE. The toolbox output is used to calculate the volume covered by the multi-camera system. In this implementation, the multiple vision systems are practically integrated using Horn’s algorithm, and the error is calculated using Dijkstra’s algorithm. The setup is employed successfully in the virtual fastening of a simulated aircraft fuselage in a CAVE.

BIBLIOGRAPHY

1) P. A. Cerfontaine, M. Schirski, D. Bundgens, T. Kuhlen, “Automatic Multi-Camera Setup Optimization for Optical Tracking,” IEEE Virtual Reality, 2006.

2) A.W. Fitzgibbon and A. Zisserman, “Automatic 3D Model Acquisition and Generation of New Images from Video Sequences,” in European Signal Processing Conference (EUSIPCO ’98), pp. 1261–1269, Rhodes, Greece, 1998.

3) M. Ribo, “State of the Art Report on Optical Tracking,” Technical Report VRVis 2001-25, TU Wien, 2001.

4) I. Kitahara, H. Saito, S. Akimichi, T. Onno, Y. Ohta, and T. Kanade, “Large–Scale Virtualized Reality,” Computer Vision and Pattern Recognition, Technical Sketches, June 2001.

5) T. Svoboda, D. Martinec, T. Pajdla, “A Convenient Multi-Camera Self-Calibration for Virtual Environments,” Teleoperators and Virtual Environments, 14(4), August 2005.

6) R. Y. Tsai, “An Efficient and Accurate Camera Calibration Technique for 3D-Machine Vision,” Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, pp. 364-374, Miami Beach, Florida, 1986.

7) Z. Zhang, “A Flexible New Technique for Camera Calibration,” IEEE Transactions on Pattern and Machine Intelligence, 22(11), pp. 1330-1334, 2000.

8) J. Heikkila and O. Silven, “A Four-Step Camera Calibration Procedure with Implicit Image Correction,” IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2000.

9) Anup M. Vader, Abhinav Chadda, Wenjuan Zhu, Ming C. Leu, Xiaoqing F. Liu, and Jonathan B. Vance, “An Integrated Calibration Technique for Multi-Camera Vision Systems”, ASME 2010 World Conference on Innovative Virtual Reality 2010-3732; pp. 267-274, 2010.

10) J. Y. Bouquet, “Camera Calibration Toolbox for Matlab,” http://www.vision.caltech.edu/bouguetj/calib_doc/index.html, 2008.

11) P. Bourke, “Minimum Distance between a Point and a Line,” http://paulbourke.net/geometry/pointline/, 1998.

12) “Right-Hand Rule,” http://en.wikipedia.org/wiki/Right-hand_rule,2011.

13) D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, "Computer Graphics: Principles and Practice," 2nd edition.

14) P. Bourke, "Determining if a Point Lies on the Interior of a Polygon," http://paulbourke.net/geometry/insidepoly/, 1997.

15) K. M. Dawson-Howe and D. Vernon, "Simple Pinhole Camera Calibration," International Journal of Imaging Systems and Technology, 1994.

16) O. Faugeras, "Three-Dimensional Computer Vision," MIT Press, 1993.

17) R. M. Freeman, S. J. Julier, A. J. Steed, "A Method for Predicting Marker Tracking Error," Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium.

18) K. Pentenrieder, P. Meier, G. Klinker, "Analysis of Tracking Accuracy for Single-Camera Square-Marker-Based Tracking," in Dritter Workshop Virtuelle und Erweiterte Realitaet der GI-Fachgruppe VR/AR, 2006.

19) M. Ding, A. Terzis, I-J. Wang, D. Lucarelli, "Multi–Modal Calibration of Surveillance Sensor Networks," MILCOM'06 Proceedings of the 2006 IEEE Conference on Military Communications.

20) J. Renno, J. Orwell and G.A. Jones, "Learning Surveillance Tracking Models for the Self-Calibrated Ground Plane," BMVC 2002.

21) R. Bowden and P. KaewTraKulPong, "Towards Automated Wide Area Visual Surveillance: Tracking Objects Between Spatially Separated, Uncalibrated Views," IEE Proc.- Vis. Image Signal Process., Vol. 152, No. 2, April 2005.

22) L. M. Fuentes and S. A. Velastin, "People Tracking in Surveillance Applications," Proceedings 2nd IEEE Int. Workshop on PETS, Kauai, Hawaii, USA, December 9 2001.

23) "Dijkstra's Algorithm," National Institute of Standards and Technology (NIST), http://xlinux.nist.gov/dads/HTML/dijkstraalgo.html.

24) Dr. –Ing . Konrad Zürl, "The Achilles´ Heels of Optical Tracking Systems: Occlusions and Optical Interferences," A.R.T. GmbH.

25) B. K. P. Horn, "Closed Form Solution of Absolute Orientation Using Unit Quaternions," Journal of the Optical Society of America A, Vol. 4, p. 629, April 1987.

26) D. Walden, "The Bellman-Ford Algorithm and Distributed Bellman-Ford," http://www.walden-family.com/public/bf-history.pdf, 2011.

27) "Guide to Vicon Motion Capturing System," http://hci.rwth-aachen.de/guide_vicon, 2011.

28) L. Downs, "Using Quaternions to Represent Rotation," http://www.genesis3d.com/~kdtop/Quaternions-UsingToRepresentRotation.htm.

29) C. T. Kelley, "Iterative Methods for Linear and Nonlinear Equations," Society for Industrial and Applied Mathematics, 1995.

30) T. Mazuryk and M. Gervautz, "Virtual Reality History, Applications, Technology and Future," Institute of Computer Graphics, Vienna University of Technology, 1996.

31) S. Dixon, "A History of Virtual Reality in Performance," International Journal of Performance Arts and Digital Media, Volume 2 Number 1, 2006.

32) D.W.F. van Krevelen and R. Poelman, "A Survey of Augmented Reality Technologies, Applications and Limitations," The International Journal of Virtual Reality, 9(2):1-20, 2010.

33) C. Cruz-Neira, A. Bierbaum, P. Hartling, C. Just, K. Meinert, "VR Juggler - An Open Source Platform for Virtual Reality Applications," 40th AIAA Aerospace Sciences Meeting and Exhibit 2002.

34) J. Ihr_en and K. J. Frisch, "The Fully Immersive CAVE," Behaviour, 1997.

35) VR Juggler – Programmer's guide, http://www.fh-kl.de/~brill/vr/Assets/doc/vrjuggler/2.2/programmer.guide/programmer.guide/index.html, Iowa State University, 2001–2007.

36) OpenGL - The Industry Standard for High Performance Graphics, www.opengl.org, 2011.

37) W. Schroeder, K. Martin, B. Lorensen, "Visualization Toolkit: An Object-Oriented Approach to 3D Graphics."

VITA

The author, Rohit Vijay Bapat, was born Pune city in India. He received his degree of Bachelor of Engineering in Mechanical Engineering from University of Pune, India in August 2008. He has worked on several university level short term projects based on Mechanical Designing. He has studied various computer languages and CAD softwares since high school. The undergraduate seminar presentation of the author was on F-1 aerodynamics and its importance in designing an F-1 car. His undergraduate final year project was Solar radiation concentrator without tracking. The project was funded by University of Pune and later selected for Nation Non-conventional Energy Sources Symposium (India)

The author worked with India’s largest software solution providing company, TATA Consultancy Services Ltd, as Associate Engineer August 2008 to July 2009.

In August 2009, he joined the Master of Science program in Mechanical Engineering at Missouri University of Science and Technology, Rolla, MO. His research work with Dr. Ming C. Leu at Missouri S&T is mainly focused on developing software toolbox for designing multi-camera vision systems and large area multi-camera motion tracking systems for. The author received his Master of Science degree in Mechanical Engineering from Missouri S&T in May 2012.

Currently, the author is working as Software/Application Engineer with Mindware Engineering, a subsidiary of ESI North America Inc, Farmington Hills, MI.