CENTER FOR CONSTRUCTION ENGINEERING AND MANAGEMENT

CAMFPLAN

A Real-time Markerless Camera Pose Estimation System for Augmented Reality

By

Srinath Sridhar[†] and Vineet R. Kamat[‡]

UMCEE Report No. 2011-01
[†] Electrical Engineering and Computer Science
[‡] Civil and Environmental Engineering
UNIVERSITY OF MICHIGAN
Ann Arbor, Michigan
April 2011

Acknowledgements

I wish to thank Prof. Vineet Kamat for the support and guidance he provided during the course of this research study. In the numerous meetings that we had together he helped me connect my core interests in computer vision and augmented reality to practical problems in construction engineering.

I would also like to thank the members of the Laboratory for Interactive Visualization in Engineering (LIVE), Manu Akula, Suyang Dong, Chen Feng and Sanat Talmaki for their valuable insight on my work. Their interesting perpectives on many technical problems that I faced were really useful in completing this research study.

Lastly, I would like to thank my parents for the unwavering support that they have provided from halfway across the world throughout this study.

Srinath Sridhar

May 25, 2011

Abstract

Augmented Reality (AR) is a technology that allows an image or an image sequence of a real scene to be augmented with virtual objects or scenes created in the computer. AR has numerous applications in construction engineering, particularly in the monitoring of large scale construction projects. The biggest hurdle in implementing an AR system that works in all situations is registration. Registration refers to the process of aligning the real scene coordinate system to the virtual object coordinate system of which camera position and orientation (pose) is a key component required. In this report, CAMFPLAN, a system that uses planar surfaces in the scene to find the pose of the camera is presented. This system uses computer vision techniques to detect and use naturally occuring planes in the scene in order to find camera pose. The advantages of this system are that it functions in real time and is designed to be used as an addon to other AR systems that require reliable camera pose estimation. Results of the system on standard datasets and code snippets of key components of the system are included in this report.

Contents

List of Figures

List of Tables

Nomenclature

API	Application Programming Interface
AR	Augmented Reality
AV	Augmented Virtuality
CAMFPLAN	CAMera From PLANes
DLT	Direct Linear Transform
FPS	Frames per second
GPS	Global Positioning System
IMU	Inertial Measurement Unit
LIVE	Laboratory for Interactive Visualization in Engineering
LMedS	Lease Median of Squares
MRF	Markov Random Field
RANSAC	RANdom SAmpling Consensus
SFM	Structure from Motion
SIFT	Scale Invariant Feature Transform

SURF	Speeded Up Robust Features
VR	Virtual Reality

Chapter 1

Introduction

1.1 Motivation and Objectives

The application of emerging computing technologies in construction management has been steadily increasing over the past decade. Some of these technologies include human computer interfaces like virtual reality (VR), interactive visual simulation and augmented reality (AR). Numerous research efforts have gone into using these emerging technologies in construction engineering (Behzadan and Kamat 2011; Fei Dai et al. 2011). AR, which is closely related to virtual reality, has been gathering a lot of interest as a potential aid in construction engineering and management operations.

In simple terms, AR is defined as the augmentation of a real scene with elements from a virtual scene. A more rigorous definition of AR can be found in section 3.1 which follows the taxonomy of Milgram and Kishino (1994). A scenario that illustrates the use of AR in construction engineering could be as follows. Consider a site for a large scale construction engineering project. The person managing the project would like to monitor the progress made both in terms of data received off site as well as visual inspection on site. In today's scenario these tasks are independent of each other and must be done separately. AR has the potential to change this scenario by combining the two tasks together by augmenting or overlaying the off site data with an image or video of

the construction site in real time. The technology exists today to make this process completely automatic, real time and visually appealing. However, there are several hurdles to making such a system viable. One of the biggest of these hurdles is the problem of ***registration*** in AR.

Any AR scene has to satisfy certain quantitative and qualitative requirements in order to look real. Quantitative requirements refer to the geometric positions of objects augmented to the real scene. Qualitative requirements refer to the visual appeal and realism that an augmented scene exhibits. Registration is part of the former requirement but also affects the visual appeal of the augmented scene. Registration is the process of aligning two 3D coordinate systems with each other. In the case of AR these coordinate systems are those of the real scene and the virtual scene. Registration involves knowing the position and orientation (pose) of the camera used to acquire pictures of the real scene as well as the definition of the virtual object coordinate system. The latter is dependent on the graphics API used and is well documented. It is the problem of camera pose estimation that is key to registration. The registration problem is probably the most researched problem in AR but is also far from being solved (Simon and Berger 2002). There are numerous solutions that exist for solving this problem, each suited to a particular scenario. For instance there are systems that perform marker based registration (Kato and Billinghurst 1999). Others use a model based registration approach that assumes that a predefined model exists for the scene.

The objective of this study is to explore ways of solving the registration problem in AR and to demonstrate a solution by performing experiments on standard datasets. Particular emphasis is placed on methods that are markerless and use certain distinct features in the image such as planes for registration. Also within the scope of this study is to explore possible applications of this system to practical problems in construction engineering and management.

In this report, CAMFPLAN (CAMera From PLANes)[1], a system that performs markerless registration based on planar structures in the scene is presented. CAMFPLAN is based on computer vision techniques that detect planar surfaces for estimating the pose of the camera. Since this sys-

[1]Pronunciation guide: The f in CAM*F*PLAN is silent.

tem is markerless it has a much broader utility in AR systems. Results of CAMFPLAN on standard datasets are shown and potential applications of CAMFPLAN to AR in construction engineering are discussed. The software implementation of this system, also called CAMFPLAN, is distributed along with this report. CAMFPLAN is a general purpose camera pose estimation system that is designed to be used as an *addon* or a *plugin* with other visual simulation and AR tools like VITASCOPE (Kamat 2003) and ARVISCOPE (Behzadan 2008). CAMFPLAN takes a video stream as the input and gives the pose of the camera in real time as its output.

Chapter 2

Literature Review

2.1 Augmented Reality

There are numerous definitions that exist in the literature for AR some of which consider it to be a device oriented system while others treat it as a component of *mixed reality* (MR). Milgram and Kishino (1994) provide a taxonomy of mixed reality systems that include VR and AR apart from augmented virtuality (AV). Probably the most widely accepted definition of AR is the one given by Azuma (1997) who defines AR to be a system that combines a real scene with a virtual scene in real time, is interactive and is registered in 3D. This definition of AR is very convenient since it includes registration as a key step and is therefore the definition adopted for this study.

2.2 Camera Pose Estimation

As described earlier, camera pose estimation is the problem of finding the position and orientation of the camera used for capturing the real scene. This is a very important process in registration and there exist various techniques for finding camera pose. These techniques can broadly be classified into two types *viz.* marker based and markerless techniques. Marker based techniques employ

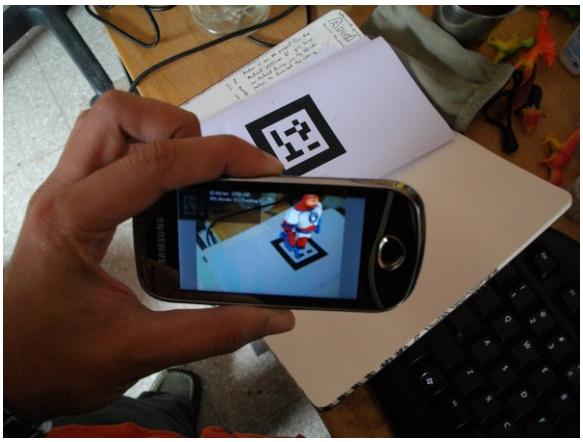


Figure 2.1: *Marker based AR on a mobile*

distinctive markers like the one shown in figure 2.1 to perform camera pose estimation. Markerless systems on the other hand do not need the scene to contain a marker *i.e.* the scene need not be prepared beforehand.

2.2.1 Marker Based Camera Pose Estimation

Kato and Billinghurst (1999) propose a marker based system for augmented teleconferencing purposes. Automatic template matching techniques are used to find the distinctive patterns and register the real and virtual scenes together. Similar methods have been adopted in almost all marker based implementations. One of the most popular applications of this and related techniques in recent times has been in the mobile applications market. Numerous applications as well as toolkits have been developed that can perform marker based tracking on mobile phones since this is a computationally non-intensive process (Qualcomm Inc. 2011).

2.2.2 Markerless Camera Pose Estimation

A more interesting problem in AR is to solve registration without using markers. It is conceivable to use positioning systems such as the GPS for direct markerless pose estimation (Behzadan and Kamat 2007). However, only computer vision based solutions are considered in this report. In order

to solve the registration problem without using markers it is essential to exploit multiple frames from an image sequence to estimate camera pose. The techniques of multiple view geometry (Hartley and Zisserman 2003) in computer vision can be used for this purpose. Fitzgibbon and Zisserman (2003) propose a method in which naturally occurring features in an image sequence are used to produce a reconstruction of a real scene in a process known as structure from motion (SFM). SFM gives the 3D structure of the scene as well as pose of the camera. This process, although highly accurate, is a computationally expensive one and cannot be done in real time. Also known as *match moving*, it is used extensively in the movie industry for adding special effects to the scene (Vicon Motion Systems 2011).

The particular focus of this study is on markerless camera pose estimation systems that use planar structures occurring naturally in a scene. Simon et al. (2000) provide an introduction to this approach that uses concepts of multiple view geometry. Surprisingly, there has been little work on plane based tracking techniques since 2003 which can be attributed to the developers of mobile AR applications who have adopted marker based techniques. It is considerably easier to implement marker based systems on mobile devices given their low computational power.

Chapter 3

System Description

The CAMFPLAN system is composed of several components that perform specific operations such as camera calibration, camera initialization, camera pose estimation and augmentation. In the following sections each component of the system is explained in detail. In order to put things into context however the registration problem in AR is explained in detail first.

3.1 Augmented Reality

Augmented Reality (AR) is a term that refers to a system that adds features from a virtual environment to a real world environment, is interactive, functions in real-time and is ***registered*** in 3D (Azuma 1997). AR is part of a spectrum of reality-virtuality experiences that include other things like *virtual reality (VR)* and *augmented virtuality (AV)*. Figure 3.1 shows this spectrum of reality-virtuality experiences as defined in the taxonomy of Milgram and Kishino (Milgram and Kishino 1994). The ends of the spectrum denote the real and virtual environments while the space between them is occupied by AR and AV which are collectively called *mixed reality*.

Figure 3.1 also shows the various user interfaces associated with each part of Milgram's Continuum. The focus of the study presented in this report is on AR. AR has been defined differently by numerous authors (Caudell and Mizell 1992) but the definition of Azuma (Azuma 1997) is adopted

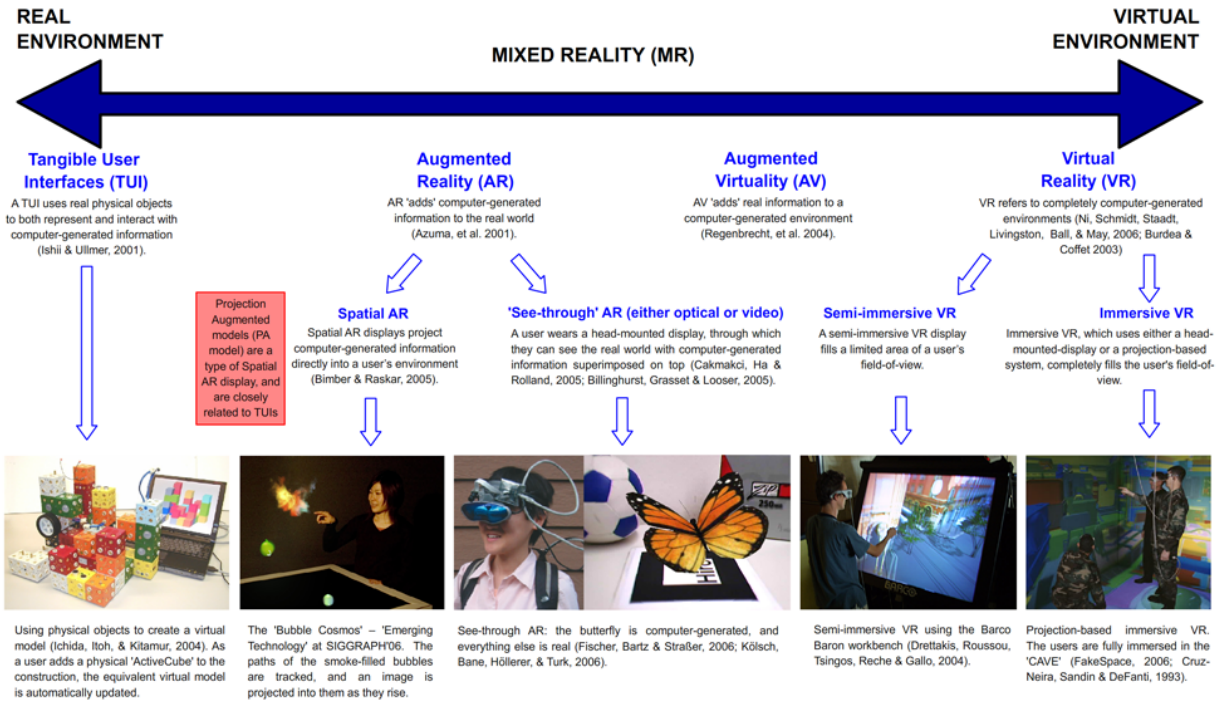


Figure 3.1: *Milgram's Continuum*(Wikimedia Commons 2011)

in this report since it includes ***registration*** and is not limited to any specific hardware devices. In Azuma's definition, the real environment is represented by image sequences or videos whereas the virtual environment refers to a virtual scene generated using a computer. When a dynamic virtual scene is combined with a dynamic real scene, we get an augmented scene. In order for the augmented scene to look realistic, the positions in the real scene where the virtual objects are added must be visually correct. For instance, in adding some virtual characters (say virtual cows) to a real scene of the North Campus Quad at the University of Michigan, Ann Arbor, the cows must appear to be standing on the grass or on the pathways and not flying at arbitrary positions. Figure 3.2 illustrates this. On the left of the image are a couple of cows at random positions in the real scene. On the right the cows are all on the ground plane, either on the grass or on the pathways and look more realistic. In the following section the factors that influence the visual accuracy of the augmented scene will be explored.

(a) Virtual cows in random positions (b) Virtual cows on the ground plane

Figure 3.2: *Example of an augmented scene*

3.2 The Registration Problem in AR

In AR, registration refers to the problem of aligning the real world coordinate system with the virtual object coordinate system. Figure 3.3 and 3.4 show the various coordinate systems associated with a typical AR system collectively and separately.

The real world coordinate system and the virtual object coordinate systems are 3D systems. Within each of these there is a 2D image coordinate system which is the 2D plane onto which the 3D points are projected for viewing. In the case of the real world coordinate system this is the camera and in the case of the virtual object coordinate system this is the viewport. In order to achieve registration the coordinate systems within the real world must be properly aligned with each other and, subsequently, the real world coordinates must be aligned with the virtual object coordinates. *Camera pose estimation* or *camera tracking* is the operation of finding the position and orientation (pose) of the camera with respect to the real world coordinate system and is explained

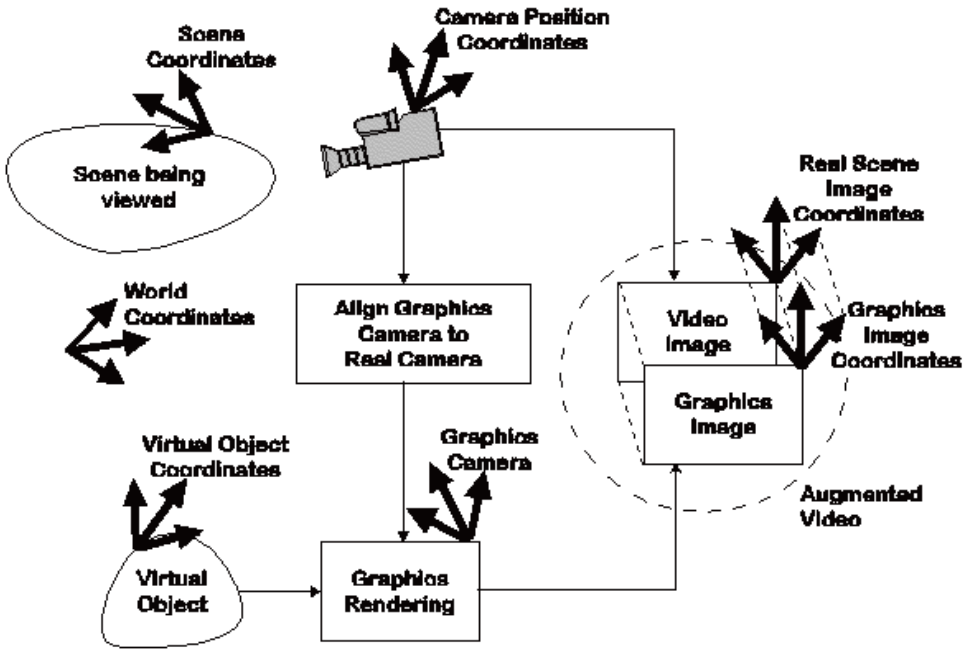


Figure 3.3: *Coordinate systems in a typical AR system (Vallino 1998)*

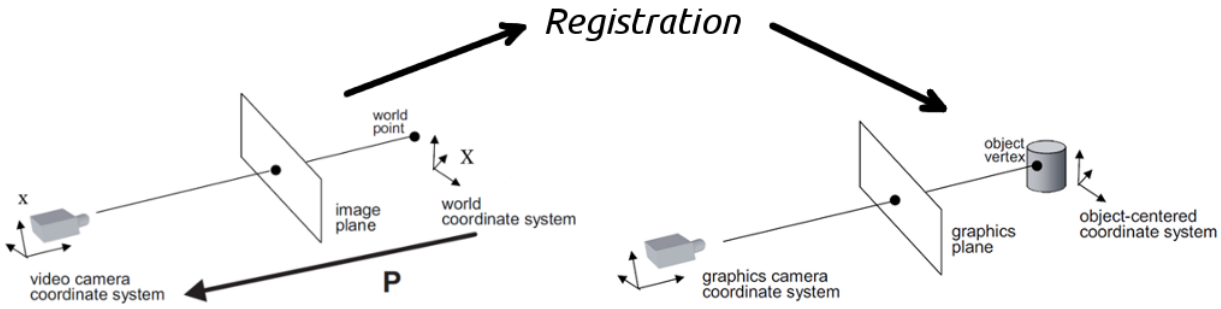


Figure 3.4: *Registration of real world and virtual object coordinate systems*

in section 3.2.1. The operation of aligning the virtual object coordinate system with the real world coordinate system is dependent on the graphics API used. In the case of *OpenSceneGraph*, which uses *OpenGL*, the transformations required are described by Li (2001).

3.2.1 Camera Pose Estimation

Cameras used to capture images of a 3D scene are usually *perspective* cameras that transform 3D points to a 2D point by means of a projection through a lens *i.e.* perspective projection. Perspective projection is modeled by the equation,

$$\mathbf{x} = \mathbf{P}\,\mathbf{X}, \tag{3.1}$$

where $\mathbf{X} = (X, Y, Z, 1)^{\mathrm{T}}$ is the homogeneous coordinate of a 3D point in the real world, $\mathbf{x} = (x, y, 1)^{\mathrm{T}}$ is the homogeneous coordinate of the projection of $\mathbf{X}$ on the image plane and $\mathbf{P}$ is the 3×4 projection matrix that models the perspective projection (Hartley and Zisserman 2003). Following the notation of Hartley and Zisserman (2003), the projection matrix, $\mathbf{P}$, can be decomposed as,

$$\mathbf{P} = \mathbf{K}\,[\mathbf{R} \mid \mathbf{t}], \tag{3.2}$$

where $\mathbf{K}$ is the camera calibration matrix, $\mathbf{R}$ is the rotation matrix that gives the orientation of the camera and $\mathbf{t}$ is the translation vector that gives the position of the camera. $\mathbf{K}$ encodes the parameters of the camera such as the focal length, image center and pixel skew. The procedure for calibrating a camera is briefly explained in section 3.3.2.

The projection matrix, $\mathbf{P}$ is the key to creating a realistic augmented scene. There are various techniques used to find the projection matrix as described in section 2.2. In the following sections the various stages of the markerless camera pose estimation system will be explained in detail. The output of this system is a series of projection matrices which can then be used in a 3D graphics API

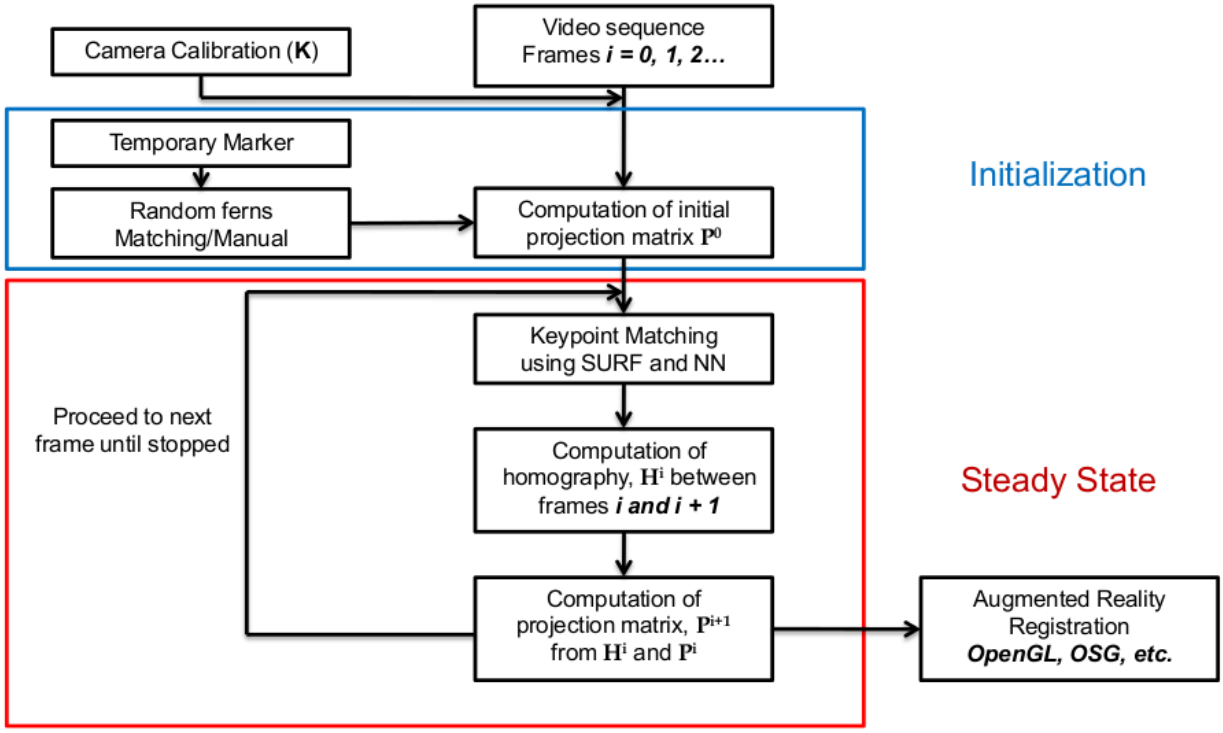


Figure 3.5: *CAMFPLAN system flow diagram*

such as *OpenSceneGraph* to synthesize an augmented scene.

3.3 CAMFPLAN System flow

The system flow diagram for CAMFPLAN is given in figure 3.5. There are two primary stages in this system *viz.* an initialization stage and a steady state stage. In the initialization stage the projection matrix corresponding to the first frame in the sequence is found by specifying the world coordinate system either manually or using a template matching algorithm. In the steady state stage the projection matrices corresponding to the frames after the first are found sequentially using the method proposed by Simon and Berger (2002); Simon et al. (2000).

3.3.1 Input

The input to the system is an image sequence or a video that is composed of sequential frames denoted by $i = 0, 1, 2 \ldots$ These frames are typically acquired from a video camera or a webcam. All the datasets in this report use webcams calibrated using the procedure explained in section 3.3.2. The important properties of the input video that affect the performance of the system include frame dimensions and frame rate (FPS). The effect of frame dimensions and the frame rate on the performance of the system are discussed in chapter 4.

3.3.2 Camera Calibration

As described earlier, the camera calibration matrix encodes the internal parameters of the camera such as the focal length, image center and pixel skew. The general structure of this matrix is given by (Hartley and Zisserman 2003),

$$\mathbf{K} = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.3}$$

where the f_x and f_y are the focal lengths along the x- and y- axes, x_0 and y_0 denote the image center and s is the pixel skew factor. Note that $\mathbf{K}$ is an upper triangular matrix with five degrees of freedom.

In the experiments conducted for CAMFPLAN, the camera calibration routine of *OpenCV* was used. The `cv::calibrateCamera` routine uses the Direct Linear Transform (DLT) algorithm (Tsai 1987) for calibration. This procedure requires a calibration pattern similar to the one shown in figure 3.6. The corners in the checkerboard pattern are detected in the image and compared to the known distance between the squares of the checkerboard pattern which gives a transformation that can be used to find $\mathbf{K}$. More details about this routine can be found in the *OpenCV* documentation

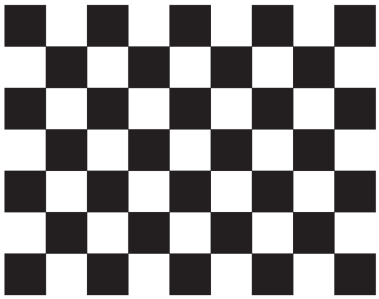


Figure 3.6: *Camera calibration pattern*

(OpenCV 2011).

3.3.3 Initialization

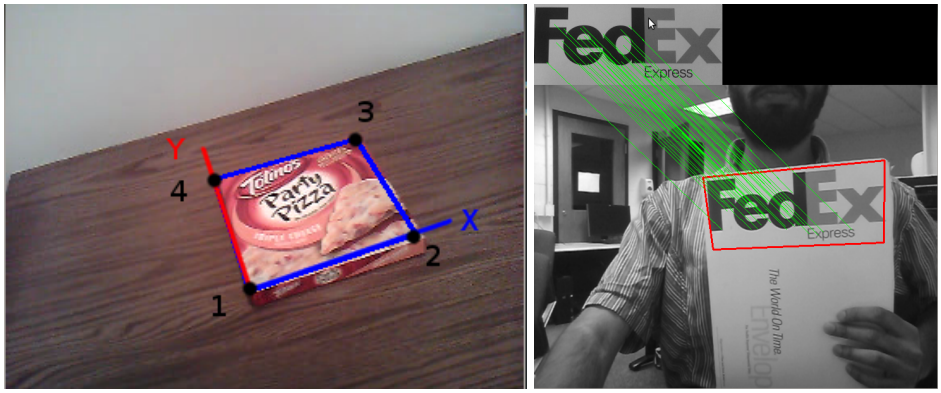
In the initialization stage the projection matrix, $\mathbf{P^0}$, corresponding to the frame, $i = 0$, is found. This is equivalent to aligning the position of the camera at frame $i = 0$ with the world coordinate system. Initialization can be done either manually by the user clicking on certain pre-defined points or automatically using a template matching algorithm.

Manual initialization involves the user knowing the position of the world coordinate system as seen in the first frame. Figure 3.7 (a) is a snapshot of CAMFPLAN in the manual initialization mode. It shows the first frame of the ***pizzabox*** sequence (see chapter 4) with four points clicked that define the x- and y- axes of the world coordinate frame. Thus, the plane where is the user selects the points is the $Z = 0$ plane. Once the four points are selected by the user, $\mathbf{P^0}$ can be found as follows.

The *homography* or *collineation* between two planes is given by the equation (Hartley and Zisserman 2003),

$$\mathbf{x}' = \mathbf{H}\,\mathbf{x}. \tag{3.4}$$

Consider a projection of the point $\mathbf{X} = (X, Y, Z, 1)^{\mathtt{T}}$ onto an image point $\mathbf{x} = (x, y, 1)^{\mathtt{T}}$ which is given by equation (3.1). From equation (3.2) we can further write $\mathbf{R} = [\mathbf{r_1}\ \mathbf{r_2}\ \mathbf{r_3}]$, where $\mathbf{r_1}, \mathbf{r_2}$ and $\mathbf{r_3}$ denote the *orthonormal* columns of the rotation matrix. In expanded form, equation (3.1) can be



(a) Manual initialization of the ***pizzabox*** sequence showing the x- and y- world axes

(b) Automatic initialization using random ferns

Figure 3.7: *Initialization in CAMFPLAN*

written as,

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{r_1} \; \mathbf{r_2} \; \mathbf{r_3} \; \mathbf{t} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}. \tag{3.5}$$

If, as a special case, the point $\mathbf{X}$ were to lie on the $Z = 0$ plane in the real world then equation (3.5) would reduce to,

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{r_1} \; \mathbf{r_2} \; \mathbf{r_3} \; \mathbf{t} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix},$$

$$\implies \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{K} \begin{bmatrix} \mathbf{r_1} \ \mathbf{r_2} \ \mathbf{t} \end{bmatrix} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \mathbf{H} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}, \tag{3.6}$$

where $\mathbf{H}$ is a homography that takes points on the $Z = 0$ plane in the world coordinate system to the xy-plane in the image coordinate system. Since the columns of $\mathbf{P}$ are orthonormal it is possible to find $\mathbf{P}$ uniquely given $\mathbf{H}$. This is because of the fact that $\mathbf{r_3} = \mathbf{r_1} \times \mathbf{r_2}$. Thus, we have,

$$\mathbf{P} = \mathbf{K} \begin{bmatrix} \mathbf{r_1} \ \mathbf{r_2} \ (\mathbf{r_1} \times \mathbf{r_2}) \ \mathbf{t} \end{bmatrix}. \tag{3.7}$$

CAMFPLAN works on the principle that when points on a plane in the real world are projected onto the image, the corresponding projection matrix reduces to a homography. Thus for the initialzation step the matrix, $\mathbf{P^0}$, can be computed by first finding the homography, $\mathbf{H^0}$, between the world coordinates of the points selected by the user and their corresponding image coordinates. This is illustrated in figure 3.8. Computation of the homography, $\mathbf{H^0}$, can be done using numerous algorithms such as RANSAC, LMedS, etc. (Hartley and Zisserman 2003) In CAMFPLAN the RANSAC algorithm is used.

3.3.4 Steady State

In the steady state stage of the system the projection matrices for every frame, $i > 0$, are found sequentially. Since this process depends only on the initialized projection matrix $\mathbf{P^0}$ and the frame-to-frame homographies, $\mathbf{H^i}$, it can continue sequentially in realtime. Thus, it is called the steady state stage of the system.

In order to find the frame to frame homographies the first step is to extract and match the *keypoints* between two successive frames. Keypoints are defined as those features on one image that can be easily identified in another image. Keypoints are generally points of high variance

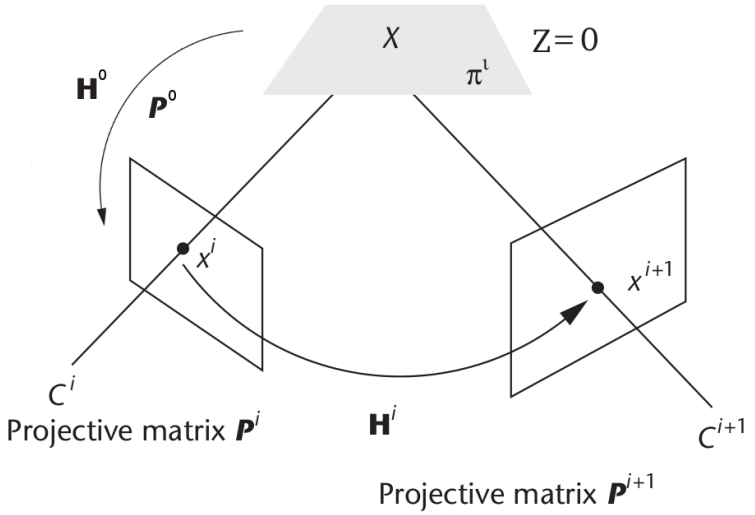


Figure 3.8: *Homography induced by a plane. Adapted from (Simon and Berger 2002)*

in the image such as an edge or a corner. Some of the important properties that keypoints must have include scale invariance, illumination invariance and pose invariance. Some of the common keypoint *detectors* used in the computer vision community include the Harris corner detector and derivative of gaussian (DoG) detector (Harris and Stephens 1988).

More recently, keypoint *descriptors* such as SIFT (Scale Invariant Feature Transform) and SURF (Speeded Up Robust Features) have been introduced (Bay et al. 2006). Descriptors are similar to detectors except that they define the characteristics of a region around a keypoint rather than just its position. Typically, descriptors are represented by a vector. CAMFPLAN uses SURF for feature extraction since the description of keypoints is in terms of a 64-length vectors as opposed to a 128-length vectors used by SIFT. This is found to considerably improve the per frame computation time.

Matching the SURF features in two frames is done using a nearest neighbor (NN) technique. Let $\mathbf{f_i^1}$ represent the SURF features for the i keypoints detected in the first frame and $\mathbf{f_j^2}$ represent the SURF keypoints for the j keypoints detected in the second frame. A keypoint l on the first frame ($\mathbf{f_l^1}$) is said to be matched to a keypoint k ($\mathbf{f_k^2}$) on the second frame if the Euclidean distance (L^2 norm) between $\mathbf{f_l^1}$ and $\mathbf{f_k^2}$ is the least among all possible forward combinations between l and j. This

Figure 3.9: *Keypoint extraction and matching in the* **pizzabox** *sequence*

vector norm based NN technique is very fast and suitable for realtime implementation. Figure 3.9 shows two frames from the ***pizzabox*** sequence matched this way.

Once all the keypoints are detected between the two frames the homography, $\mathbf{H}^{\mathbf{1}}_{\mathbf{p}}$, is computed using the method described in section 3.3.3. The RANSAC algorithm is used for estimating the homography since it eliminates outlier matches. Assuming that the plane occupies a major portion of the two frames this produces good estimates for the homography. Section 3.4.1 explores how this can be overcome by using a dominant plane detection algorithm.

Once the homography, $\mathbf{H}^{\mathbf{1}}_{\mathbf{p}}$, is estimated it can be used to find the projection matrix, $\mathbf{P^1}$, as follows. Let $\mathbf{H^1}$ denote the homography between the points on the plane $Z = 0$ and the first frame. Since transformations are linear we have $\mathbf{H^1} = \mathbf{H}^{\mathbf{1}}_{\mathbf{p}}\,\mathbf{H^0}$ (see figure 3.8).

We can find the projection matrix, $\mathbf{P^1}$, from $\mathbf{H^1}$ by exploiting the orthonormal columns in the rotation matrix contained in $\mathbf{P^1}$ according to equation (3.7). $\mathbf{P^1}$ can subsequently be used as given by Li (2001) for augmenting the real scene with virtual objects. The steady state process explained above is repeated for all frames, $i > 0$.

3.4 Improvements

As mentioned earlier there are several scenarios in which the algorithm described above performs poorly. One such scenario is in the matching stage where the matching has to happen between

points on the same plane. The RANSAC algorithm does eliminate some of the outliers (Simon and Berger 2002) but is not always consistent. Thus, an optical flow based dominant plane detection algorithm was tested with CAMFPLAN. Although better results were obtained it was not very helpful since optical flow is a computationally intensive operation.

3.4.1 Dominant Plane Detection

The detection of dominant plane using the method proposed by Ohnishi and Imiya (2006) requires two images with a small camera movement as the input. The term dominant plane refers to the plane which occupies the largest domain in an image. Two images with a small camera movement between them are first used to estimate the optical flow. Subsequently, the dominant plane motion (planar flow) is found as well. As described by Ohnishi and Imiya, the following geometric assumptions are made about the dominant plane.

1. The dominant plane always occupies more than 50% of the image.
2. There is at least one dominant plane visible in the image.
3. The dominant plane is a finite plane.

The flow chart in figure 3.10 shows the flowchart for the dominant plane detection. First, the method for finding the optical flow and planar flow are described individually followed by the actual dominant plane detection algorithm. The output of this this algorithm is a binary image with white indicating areas where the dominant plane was detected.

Optical Flow

Optical flow is defined as the pattern of motion of scene objects in an image. Optical flow characterizes the motion of objects, surfaces and edges in an image sequence. This motion has both magnitude & direction and therefore is described by a collection of motion vectors, $(\dot{u}, \dot{v})^{\mathrm{T}}$. Optical flow can be found by using several algorithms the most popular of which are Lucas et al. (1981)

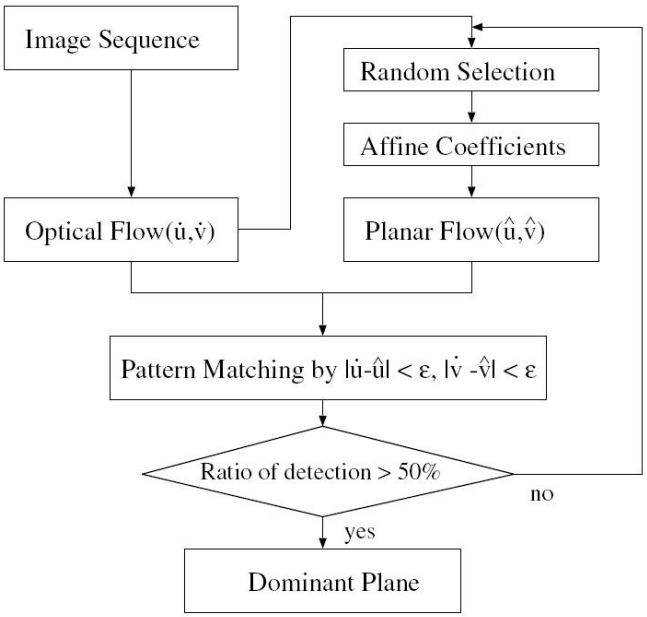


Figure 3.10: *Flowchart of the Dominant Plane Detection Algorithm*

and Horn and Schunck (1981). The general representation of optical flow is given by the equation,

$$\frac{\partial I}{\partial x}\frac{\partial V_x}{} + \frac{\partial I}{\partial y}\frac{\partial V_y}{} + \frac{\partial I}{\partial t} = 0, \tag{3.8}$$

where V_x and V_y are the x and y components of the optical flow of $I(x,y,t)$. Because this equation contains two unknowns special conditions are introduced by all methods to estimate the optical flow.

The Horn-Schunck method solves the optical flow globally, while Lucas-Kanade solves the optical flow in a local neighborhood of the pixels. The dominant plane detection algorithm is based on optical flow estimation, and its performance depends largely on correctness of the optical flow. Analysis of the results showed that Lucas-Kanade with pyramids as was used by Ohnishi and Imiya (2006) performed the best for dominant plane detection.

The Lucas-Kanade method sets a window of a certain size, 5×5 in the current case, and iteratively finds the solution to the optical flow equation by the least squares criterion. Lucas-Kanade

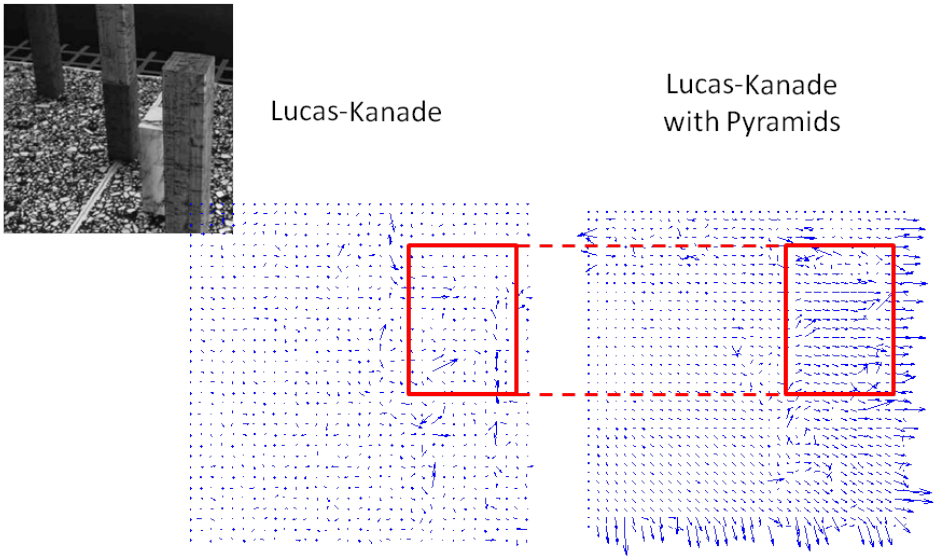


Figure 3.11: *Results from Lucas-Kanade and Lucas-Kanade with pyramids*

with pyramids finds the optical flow in a similar manner but for an image pyramid constructed from the original image. It solves the least squares equation from coarse to fine levels. At the highest level, a smaller number of pixels are used with Lucas-Kanade, and then the results are upsampled and taken to the next level, where even more pixels are again applied with Lucas-Kanade. This process continues to iteratively build a *pyramid*. In experiments, a 4-level pyramid was used and the result in figure 3.11 shows that Lucas-Kanade with pyramids successfully characterizes the object motion while Lucas-Kanade alone is unable to provide information in the uniform regions of the image.

Planar Flow

Given a small camera displacement the corresponding points on the dominant plane in a pair of successive images can be connected by a projection, which is represented by the means of an affine

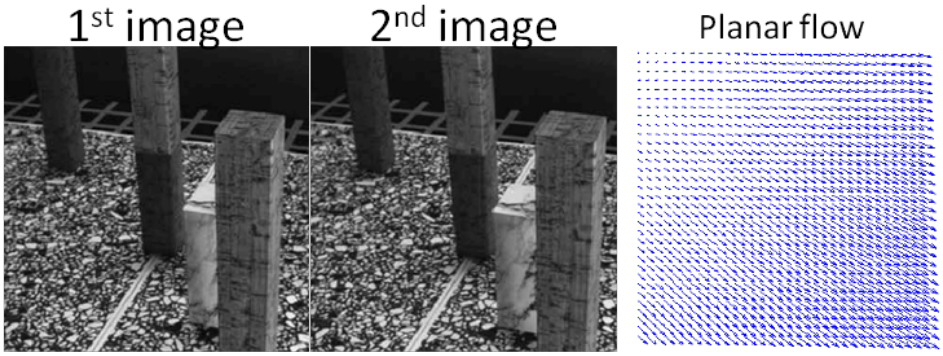


Figure 3.12: *The planar flow obtained from the first and second images*

transformation. This affine transformation can be written as,

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} c \\ f \end{bmatrix}, \tag{3.9}$$

where $(u,v)^{\mathrm{T}}$ and $(u',v')^{\mathrm{T}}$ are the corresponding points on the dominant plane solved by the optical flow estimation. When three points on the dominant plane are properly located (non-linearly), affine coefficients, a,b,c,d,e and f, can be uniquely solved. Once the affine coefficients are computed the motion of the image sequence can be computed as,

$$\begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} = \begin{bmatrix} u' \\ v' \end{bmatrix} - \begin{bmatrix} u \\ v \end{bmatrix}, \tag{3.10}$$

which can be regarded as the motion flow of the dominant plane on the basis of camera displacement. This is called the *planar flow* and is shown in figure 3.12.

Pattern Matching

Optical flow and planar flow were derived as the motion of the objects and the motion of the dominant plane respectively. So by matching the pattern of these two flows through a tolerance

Figure 3.13: *Results from (a) CAMFPLAN (b) Ohnishi and Imiya (2006)*

number, ε, the dominant plane can be found. This is represented by,

$$\left\| \begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} - \begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} \right\| < \varepsilon, \tag{3.11}$$

where the points $(u,v)^{\mathrm{T}}$ are accepted as points on the dominant plane when the equation is satisfied. Each time three pairs of corresponding points in the two images are randomly selected to find the planar flow and then the pattern matching is performed. If the detected plane occupies more than half of the image the dominant plane is considered found. If not, another set of random points are selected and the planar flow is recomputed. This procedure is repeated until the dominant plane is detected (figure 3.13 *(a)*). In addition, a Markov Random Field (MRF) based noise reduction technique was applied to the resulting images to clump the black and white regions together. Shown in figure 3.13 *(b)* is the results from Ohnishi and Imiya (2006). A simple visual comparison shows that the results obtained look similar to those of Ohnishi and Imiya (2006).

Chapter 4

Results and Discussion

In chapter 3 the plane based tracking technique of the system was described in detail with the help of results on some datasets. In order to asses the performance of the system in a specific scenario it is important to test the system using *standard* datasets. In the computer vision community standard datasets are used as a benchmark for assessing the performance of different algorithms. Examples of publicly available standard datasets are the Middlebury dataset (University of Middlebury 2011) for optical flow and the PASCAL dataset (University of Southampton 2011) for object recognition.

In this chapter the results of the system on a standard dataset are shown. Analysis of computation times and error analysis are also shown.

4.1 Video Datasets

For evaluating the performance of CAMFPLAN a dataset with specific characteristics was required. Firstly, the user needs to known the world coordinate system so that the initialization stage can be completed. Secondly, there has to be dominant plane in the image sequence so that the camera can be tracked successfully using the approach described earlier. Lastly, the calibration of the camera must be known. There are many standard datasets available that meet some of these requirements, but none that meet all. Thus, the system was evaluated using datasets acquired in the Laboratory

Sequence Name	Frame Size	No. of Frames	Length	Format
calibration	320×240	894	0:29	OGV Theora
hallway_down	320×240	595	0:19	OGV Theora
hallway_down2	320×240	533	0:17	OGV Theora
hallway_up	320×240	650	0:21	OGV Theora
notebook	320×240	351	0:11	OGV Theora
pizzabox	320×240	558	0:18	OGV Theora

Table 4.1: Video sequences obtained and their details

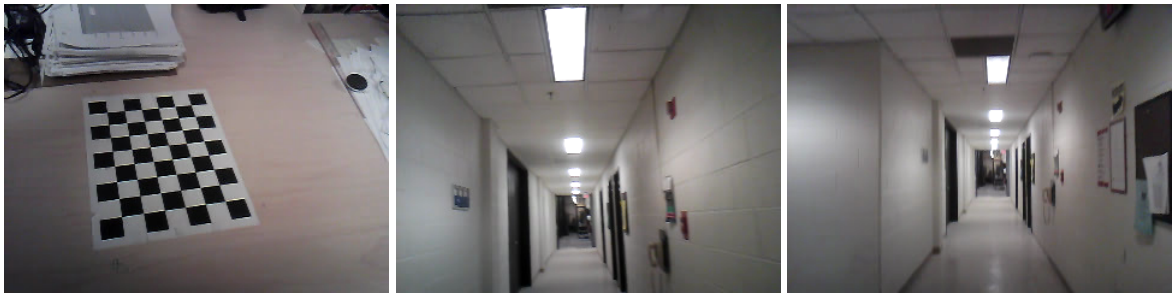
of Interactive Visualization (LIVE) at the University of Michigan, Ann Arbor. The video datasets were acquired using standard PC webcams. Each video sequence was given a name *e.g.* ***pizzabox***, ***notebook***, etc. The following table contains the details of all the datasets. Figure 4.1 shows one frame from each of the datasets acquired. All the datasets have been made available online[1].

4.2 Camera Calibration

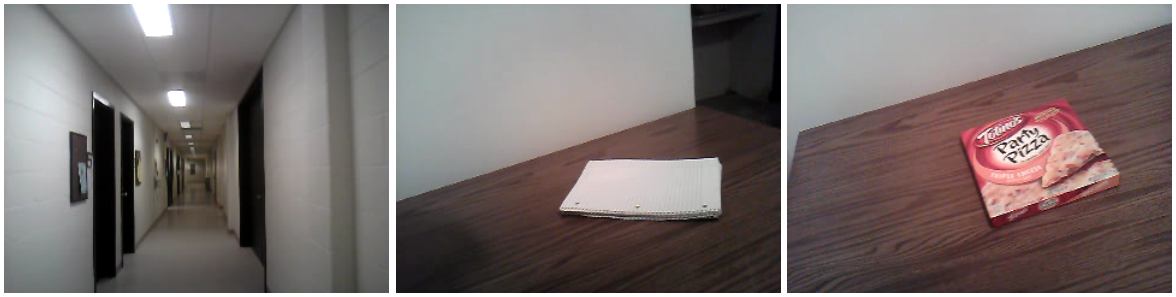
The camera used for all the video sequences was the same and thus a single calibration was sufficient. Calibration was performed using the *OpenCV* library's `cv::calibrateCamera` routine. As mentioned earlier, this routine uses the DLT algorithm and finds the parameters given in the camera calibration matrix $\mathbf{K}$ (equation (3.3)). The ***calibration*** sequence contains a calibration grid and this was used to calibrate the camera. The internal parameters in terms of frame pixel units were found to be,

$$\mathbf{K} = \begin{bmatrix} 201.502106 & 0. & 166.057602 \\ 0. & 184.796478 & 115.120590 \\ 0. & 0. & 1. \end{bmatrix}. \tag{4.1}$$

[1] http://www.umich.edu/~srinaths/live



(a) ***calibration*** Sequence (b) ***hallway_down*** Sequence (c) ***hallway_down2*** Sequence



(d) ***hallway_up*** Sequence (e) ***notebook*** Sequence (f) ***pizzabox*** Sequence

Figure 4.1: *Frame grabs from each of the datasets acquired*

4.3 Initialization

The result of manual initialization of the ***pizzabox*** sequence is shown in figure 4.2. The points marked $1, 2, 3$ and 4 are the manual click points and they define the XY-plane of the world. Automatic initialization was not performed on any of the datasets obtained but results of this are shown in figure 3.7.

4.4 Steady State

This section shows the results of the camera pose estimation on the acquired datasets. Since it is difficult to visualize the camera pose obtained the points tracked are *reprojected* onto the image. This visual assessment is equivalent to assessing the camera pose. Tiny variations in the camera pose affect the visual appearance of the reprojected points. Once the user initializes the system the system starts tracking the reprojected plane continuously. The results of the tracking for the

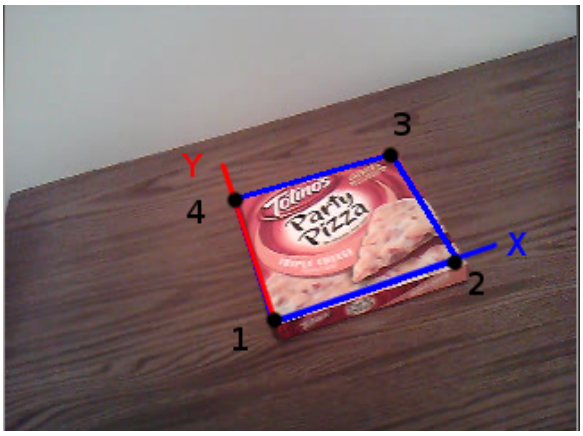


Figure 4.2: *Manual initialization of the* **pizzabox** *sequence*

pizzabox sequence are shown in figure 4.3.

An important observation to be made here is the fact that although it seems like the pizza box itself is being tracked, it is actually the plane that is tracked and not the pizza box. Thus, for every frame a projection matrix of the form given in equation (3.2) is computed. In order to demonstrate this, a simple augmentation of a unit cube on top of the pizza box is shown in figure 4.4. Figure 4.5 also shows incorrect tracking of the camera pose and, consequently, incorrect positioning of the cuboid augmented on the ***calibration*** sequence. The reason for this error is the clutter present in the image around the top which affects the image matching algorithm.

4.5 Dominant Plane Detection

Figure 4.6 shows the results of the dominant plane detection algorithm on the ***pizzabox*** sequence. As is clear from the figure the output does not look as expected. An important thing to note here is that the dominant plane detection algorithm is an iterative approach that needs to converge. Thus, it produces inconsistent results each time. Because of this and the issue of computation time this was not included in the final CAMFPLAN implementation.

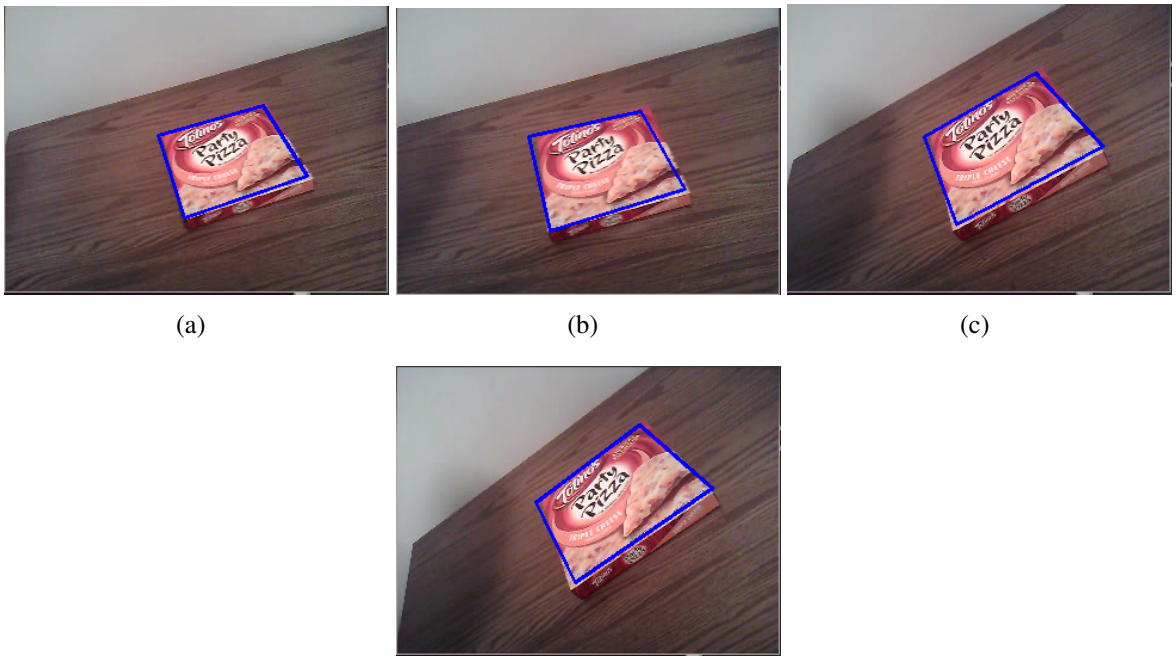


Figure 4.3: *Automatic tracking of the XY-plane*

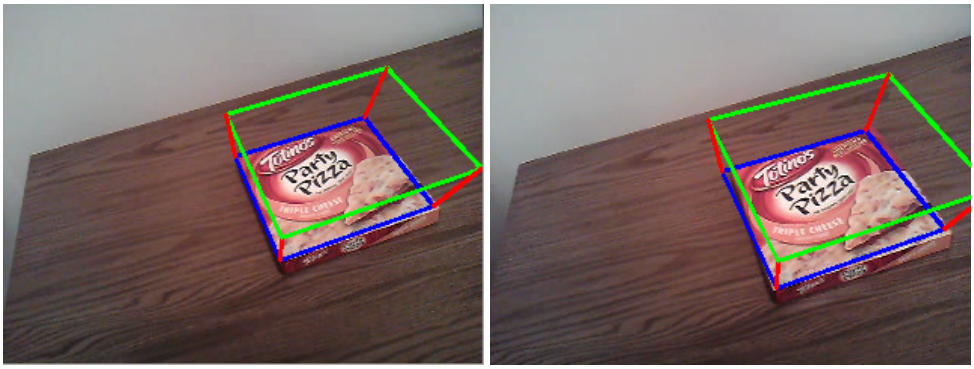
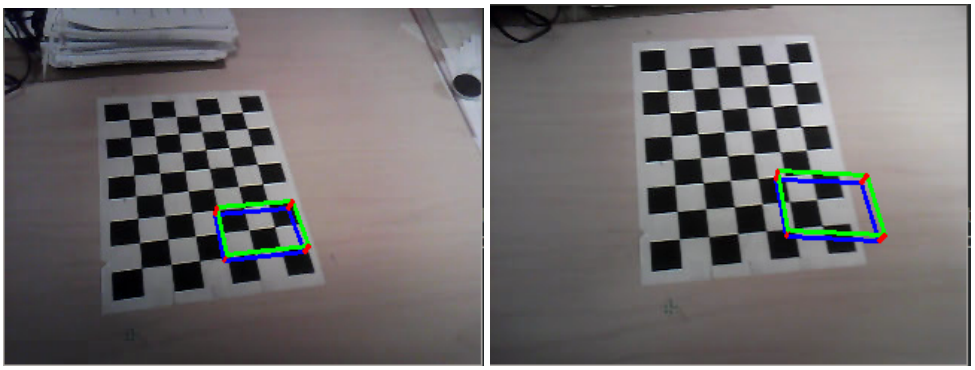


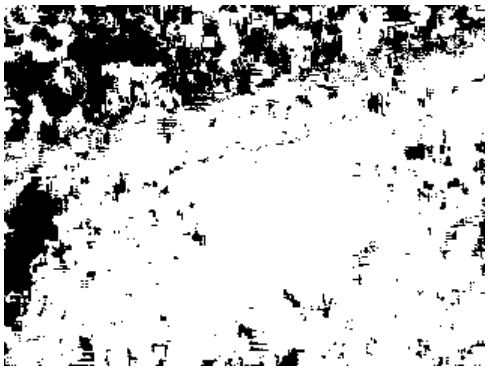
Figure 4.4: *Simple augmentation of a cube on the* **pizzabox** *sequence*



(a) (b)

Figure 4.5: *Incorrect augmentation of a cuboid on the* **calibration** *sequence*

(a) (b)



(c)

Figure 4.6: *Dominant plane detected from two frames of the* **pizzabox** *sequence*

Frame Size	**Initialization** (Random Ferns)	**Plane Tracking** (Matching)	**Plane Tracking** (Homography)
320×240	45 ms (22 FPS)	60 ms	10 ms
		70 ms (14.3 FPS)	

Table 4.2: Computation times for various stages of the system

4.6 Computation Time

The computation times for each component of the system and the system as a whole is given in table 4.2. The nominal FPS for the implementation is around 15 FPS. A system is said to have realtime performance when it is capable of being executed at 30 FPS. However, assuming camera pose does not change too abruptly, *frame skipping* may be employed to achieve realtime performance. In this technique every alternate frame is skipped during the steady state stage thus achieving realtime performance.

Chapter 5

Future Work and Conclusion

As was discussed in section 3.4 there are still many improvements that can be incorporated into the system. Some important improvements among these are robustness to varying lighting conditions and robustness to rapid camera motions. Several image processing techniques can be used to filter the input image sequence before being used for camera pose estimation. The image processing literature contains a lot of work on image filtering such as the one by Nitzberg and Shiota (1992).

Robustness to rapid camera motions has been explored by You et al. (2002) who use a multi-sensor fusion approach to tackle the registration problem. Aron et al. (2007) suggest a way in which inertial measurement units (IMU) can be used along with plane based techniques to achieve robust tracking. This approach is particularly suitable for implementation given that IMUs are readily available and are independent of external data sources (eg.: GPS).

In this report CAMFPLAN, a system for markerless camera pose estimation was presented. Results of the system using non-standard and standard datasets were shown with discussions on performance relative to other systems and ground truth. The potential applications of the system in construction engineering and management were also discussed. The appendix of this report contains the code listing for the C++ implementation of CAMFPLAN.

Appendix A

CAMFPLAN Code Listing

Figure A.1 shows the class diagram of CAMFPLAN. The C++ class definitions and descriptions are listed below.

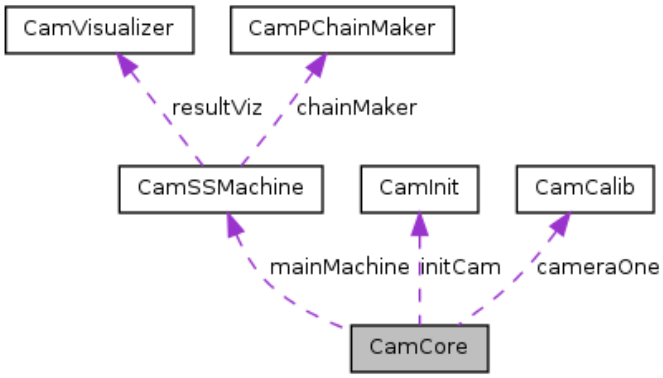


Figure A.1: *Class diagram of CAMFPLAN*

A.1 CAMFPLAN `CamCore`

```
1 #ifndef _CAMCORE_H_
2 #define _CAMCORE_H_
3
4 #include <QDebug>
5
```

```
6  #include "CamCalib.hpp"
7  #include "CamInit.hpp"
8  #include "CamSSMachine.hpp"
9
10 class CamCore
11 {
12 public:
13         CamCore();
14         virtual ~CamCore();
15
16         // Init function
17         void init(void);
18   void setCalib(QString incCalibFileName);
19   void startInit(QString incInitFileName);
20   void previewResult(void);
21
22 private:
23   CamCalib * cameraOne;
24   CamInit * initCam;
25   CamSSMachine * mainMachine;
26
27   QString incVideoFile;
28 };
29
30 #endif // _CAMCORE_HPP_
```

```
1  #include "CamCore.hpp"
2
3  CamCore::CamCore()
4  {
5    cameraOne = new CamCalib;
6    initCam = new CamInit;
7    mainMachine = new CamSSMachine;
8  }
9
10 CamCore::~CamCore()
11 {
12   delete cameraOne;
13   delete mainMachine;
14   delete initCam;
15 }
16
17 // 1. Calibrate camera
18 void CamCore::setCalib(QString incCalibFileName)
19 {
20   cameraOne->init(incCalibFileName);
```

```
21 }
22
23 // 2. Initialization step
24 void CamCore::startInit(QString incInitFileName)
25 {
26   incVideoFile = incInitFileName;
27   initCam->setFrameLimits(0, 500); // Set frame start and end
28   initCam->init(cameraOne->getCamKMatrix(), incVideoFile);
29 }
30
31 // 3. Start steady state machine
32 void CamCore::previewResult(void)
33 {
34   mainMachine->setFrameLimits(1, 500); // Set fram start and end
35   mainMachine->init(initCam, incVideoFile);
36 }
37
38 void CamCore::init(void)
39 {
40
41 }
```

A.2 CAMFPLAN `CamCalib`

```
1  #ifndef _CAMCALIB_H_
2  #define _CAMCALIB_H_
3
4  #include <cv.h>
5  #include <cvaux.h>
6  #include <cxcore.h>
7  #include <highgui.h>
8  #include <QDebug>
9
10 class CamCalib
11 {
12 public:
13         CamCalib();
14         virtual ~CamCalib();
15
16         // Init function
17         void init(QString incCalibFileName);
18
19   // Get camera internal matrix from outside
20   CvMat * getCamKMatrix(void){ return camInternMatrix; }
```

```
21
22  private:
23    // File import function
24    void camCalibImport(void);
25
26    // Calibration from video function
27    void camCalibVideo(void);
28
29    // Calibration Data for import
30    CvMat * camInternMatrix;
31    QString calibFileName;
32  };
33
34  #endif // _CAMCALIB_HPP_
```

```
1   #include "CamCalib.hpp"
2
3   CamCalib::CamCalib()
4   {
5
6   }
7
8   CamCalib::~CamCalib()
9   {
10    cvReleaseMat(&camInternMatrix);
11  }
12
13  void CamCalib::init(QString incCalibFileName)
14  {
15    // STEPS
16    // 1. Calibrate camera first. Have 2 options for this - import file
        or do calibration
17    // Also have option to export calibration information if doing
        yourself
18
19    calibFileName = incCalibFileName;
20    camCalibImport();
21  //  camCalibVideo();
22  }
23
24  void CamCalib::camCalibImport(void)
25  {
26    camInternMatrix = (CvMat*) cvLoad(calibFileName.toAscii());
27
28    if(camInternMatrix != NULL)
```

```
29      qDebug() << "Calibration matrix loaded successfully from file." <<
            endl;
30    else
31      qDebug() << "Calibration matrix load failure." << endl;
32  }
33
34  void CamCalib::camCalibVideo(void)
35  {
36    // Done separately
37  }
```

A.3 CAMFPLAN CamInit

```
1   #ifndef _CAMINIT_H_
2   #define _CAMINIT_H_
3
4   #include <cv.h>
5   #include <cvaux.h>
6   #include <cxcore.h>
7   #include <highgui.h>
8   #include <QDebug>
9   #include <algorithm>
10  #include <iostream>
11  #include <vector>
12
13  using namespace cv;
14
15  class CamInit
16  {
17  public:
18          CamInit();
19          virtual ~CamInit();
20
21          // Init function
22          void init(CvMat * incKMatrix, QString incInitFileName);
23
24    // Get points from user
25    void getUserPoints(QString incInitFileName);
26
27    // Get imgPlane
28    CvMat * getUserClicks(void){ return imgPlane; };
29
30    // Find P
31    void findProjection(CvMat * incKMatrix);
```

```
32
33    // Set start and end frames
34    void setFrameLimits(int incStart, int incEnd);
35
36    // Get projection matrix
37    CvMat * getProjection(void){ return projectionMat; };
38
39    // Find homography matrix
40    CvMat * getHomography(void){ return homography; };
41
42    // Fern training based matcher
43    void matchWithDatabase(void);
44
45    void setMWDParams(void);
46
47  private:
48    IplImage * tmpFrame, * dispFrame;
49    int startFrm, endFrm;
50
51    CvMat * calibPlane, * imgPlane;
52    CvMat * homography, * projectionMat;
53
54    // Template matching using Fern Classifier
55    CvCapture * frameCap;
56    QString templateFName, sceneFName;
57    // Using OpenCV C++ API from now that I am learning
58    Mat templateImg;
59    double imgscale;
60    Size patchSize;
61    LDetector ldetector;
62    PlanarObjectDetector detector;
63    vector<Mat> objpyr, imgpyr;
64    int blurKSize;
65    double sigma;
66    vector<KeyPoint> objKeypoints, imgKeypoints;
67    PatchGenerator gen;
68    FileStorage fs;
69    vector<Point2f> dst_corners;
70  };
71
72  #endif // _CAMINIT_HPP_
```

```
1  #include "CamInit.hpp"
2  #include "helper.hpp"
3
4  int pointCount = 0;
```

```
5  void mouseCallFn(int button, int x, int y, int state, void *
       incImgPlane)
6  {
7    if(button == CV_EVENT_LBUTTONDOWN)
8          {
9      if(pointCount == 0)
10     {
11       cvmSet((CvMat *) incImgPlane, pointCount, 0, (double) x);
12       cvmSet((CvMat *) incImgPlane, pointCount, 1, (double) y);
13     }
14     else if(pointCount == 1)
15     {
16       cvmSet((CvMat *) incImgPlane, pointCount, 0, (double) x);
17       cvmSet((CvMat *) incImgPlane, pointCount, 1, (double) y);
18     }
19     else if(pointCount == 2)
20     {
21       cvmSet((CvMat *) incImgPlane, pointCount, 0, (double) x);
22       cvmSet((CvMat *) incImgPlane, pointCount, 1, (double) y);
23     }
24     else if(pointCount == 3)
25     {
26       cvmSet((CvMat *) incImgPlane, pointCount, 0, (double) x);
27       cvmSet((CvMat *) incImgPlane, pointCount, 1, (double) y);
28     }
29     else
30       qDebug() << "There␣are␣4␣points␣already.␣Press␣any␣key␣to␣
             continue...";
31     qDebug() << x << y;
32     pointCount++;
33   }
34 }
35
36 CamInit::CamInit()
37 {
38   projectionMat = cvCreateMat(3, 4, CV_64FC1);
39   homography = cvCreateMat(3, 3, CV_64FC1);
40   imgPlane = cvCreateMat(4, 2, CV_64FC1);
41   calibPlane = cvCreateMat(4, 2, CV_64FC1);
42
43   // Assuming square of side 1
44   cvmSet(calibPlane, 0, 0, 0.0);
45   cvmSet(calibPlane, 0, 1, 0.0);
46   cvmSet(calibPlane, 1, 0, 1.0);
47   cvmSet(calibPlane, 1, 1, 0.0);
48   cvmSet(calibPlane, 2, 0, 1.0);
```

```
49    cvmSet(calibPlane, 2, 1, 1.0);
50    cvmSet(calibPlane, 3, 0, 0.0);
51    cvmSet(calibPlane, 3, 1, 1.0);
52  }
53
54  CamInit::~CamInit()
55  {
56    cvReleaseMat(&calibPlane);
57    cvReleaseMat(&imgPlane);
58    cvReleaseMat(&homography);
59    cvReleaseMat(&projectionMat);
60
61    cvReleaseCapture(&frameCap);
62    cvReleaseImage(&tmpFrame);
63    cvReleaseImage(&dispFrame);
64
65    delete imgPlane;
66    delete calibPlane;
67  }
68
69  void CamInit::setFrameLimits(int incStart, int incEnd)
70  {
71    startFrm = incStart;
72    endFrm = incEnd;
73  }
74
75  void CamInit::findProjection(CvMat * incKMatrix)
76  {
77    // Find simplified homography first
78    cvFindHomography(calibPlane, imgPlane, homography, CV_RANSAC, 1);
79    findPfromH(homography, incKMatrix, projectionMat);
80    cvSave("frame0.xml", projectionMat);
81  }
82
83  void CamInit::getUserPoints(QString incInitFileName)
84  {
85    frameCap = cvCaptureFromFile(incInitFileName.toAscii());
86    if(!frameCap)
87    {
88      qDebug() << "CamInit::getUserPoints(QString␣incInitFileName):␣Error
          ,␣cannot␣read␣from␣file.";
89    }
90
91    cvSetCaptureProperty(frameCap, CV_CAP_PROP_POS_FRAMES, startFrm);
92    tmpFrame = cvQueryFrame(frameCap);
```

```
93    dispFrame = cvCreateImage(cvSize(tmpFrame->width, tmpFrame->height),
          IPL_DEPTH_8U, 3);
94    cvCopy(tmpFrame, dispFrame);
95
96    cvNamedWindow("Initialize Camera", 1);
97    cvShowImage("Initialize Camera", dispFrame);
98
99    cvSetMouseCallback("Initialize Camera", mouseCallFn, imgPlane);
100   cvWaitKey(0);
101
102     // SETTING ALL TO DEFAULT points for now - Frame 0
103     cvmSet(imgPlane, 0, 0, 185.0);
104     cvmSet(imgPlane, 0, 1, 172.0);
105
106     cvmSet(imgPlane, 1, 0, 288.0);
107     cvmSet(imgPlane, 1, 1, 136.0);
108
109     cvmSet(imgPlane, 2, 0, 242.0);
110     cvmSet(imgPlane, 2, 1, 77.0);
111
112     cvmSet(imgPlane, 3, 0, 156.0);
113     cvmSet(imgPlane, 3, 1, 103.0);
114
115   cvDestroyWindow("Initialize Camera");
116 }
117
118 void CamInit::setMWDParams(void)
119 {
120   imgscale = 1;
121   patchSize = Size(32, 32);
122   ldetector = LDetector(7, 20, 2, 2000, patchSize.width, 2);
123   blurKSize = 3;
124   sigma = 0;
125   gen = PatchGenerator(0,256,5,true,0.8,1.2,-CV_PI/2,CV_PI/2,-CV_PI/2,
          CV_PI/2);
126
127   sceneFName = QString("../../data/matching/fedex_small/fedex_small.
          jpg_model.xml.gz");
128   fs = FileStorage((const char *) sceneFName.toAscii(), FileStorage::
          READ);
129   if(fs.isOpened())
130   {
131       detector.read(fs.getFirstTopLevelNode());
132       qDebug() << "Successfully loaded training data";
133   }
134   else
```

```
135     {
136         printf("The file not found and can not be read. Let's train the
            model.\n");
137         printf("Step 1. Finding the robust keypoints ...\n");
138         ldetector.setVerbose(true);
139         ldetector.getMostStable2D(templateImg, objKeypoints, 100, gen);
140         printf("Done.\nStep 2. Training ferns-based planar object
            detector ...\n");
141         detector.setVerbose(true);
142
143         GaussianBlur(templateImg, templateImg, Size(blurKSize, blurKSize)
            , sigma, sigma);
144         buildPyramid(templateImg, objpyr, ldetector.nOctaves-1);
145         detector.train(objpyr, objKeypoints, patchSize.width, 100, 11,
            10000, ldetector, gen);
146         printf("Done.\nStep 3. Saving the model to %s ...\n", (const char
             *) sceneFName.toAscii());
147         if( fs.open((const char *) sceneFName.toAscii(), FileStorage::
            WRITE) )
148             detector.write(fs, "ferns_model");
149     }
150     fs.release();
151
152
153     ldetector.setVerbose(true);
154 }
155
156 void CamInit::matchWithDatabase(void)
157 {
158   // Using fedex_small as it seems to produce best results
159   cvNamedWindow("Object Correspondence", 1);
160
161   templateFName = QString("../../data/matching/fedex_small/fedex_small.
        jpg");
162   templateImg = imread((const char *) templateFName.toAscii(),
        CV_LOAD_IMAGE_GRAYSCALE);
163   if(!templateImg.data)
164     qDebug() << "Problem loading database image";
165
166   setMWDParams();
167
168 //     frameCap = cvCaptureFromAVI(video_filename);
169   frameCap = cvCaptureFromCAM(1);
170   assert(frameCap);
171
172   int i;
```

```
173     for(;;)
174     {
175       Mat color_image = cvQueryFrame(frameCap);
176       Mat _image, image;
177       cvtColor(color_image, _image, CV_BGR2GRAY);
178       resize(_image, image, Size(), 1./imgscale, 1./imgscale, INTER_CUBIC
              );
179
180       GaussianBlur(image, image, Size(blurKSize, blurKSize), sigma, sigma
              );
181       buildPyramid(image, imgpyr, ldetector.nOctaves-1);
182
183       Mat correspond( templateImg.rows + image.rows, std::max(templateImg
              .cols, image.cols), CV_8UC3);
184       correspond = Scalar(0.);
185       Mat part(correspond, Rect(0, 0, templateImg.cols, templateImg.rows)
              );
186       cvtColor(templateImg, part, CV_GRAY2BGR);
187       part = Mat(correspond, Rect(0, templateImg.rows, image.cols, image.
              rows));
188       cvtColor(image, part, CV_GRAY2BGR);
189
190       vector<int> pairs;
191       Mat H;
192
193       double t = (double)getTickCount();
194       objKeypoints = detector.getModelPoints();
195       ldetector(imgpyr, imgKeypoints, 300);
196
197       bool found = detector(imgpyr, imgKeypoints, H, dst_corners, &pairs)
              ;
198       t = (double)getTickCount() - t;
199
200       if(found)
201       {
202           for( i = 0; i < 4; i++ )
203           {
204               Point r1 = dst_corners[i%4];
205               Point r2 = dst_corners[(i+1)%4];
206               line( correspond, Point(r1.x, r1.y+templateImg.rows),
207                     Point(r2.x, r2.y+templateImg.rows), Scalar(0,0,255), 3
                          );
208           }
209       }
210
211       for( i = 0; i < (int)pairs.size(); i += 2 )
```

```
212     {
213         line( correspond, objKeypoints[pairs[i]].pt,
214               imgKeypoints[pairs[i+1]].pt + Point2f(0,templateImg.rows),
215               Scalar(0,255,0));
216     }
217     imshow( "Object Correspondence", correspond );
218     int key = cvWaitKey(5);
219     bool exitFlag = false;
220     bool saveState = false;
221     if(key >= 0)
222     {
223       switch(char(key))
224       {
225         case 'c':
226           saveState = true;
227           break;
228         case 27:
229           exitFlag = true;
230           break;
231       }
232     }
233     if(exitFlag)
234       break;
235     if(saveState)
236     {
237       cvmSet(imgPlane, 0, 0, dst_corners[3].x);
238       cvmSet(imgPlane, 0, 1, dst_corners[3].y);
239
240       cvmSet(imgPlane, 1, 0, dst_corners[2].x);
241       cvmSet(imgPlane, 1, 1, dst_corners[2].y);
242
243       cvmSet(imgPlane, 2, 0, dst_corners[1].x);
244       cvmSet(imgPlane, 2, 1, dst_corners[1].y);
245
246       cvmSet(imgPlane, 3, 0, dst_corners[0].x);
247       cvmSet(imgPlane, 3, 1, dst_corners[0].y);
248
249       qDebug() << "Done saving coordinates";
250       cvDestroyAllWindows();
251
252       break;
253     }
254   }
255 }
256
257 void CamInit::init(CvMat * incKMatrix, QString incInitFileName)
```

```
258 {
259   // Will be replaced by ferns matching
260   getUserPoints(incInitFileName);
261
262 //   matchWithDatabase();
263   cvReleaseCapture(&frameCap);
264
265   qDebug() << "Computing projection matrix...";
266   findProjection(incKMatrix);
267   qDebug() << "Done!";
268 }
```

A.4 CAMFPLAN CamSSMachine

```
1  #ifndef _CAMSSMACHINE_H_
2  #define _CAMSSMACHINE_H_
3
4  #include <cv.h>
5  #include <cvaux.h>
6  #include <cxcore.h>
7  #include <highgui.h>
8  #include <iostream>
9  #include <QDebug>
10 #include <vector>
11
12 #include "CamPChainMaker.hpp"
13 #include "CamInit.hpp"
14 #include "CamVisualizer.hpp"
15 #include "helper.hpp"
16 #include "matcherSURF.hpp"
17
18 class CamSSMachine
19 {
20 public:
21         CamSSMachine();
22         virtual ~CamSSMachine();
23
24         // Init function
25         void init(CamInit * incInitCam, QString incVideoFile);
26   void setFrameLimits(int incStart, int incEnd);
27
28 private:
29   CamPChainMaker * chainMaker;
30
```

```
31    // Capture stuff
32    CvCapture * capPast;
33    CvCapture * capPresent;
34    CvCapture * capLive;
35    IplImage * pastFrame;
36    IplImage * presentFrame;
37    IplImage * pastFrameGray;
38    IplImage * presentFrameGray;
39    int capFPS, frameCount;
40    int startFrm, endFrm, skipCount, keyFrame;
41    CvSize frameSz;
42
43    // Looping matching stuff
44    CvMat * pastMatches;
45    CvMat * presentMatches;
46    int numMatches;
47
48    // Visualization
49    CamVisualizer * resultViz;
50
51    // Initialize capture parameters
52    void initCaptureParams(QString incVideoFile);
53
54    // Matcher Loop - video
55    void startMatcherLoop(CvMat * incUserClicks);
56
57    // Matcher Loop - live cam
58    void startLiveFeedLoop(void);
59 };
60
61 #endif // _CAMSSMACHINE_HPP_
```

```
1  #include "CamSSMachine.hpp"
2
3  CamSSMachine::CamSSMachine()
4  {
5    chainMaker = new CamPChainMaker;
6    skipCount = 0;
7    keyFrame = 0;
8  }
9
10 CamSSMachine::~CamSSMachine()
11 {
12   delete chainMaker;
13   delete resultViz;
14
```

```
15    cvReleaseCapture(&capPast);
16    cvReleaseCapture(&capPresent);
17    cvReleaseImage(&pastFrame);
18    cvReleaseImage(&presentFrame);
19    cvReleaseImage(&pastFrameGray);
20    cvReleaseImage(&presentFrameGray);
21
22    cvReleaseMat(&pastMatches);
23    cvReleaseMat(&presentMatches);
24
25    delete resultViz;
26  }
27
28  void CamSSMachine::setFrameLimits(int incStart, int incEnd)
29  {
30    startFrm = incStart;
31    endFrm = incEnd;
32    keyFrame = startFrm;
33  }
34
35  void CamSSMachine::initCaptureParams(QString incVideoFile)
36  {
37    // Creating two captures - TODO: Live webcam feeds
38    capPast = cvCaptureFromFile(incVideoFile.toAscii());
39    capPresent = cvCaptureFromFile(incVideoFile.toAscii());
40    if(!capPast || !capPresent)
41      qDebug() << "ERROR: One of the captures is NULL \n" << endl;
42
43    capFPS = cvGetCaptureProperty(capPast, CV_CAP_PROP_FPS); // Can use
        any
44    frameCount = cvGetCaptureProperty(capPast, CV_CAP_PROP_FRAME_COUNT);
45    qDebug() << "Video FPS is " << capFPS << "\tFrame count is " <<
        frameCount;
46
47    // Allocate memory
48    frameSz.width = cvGetCaptureProperty(capPresent,
        CV_CAP_PROP_FRAME_WIDTH);
49    frameSz.height = cvGetCaptureProperty(capPresent,
        CV_CAP_PROP_FRAME_HEIGHT);
50    pastFrameGray = cvCreateImage(frameSz, IPL_DEPTH_8U, 1);
51    presentFrameGray = cvCreateImage(frameSz, IPL_DEPTH_8U, 1);
52    resultViz = new CamVisualizer;
53  }
54
55  void CamSSMachine::startMatcherLoop(CvMat * incUserClicks)
56  {
```

```
57    cvSetCaptureProperty(capPresent, CV_CAP_PROP_POS_FRAMES, startFrm);
58    double ttime, msFrm = 0.0;
59
60    int k, l;
61    l = 1;
62    for(k = 1; ; k += 1)
63    {
64      // NOTE: keyFrame has become one of the parameters of your system
65      // changing this will work well or not depending on scenario.
66      // Try to make this adaptive if possible
67  //    if((k-1)%5 == 0 && k > 1)
68  //      keyFrame += 5;
69  //    keyFrame = l - skipCount - 1;
70      keyFrame = k - 1;
71  //    qDebug() << keyFrame << k;
72
73      ttime = (double) cvGetTickCount();
74
75      cvSetCaptureProperty(capPast, CV_CAP_PROP_POS_FRAMES, startFrm +
           keyFrame);
76      pastFrame = cvQueryFrame(capPast);
77
78      cvSetCaptureProperty(capPresent, CV_CAP_PROP_POS_FRAMES, startFrm +
           k);
79      presentFrame = cvQueryFrame(capPresent);
80
81      // Convert to grayscale
82      cvCvtColor(pastFrame, pastFrameGray, CV_RGB2GRAY);
83      cvCvtColor(presentFrame, presentFrameGray, CV_RGB2GRAY);
84
85      // NOTE: SURFParams is another of your system's parameters
86      // Adaptive possible?
87      findSURFMatches(pastFrameGray, presentFrameGray, 700, 1, true,
88                      &pastMatches, &presentMatches, &numMatches);
89
90      if(numMatches < 4)
91      {
92        skipCount++;
93        continue;
94      }
95
96      chainMaker->init(pastMatches, presentMatches, keyFrame);
97      msFrm += ((double)cvGetTickCount() - ttime)/(1000.0 *
           cvGetTickFrequency());
98
```

```
99      if((resultViz->init(chainMaker->getProjMat(l - skipCount),
            presentFrame)) || (k + startFrm == endFrm) || (k == frameCount -
            1))
100       break;
101
102 //      QString fileName, fileCount;
103 //      fileName = QString("dump/frames/");
104 //      fileCount = QString::number(k, 10);
105 //      while(fileCount.size() < 4)
106 //        fileCount.prepend(QString('0'));
107 //      fileName += fileCount + QString(".png");
108
109 //      cvSaveImage(fileName.toAscii(), presentFrame);
110     l++;
111   }
112   msFrm /= (k - skipCount);
113   qDebug() << "Actual frame rate is" << 1000.0/msFrm << "fps";
114
115   cvReleaseCapture(&capPast);
116   cvReleaseCapture(&capPresent);
117 }
118
119 void CamSSMachine::startLiveFeedLoop(void)
120 {
121   capLive = cvCaptureFromCAM(1);
122   if(!capLive)
123     qDebug() << "ERROR: Unable to capture from camera \n" << endl;
124
125   frameSz.width = cvGetCaptureProperty(capLive, CV_CAP_PROP_FRAME_WIDTH
        );
126   frameSz.height = cvGetCaptureProperty(capLive,
        CV_CAP_PROP_FRAME_HEIGHT);
127   pastFrameGray = cvCreateImage(frameSz, 8, 1);
128   presentFrameGray = cvCreateImage(frameSz, 8, 1);
129   resultViz = new CamVisualizer;
130
131   double ttime, msFrm = 0.0;
132
133   IplImage * tmpFrm;
134   int k, l;
135   l = 1;
136   for(k = 1; ; k += 1)
137   {
138     if(k == 1)
139     {
140       tmpFrm = cvQueryFrame(capLive);
```

```
141         pastFrame = cvCreateImage(frameSz, 8, 3);
142         cvCopy(tmpFrm, pastFrame);
143 //        cvReleaseImage(&tmpFrm);
144         continue;
145       }
146
147       ttime = (double) cvGetTickCount();
148
149       presentFrame = cvQueryFrame(capLive);
150
151       // Convert to grayscale
152       cvCvtColor(pastFrame, pastFrameGray, CV_BGR2GRAY);
153       cvCvtColor(presentFrame, presentFrameGray, CV_BGR2GRAY);
154
155       findSURFMatches(pastFrameGray, presentFrameGray, 700, 1, true,
156                       &pastMatches, &presentMatches, &numMatches);
157
158       if(numMatches < 4)
159       {
160         skipCount++;
161         continue;
162       }
163
164       chainMaker->init(pastMatches, presentMatches, keyFrame);
165       msFrm += ((double)cvGetTickCount() - ttime)/(1000.0 *
              cvGetTickFrequency());
166
167 //      if(k > 1)
168 //      {
169 //        tmpFrm = cvQueryFrame(capLive);
170 //        pastFrame = cvCreateImage(frameSz, 8, 3);
171 //        cvCopy(tmpFrm, pastFrame);
172 //        cvReleaseImage(&tmpFrm);
173 //      }
174
175       if((resultViz->init(chainMaker->getProjMat(l - skipCount),
              presentFrame)))
176       {
177         break;
178       }
179       l++;
180     }
181     msFrm /= (k - skipCount);
182     qDebug() << "Actual frame rate is" << 1000.0/msFrm << "fps";
183
184     cvReleaseCapture(&capLive);
```

```
185 }
186
187 void CamSSMachine::init(CamInit * incInitCam, QString incVideoFile)
188 {
189   // NOTE: Call this only once with P0 from CamInit!
190   chainMaker->setP0Mat(incInitCam->getProjection());
191
192   initCaptureParams(incVideoFile);
193   startMatcherLoop(incInitCam->getUserClicks());
194 //   startLiveFeedLoop();
195
196   // Compute Homography (Various techniques - LIN1, LIN2)
197 }
```

A.5 CAMFPLAN CamPChainMaker

```
1  #ifndef _CAMPCHAINMAKER_H_
2  #define _CAMPCHAINMAKER_H_
3
4  #include <cv.h>
5  #include <cvaux.h>
6  #include <cxcore.h>
7  #include <highgui.h>
8  #include <QDebug>
9  #include <vector>
10
11 #include "helper.hpp"
12
13 using namespace std;
14
15 class CamPChainMaker
16 {
17 public:
18         CamPChainMaker();
19         virtual ~CamPChainMaker();
20
21         // Init function
22         void init(CvMat * incPastMatches, CvMat * incPresentMatches,
              int incKeyFrame);
23
24   // Get projection matrix at index
25   CvMat * getProjMat(int index){ return projectionChain.at(index); }
26
27   // Set the first frame projection matrix P0
```

```
28    // Call only once! Have a singleton something here later
29    void setP0Mat(CvMat * incProjection0)
30    {
31      projectionChain.push_back(incProjection0);
32    }
33  private:
34    vector<CvMat *> projectionChain;
35  };
36
37  #endif // _CAMPCHAINMAKER_HPP_
```

```
1   #include "CamPChainMaker.hpp"
2
3   CamPChainMaker::CamPChainMaker()
4   {
5
6   }
7
8   CamPChainMaker::~CamPChainMaker()
9   {
10
11  }
12
13  void CamPChainMaker::init(CvMat * incPastMatches, CvMat *
      incPresentMatches, int incKeyFrame)
14  {
15    CvMat * tmpHomography = cvCreateMat(3, 3, CV_64FC1); // Allocate
        memory each time
16    CvMat * tmpProjection = cvCreateMat(3, 4, CV_64FC1); // Allocate
        memory each time
17
18    cvFindHomography(incPastMatches, incPresentMatches, tmpHomography,
        CV_RANSAC, 1); // LMEDS or RANSAC or 0?
19
20    cvMatMul(tmpHomography, projectionChain.at(incKeyFrame),
        tmpProjection);
21    projectionChain.push_back(tmpProjection);
22  }
```

A.6 CAMFPLAN CamVisualizer

```
1   #ifndef _CAMVISUALIZER_H_
2   #define _CAMVISUALIZER_H_
3
```

```
4  #include <cv.h>
5  #include <iostream>
6  #include <QDebug>
7  #include <vector>
8
9  #include "CamPChainMaker.hpp"
10
11 class CamVisualizer
12 {
13 public:
14         CamVisualizer();
15         virtual ~CamVisualizer();
16
17         // Init function
18         bool init(CvMat * incProjMat, IplImage * incPrintFrame);
19
20 private:
21   CvMat * tmpPt1[4];
22   CvMat * tmpPt2[4];
23   CvMat * tmpPt3[4];
24   CvMat * tmpPt4[4];
25
26   CvMat * calibPlane;
27
28   IplImage * matchedFrames;
29 };
30
31 #endif // _CAMVISUALIZER_HPP_
```

```
1  #include "CamVisualizer.hpp"
2
3  CamVisualizer::CamVisualizer()
4  {
5    calibPlane = cvCreateMat(4, 2, CV_64FC1);
6    // Assuming square of side 1 for now
7    cvmSet(calibPlane, 0, 0, 0.0);
8    cvmSet(calibPlane, 0, 1, 0.0);
9    cvmSet(calibPlane, 1, 0, 1.0);
10   cvmSet(calibPlane, 1, 1, 0.0);
11   cvmSet(calibPlane, 2, 0, 1.0);
12   cvmSet(calibPlane, 2, 1, 1.0);
13   cvmSet(calibPlane, 3, 0, 0.0);
14   cvmSet(calibPlane, 3, 1, 1.0);
15
16   cvNamedWindow("Tracking", 1);
17   cvMoveWindow("Tracking", 400, 200);
```

```
18 }
19
20 CamVisualizer::~CamVisualizer()
21 {
22   cvDestroyWindow("Tracking");
23 }
24
25 bool CamVisualizer::init(CvMat * incProjMat, IplImage * incPrintFrame)
26 {
27     for(int j = 0; j < 4; j++)
28     {
29       tmpPt1[j] = cvCreateMat(4, 1, CV_64FC1);
30       tmpPt2[j] = cvCreateMat(3, 1, CV_64FC1);
31       cvmSet(tmpPt1[j], 0, 0, cvmGet(calibPlane, j, 0));
32       cvmSet(tmpPt1[j], 1, 0, cvmGet(calibPlane, j, 1));
33       cvmSet(tmpPt1[j], 2, 0, 0.0);
34       cvmSet(tmpPt1[j], 3, 0, 1.0);
35
36       tmpPt3[j] = cvCreateMat(3, 1, CV_64FC1);
37       cvMatMul(incProjMat, tmpPt1[j], tmpPt3[j]);
38       normalizeVector(tmpPt3[j]);
39     }
40
41     for(int j = 0; j < 3; j++)
42     {
43       cvLine(incPrintFrame, cvPoint(cvmGet(tmpPt3[j], 0, 0), cvmGet(
             tmpPt3[j], 1,
44       0)), cvPoint(cvmGet(tmpPt3[j + 1], 0, 0), cvmGet(tmpPt3[j + 1],
             1, 0)),
45       getColour(1), 2);
46     }
47     cvLine(incPrintFrame, cvPoint(cvmGet(tmpPt3[3], 0, 0), cvmGet(
          tmpPt3[3], 1,
48     0)), cvPoint(cvmGet(tmpPt3[0], 0, 0), cvmGet(tmpPt3[0], 1, 0)),
49     getColour(1), 2);
50
51     cvShowImage("Tracking", incPrintFrame);
52     int key = cvWaitKey(3);
53     if(char(key) == 'q')
54       return true;
55
56   return false;
57 }
```

Bibliography

Aron, M., G. Simon, and M. O. Berger (2007). Use of inertial sensors to support video tracking. *Computer Animation and Virtual Worlds 18*(1), 57–68.

Azuma, R. T. (1997). A Survey of Augmented Reality. *Presence-Teleoperators and Virtual Environments 6*(4), 355–385.

Bay, H., T. Tuytelaars, and L. Van Gool (2006). SURF: Speeded Up Robust Features. *Computer Vision–ECCV 2006 3951/2006*, 404–417.

Behzadan, A. H. (2008). Arviscope: Georeferenced visualization of dynamic construction processes in three-dimensional outdoor augmented reality.

Behzadan, A. H. and V. R. Kamat (2007). Georeferenced registration of construction graphics in mobile outdoor augmented reality. *Journal of computing in civil engineering 21*, 247.

Behzadan, A. H. and V. R. Kamat (2011). Integrated Information Modeling and Visual Simulation of Engineering Operations using Dynamic Augmented Reality Scene Graphs.

Caudell, T. P. and D. W. Mizell (1992). Augmented reality: An application of heads-up display technology to manual manufacturing processes. *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on 2*, 659–669.

Fei Dai, S. M., M. Lu, V. R. Kamat, et al. (2011). Analytical approach to augmenting site photos

with 3d graphics of underground infrastructure in construction engineering applications. *Journal of Computing in Civil Engineering 25*, 66.

Fitzgibbon, A. and A. Zisserman (2003). Automatic Camera Tracking. *Video Registration*, 18–35.

Harris, C. and M. Stephens (1988). A Combined Corner and Edge Detector. *Alvey Vision Conference 15*, 50.

Hartley, R. and A. Zisserman (2003). *Multiple View Geometry in Computer Vision*. Cambridge University Press.

Horn, B. K. P. and B. G. Schunck (1981). Determining Optical Flow. *Artificial Intelligence 17*(1-3), 185–203.

Kamat, V. R. (2003). *VITASCOPE: Extensible and scalable 3D visualization of simulated construction operations*. Ph. D. thesis, University Libraries, Virginia Polytechnic Institute and State University.

Kato, H. and M. Billinghurst (1999). Marker tracking and hmd calibration for a video-based augmented reality conferencing system. pp. 85.

Li, M. (2001). Correspondence analysis between the image formation pipelines of graphics and vision. pp. 187–192.

Lucas, B. D., T. Kanade, et al. (1981). An Iterative Image Registration Technique with an Application to Stereo Vision. *International Joint Conference on Artificial Intelligence 3*, 674–679.

Milgram, P. and F. Kishino (1994). A Taxonomy of Mixed Reality Visual Displays. *IEICE Transactions on Information and Systems E series D 77*, 1321–1321.

Nitzberg, M. and T. Shiota (1992). Nonlinear image filtering with edge and corner enhancement. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 826–833.

Ohnishi, N. and A. Imiya (2006). Dominant Plane Detection from Optical Flow for Robot Navigation. *Pattern Recognition Letters 27*(9), 1009–1021.

OpenCV (2011, April). OpenCV - an open source computer vision library. http://opencv.willowgarage.com/wiki/.

Qualcomm Inc. (2011, April). Qualcomm AR SDK. http://developer.qualcomm.com/dev/augmented-reality.

Simon, G. and M. O. Berger (2002). Pose Estimation for Planar Structures. *IEEE Computer Graphics and Applications*, 46–53.

Simon, G., A. Fitzgibbon, and A. Zisserman (2000). Markerless Tracking using Planar Structures in the Scene. *Proc. International Symposium on Augmented Reality*, 120–128.

Tsai, R. (1987). A Versatile Camera Calibration Technique for High-accuracy 3d Machine Vision Metrology using Off-the-shelf TV Cameras and Lenses. *Robotics and Automation, IEEE Journal of 3*(4), 323–344.

University of Middlebury (2011, April). The middlebury computer vision pages - A repository for computer vision evaluations and datasets. http://vision.middlebury.edu/.

University of Southampton (2011, April). Pascal2 - Standardised databases for object recognition. http://pascallin.ecs.soton.ac.uk/challenges/VOC/.

Vallino, J. R. (1998). *Interactive Augmented Reality*. Ph. D. thesis, Citeseer, Rochester Institute of Technology.

Vicon Motion Systems, O. p. (2011, April). Boujou - Seamlessly add cg into your film or video footage. http://www.vicon.com/boujou/.

Wikimedia Commons (2011, April). Adapted milgram's continuum. http://en.wikipedia.org/wiki/File:Adapted_milgrams_VR-AR_continuum.png.

You, S., U. Neumann, and R. Azuma (2002). Orientation Tracking for Outdoor Augmented Reality Registration. *Computer Graphics and Applications, IEEE 19*(6), 36–42.