Aware Home Visual Perception (Part II): Impelementation and references

Zhonghao Yang
Graphics, Visualization, and Usability Center
Georgia Institute of Technology, Atlanta, GA, USA
yangzh@cc.gatech.edu

Abstract

This Technical Report is presented as a starting point of the User's Manual for Aware Home visual perception system. The potential audience will be anyone who want to understand the implementation details or want to learn the interface for building their own applications based on this system.

1 Introduction

This document serves as a starting point of the User's Manual for the Aware Home visual perception system. This Technical Report includes:

1. System architecture and module responsibilities;
2. System implementation and current status;
3. Explanation of various utility programs for the system;
4. System deployment, debugging and maintenance issues;

1.1 Automate documentation process

The practice of writing documentation with code is always strongly encouraged. Well-commented source can dramatically boost development efficiency and reduce system maintenance pain. Throughout this project we use the excellent Doxygen tool (www.doxygen.org) to automatically extract comments from source files, and reference manuals are available in both PDF and HTML formats for each modules. The compact PDF format is more suitable for off-line reading while HTML version is ideal for on-line browsing. Both versions provide essentially the same contents.

1.2 The audience

For this documentation, I assume the audience has significant working experience in object-oriented design and C/C++ programming, and basic knowledge for Windows operating system, especially about Microsoft COM/DCOM. Knowledge about linear algebra, image analysis, computational geometry, computer vision will be helpful but not strictly required.

The audience should at least have access to the following softwares:

1. Microsoft Visual Studio C++ 6.0 with the latest service pack, MSDN (Microsoft Developer's Network), optional latest Microsoft Core SDK;
2. Doxygen: good documentation generation and management tool;
3. WinCVS: Windows front-end for CVS (Concurrent Version System);
4. OpenCV (Open-source Computer Vision library), available at `sourceforge`;

5. STLFilt: Open source tools to filter lengthy and unnecessary STL messages, available at `http://www.bdsoft.com`;
6. any text editors, like UltraEdit from `idmcomp`);
7. any image viewers, we recommend IrfanView;

2 System Overview

As explained in my previous Technical Report, with distributed cameras and computers, we want the Aware Home to have the visual perception capability across the house. An ideal visual perception system will include the following feature:

- Reliably tracking targets and maintain their identities across multiple rooms throughout their lifetime, under reasonable wide range of illuminations and other environment noise;
- Function in real-time with distributed computing environment. The tracking system should not be subject to specific camera deployments and machine configurations;
- High degree of automation and limited manual intervention during extended period of time. The service is expected to be running 24-by-7.

This complicated perception task is achieved with the careful combination of both hardware and software pieces. The design details for various algorithms has been described in my previous Technical Report and this document is intend to focus on system engineering details. The intended audience should be anyone who want to further develop the system or anyone who want to build their own applications on top of this system.

2.1 Hardware

Cameras: Currently the 2700-square-feet Aware Home has 27 fixed cameras (MB-1850U, from http://www.polarisusa.com/) mounted in ceiling of both floors. These cameras are analog micro-board cameras equipped with plastic wide-angle lens. According to our experience, AGC (Automatic Gain Control) causes some troubles during dramatic illumination change, and we manually turned this feature off. Please refer to the camera's manual for instructions.

There are also a few PTZ cameras scattered around the house, we are developing experimental control module for them.

Cables: BNC cables are used to connect cameras with video capture card in the basement, we did not use any video amplifiers in between and the video signal seems fine. Sometime we also need some extra RCA-to-BNC adaptors.

Framegrabbers: Each machine is equipped with two video capture cards (Osprey 100), with each card connecting to only one video camera. We deliberately did not push too hard on hardware resource. Further engineering might consider switching between video channels to drive down the overall hardware cost.

Machines: 8 DELL machines are allocated to control/process 14 cameras on the first floor (`vision1` to `vision8`), while another 8 newer DELL machines for another 13 cameras on the second floor (`vsensor1` to `vsensor8`). All 16 machines are located in the basement for a centralized development environment, and currently `vsensor8` is used as primary development machines with all latest source codes, documentations, etc.

2.2 Software

All machines run Windows 2000 system and constitutes a distributed computing setting, suggesting the necessity of managing resources across process and machine boundaries. We have thus established a middleware layer to enable module communication and management across processes and machines based on Microsoft's COM/DCOM technology. Of course there are other alternatives such as CORBA and Java RMI in terms of middleware package. However, the COM/DCOM libraries and executables are readily available for all Windows platform, requiring less learning and deployment efforts, and our project has the right scale for it. CORBA is still undergoing standard evolution and requires greater learning efforts. Java RMI is not suitable for performance-sensitive video applications.

Throughout the project development, we maintain clear separation between the middleware layer and algorithm codes. The middleware layer is ultimately system-specific, while the perception algorithms are written in ANSI C/C++ to ensure platform-independent. This is to ensure system's portability, independent on the middleware implementation.

The system makes heavy use of OpenCV library, available for both Windows and Linux platform. This library provides many useful routines for computer vision research and development.

Currently we use `vsensor8` as our major development machine, and it has the latest source codes as well as documentations. All system codes (note only codes, not documentations) are achieved in CVS (Concurrent Version System) on `jetson.cc.gatech.edu` with module name of `Track8`. Also the latest codes, documentations, and related materials are available in CD-ROM media.

2.2.1 Further information about COM/DCOM

Throughout the system, we use Microsoft COM/DCOM as underlying protocol to enable distributed computing and resource management, all developers need to have at least a basic understanding of COM/DCOM principles and programming techniques. Search MSDN using keyword `Using COM` for some introductions.

COM (Component Object Model) is a way to define common programming interfaces so that different piece of code can work under pre-defined interfaces. This feature is quite helpful and make it possible to separate the design and implementation phase for different time and parties. DCOM (COM on a wire) takes one more step by expanding COM to support communication and invocation among objects on different computers (Search MSDN using `DCOM Technical Overview`). Ideally any piece of code that is written in DCOM paradigm can interact with each other across machine and process boundaries.

Please review basics about COM/DCOM before going on. For serious system development, the audience are required to develop more in-depth knowledge if they do not.

3 System details

All names for variables, functions, structures or classes are represented by `verbatim`;

`$TRACKER_SYS$` is the root directory for the visual tracking system. All directory is relative to this root.

Source codes resides in their own module directories, and `doc` directory has documentation in both PDF and HTML format, where detailed explanations on classes, structures, and algorithms are presented. System/application developers should refer to those manuals for coding-related issues.

In order to keep the system flexible and portable, we keep the implementation for algorithm and middleware separate. Here is a list of our modules in the system together with their brief descriptions:

- `primitive_lib`: provides the basic primitives used for image/video processing, computer vision and other system engineering. Basic image structure, time stamp, common interfaces for algorithm can be comfortably fit here;
- `visionGen_lib`: provides platform-independent descriptions for various computer vision algorithms. Later libraries can further subclass them for specific purpose;
- `visionWin_lib`: contains Windows-specific variants for all algorithms based on `visionGen` library;
- `dcomRemote`: contains the middleware wrapper for COM/DCOM. Specifically, we designed client-side agents to store states and manage the communication with remote servers, so that user application's coding responsibility is greatly reduced. In the future, this library can possibly be replaced by better middleware implementation without sacrificing portability;
- `dcomCapture`: a standalone DCOM server to provide video capturing capabilities from regular camera, AVI files and possibly PTZ cameras. This module is designed to deliver flexible video streams to any clients (applications / other DCOM modules);
- `dcomBSensor`: a standalone DCOM server to provide background subtraction for any incoming video streams. This module is designed to transform raw video stream into blob representations for any clients (applications / other DCOM modules). The visual information is further condensed;
- `dcomTracker`: a standalone DCOM server to provide clique-level visual fusing and tracking capacity. This module is designed to convert blob representations from multiple synchronized `dcomBSensor`s to local target trajectories;

- `dcomClique`: the experimental `cliqueManager`, another standalone DCOM server for managing global targets. This module is designed to provide global target labels and trajectories for query and archiving. Currently there is no stable implementation for this, and please refer to our previous Technical Report for some algorithm ideas;
- `dcomConfigServer`: a standalone DCOM server is designed to provide system-wide configuration service, any user applications or DCOM servers can query specific `key = value` pair as specified in configuration file;
- `visionDebug`: a Windows-based frontend to interact with the underlying system for debugging purpose. For example, user can initiate multiple remote cameras, etc.

For DCOM servers, the readers can also refer to Fig. 1.

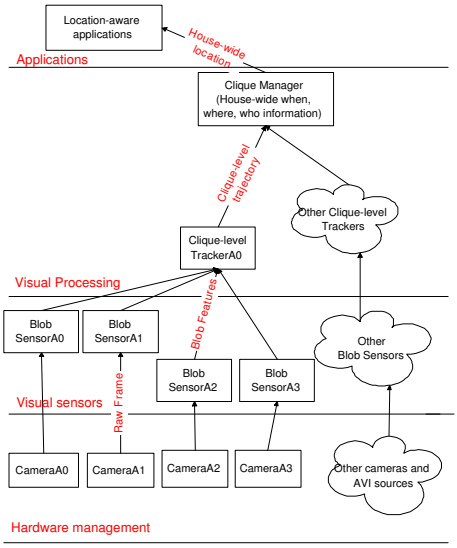

Figure 1: system schema

3.1 System configuration and configServer

To offer maximum flexibility in configuring distributed hardware and software pieces, we use text-based configuration file. The configuration file is organized as standard Windows .INI format, with only two-level hierarchies for ease of reading and parsing. The variable `defaultLocalConfigFileName` in the file `dcomRemote/dcomEntities.h` stores the location for the system's configuration file (one of the few hard-coded location throughout the system).

A standalone DCOM server (configServer) will load the configuration file `visionConfig.ini` in a pre-defined location and serve configuration queries for all other DCOM modules and applications. The query operations can be performed independent of machine and process boundaries, thanks to the COM/DCOM-based communication semantics. In fact the configuration server is automatically started once there is a client who potentially need configuration information in the system, refer to the functions `startLocalConfigAgent` and `stopLocalConfigAgent` in `dcomRemote` library: a reference count mechanism is implicitly used.

All system-level entities (including hardware and software pieces) are organized into categories in the configuration file, namely the file `visionConfig.ini`:

1. `cameraModel`: the camera model describes logically how camera works, including intrinsic, extrinsic parameters and camera's pose with respect to the world. This entity also contains distortion parameter for possible image unwarping;
2. `camera`: the camera entity controls the camera hardware by specifying the camera's physical property, such as host machine, and device identification, etc. Other hardware-related properties can also be added here later. Cameras on the first floor is identified by a letter `A` (such as `cameraA0`), while cameras on the second floor is `B` (such as `cameraB0`). It's one variant of (video) stream sources, managed by `dcomCapture`;

3. `aviFile`: the video file in .avi format. Each instance of this entities correspondents to an .avi file that resides on a specified machine. As one variant of (video) stream source managed by `dcomCapture`, it can be used to feed video for `blobBSensor`, for example;

4. `imageSeq`: the image sequence as one variant of (video) stream sources, managed by `dcomCapture`. Each instance of this category can be used as video source for providing frames;

5. `blobSensor`: this is the DCOM module that is in charge of local visual sensing for blobs. Internally, each instance use background substraction algorithm to process frames from any stream sources (camera/aviFile/imageSeq, etc) and generates blobs corresponding to moving pixels;

6. `clique`: this category contains geometry (size, boundary) and other information about each clique (room). A list of `blobBSensor`s within the physical coverage of this clique is also specified, used as part of the scene modeling for the visual fusing algorithm;

7. `blobTracker`: this category describes software piece for the DCOM server, that is in charge of clique-level visual tracking. Internally each instance will perform visual integration, motion correspondence, and compute clique-level trajectories of targets in one clique;

8. `cliqueManager`: this category describes software piece of one DCOM server, in charge of house-wide visual perception service by considering labeling transfer between cliques. This information exchange is necessary to ensure proper target identity handover between cliques for a consistent house-wide visual tracking task. Actually there is only one active instance for the whole house.

I'll provide more detailed information on each category in the following subsections.

3.2 cameraModel

In order for a visual tracking system to work in real environment, we first need to know something about the cameras and the environment. As an essential part for such prior knowledge, camera modeling includes the determination of both intrinsic and extrinsic parameters (with respect to the global world). We achieve this task using an utility program based on the OpenCV library.

We will first introduce some mathematical background for the camera calibration, followed by explanations on how to use the utility program to calibrate camera. Finally some discussions and possible improvements are provided.

3.2.1 Mathematical background

The intrinsic matrix for any camera is as following:

$$\begin{bmatrix} f_x & 0 & u \\ & f_y & v \\ & & 1 \end{bmatrix} \tag{1}$$

f_x and f_y are the focal length along the image X and Y axis measured in pixels. For most practical cameras, they are slightly different. $(u, v)^T$ is the image coordinate where the optical axis intersects with the image plane, for most practical camera this will be slightly off the frame center. We assume pixels are perfectly rectangular.

For intrinsics, we use the proposed method in OpenCV library. The chessboard pattern of original size (8-by-11 inch) is available at `misc/calibrationPattern`. Note the distortion parameter definition is slightly different from another popular calibration package (Camera calibration toolbox for Matlab from Jean-Yves Bouguet): the first 4 doubles are of the identical meaning and can be used inter-changeably.

For pinhole cameras, the projective projection is modeled as:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \sim \begin{bmatrix} f_x & 0 & u \\ & f_y & v \\ & & 1 \end{bmatrix} \cdot \begin{bmatrix} P_{3\times 3} & t_{3\times 1} \end{bmatrix} \cdot \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \\ 1 \end{bmatrix} \tag{2}$$

where $P_{3\times3}$ and $t_{3\times1}$ are the rotation matrix and translation vector for the world system in the camera system, and $(X_0, Y_0, Z_0, 1)^T$ is the homogenious coordinate for any 3D point in world system. Later we will define in details all related systems.

Theoretically if we can accumulate a number of frames with all image features detected (48 inner corners for our chessboard pattern per frame), epi-polar constraints can be exploited to find optimal solution for intrinsics and distortion parameters in terms of squared error, using non-linear optimization techniques. Search online by `camera calibration` for technical details. The camera calibration toolbox for Matlab from Jean-Yves Bouguet has very good materials to start with.

It's a little bit complicated to determine extrinsics: 1) it is not fully automatic and require some manual intervention; 2) it concerns several coordinate systems that we need to define here:

- camera system S_2: S_2 is defined with X, Y axes aligned with image plane's X, Y axes, and Z axis pointing out. The origin coincide with the principal point. Intrinsic matrix is further introduced to account for the camera's own geometry;
- intermediate world system S_1: the origin sits on *one of the inner corners* as shown in Fig. 2. The Y axis should always be along the longer edge of the pattern, while X axis takes the shorter edge, due to the coding convention in OpenCV implementation. Under this definition, note that 1) the origin is *NOT* the corner of the whole chessboard pattern; 2) the X axis is not necessarily horizontal in captured image, and Y axis is not necessarily vertical; 3) given four inner corners (the green corners as in the Fig. 2), the intermediate world system is uniquely defined;
- global world system S_0: each floor of the house is modeled identically but independently, with origin located at the southeast inner corner of the house. i.e., the southeast corner of the living room. With world X axis aligned with NORTH wall, and Y axis WEST wall, a natural right-handed world coordinate system is defined with Z axis up. All measurements for this system are in centimeters. The floor plan is available in `misc/floormap`.

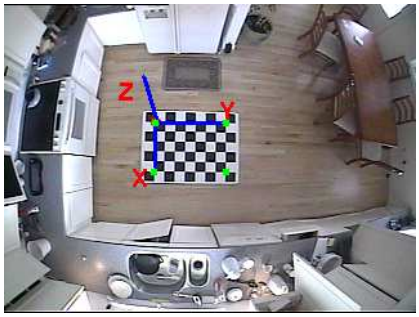

Figure 2: Intermediate system S_1

Typically we use a 4x chessboard pattern (32-by-44 inch) for extrinsic calibration. Essentially, given correspondence of all 48 inner corners for their image and world coordinates, the OpenCV function of `cvFindExtrinsicCameraParams` (or its double variant) is able to compute the rotation and translation of the world (implied by the *world coordinate* arguments for the function) in camera's system S_2.

There are two approaches to compute the global world S_0 with respect to camera system S_2:

1. Correspondence is given and computed between S_2 and S_1, manually measure and determine the matrix $T_{1\leftarrow0}$ (so that $X_1 = T_{1\leftarrow0} \cdot X_0$), finally $T_{2\leftarrow0} = T_{2\leftarrow1} \cdot T_{1\leftarrow0}$;
2. Correspondence is directly given between S_2 and S_0. In other words, the features' world coordinates are directly measured in S_0, $T_{2\leftarrow0}$ can be computed in one single step without additional matrix multiplication;

We use the first approach in all our utility program, however, the second approach turns out to be much easier and should be preferred in later implementation.

Even though we can calibrate individual camera to reasonable accuracy, there are still errors for projection of a common feature onto floor space by two adjacent cameras, resulted from individual camera's intrinsic and extrinsic errors. In our system, we are particularly concerned about the accuracy of projection on the floor by each cameras, and we introduce additional floor alignment process to further minimize projection error.

Here is our solution: for each camera, we manually label common image features (4 corners of a poster board) $(x_i, y_i), 0 \leq i < 4$, project them onto the floor $(X_i, Y_i, 0), 0 \leq i < 4$ with this camera's calibration. Tape-measure the ground truth $(X_i', Y_i', 0)$ for those corners in world system S_0. Fit the model

$$\begin{bmatrix} X_i' \\ Y_i' \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \cdot \begin{bmatrix} X_i \\ Y_i \end{bmatrix} (0 \leq i < 4) \tag{3}$$

using square error minimization. The alignment matrix $A = \{a_{ij}\}$ captures the variation in individual camera's calibration.

3.2.2 Utility for camera calibration

Available in `misc/calib`, the utility is a mini-project, with similar `CCameraModel` class and a driver class `CCaptureVFW`, which essentially perform video capturing using Video For Windows SDK. The whole project is moderately documented, and the audience should not have too much trouble understanding the idea and codes.

The calibration utility here is an extension based on the OpenCV's implementation. The fact that the cameras and the host machines are far apart requires some design work to ensure we can smoothly gather enough number of successfully detected patterns at different poses, even in uncontrolled illumination. Those frames will be used in the numerical optimization for camera calibration.

For intrinsics, the utility works as following: It continuously detects the desired pattern feature in every incoming frame (supplied by the driver procedure). Once it successfully detects all 48 inner corners in one frame, the program will give an audio feedback and accumulate the data. At the same time, the utility will freeze video capture for two second, allowing the operator to move the chessboard for next pose: similar pose will cause numerical instability and thus is not helpful in numerical optimization at all. During this continuous capturing and detecting, the operator is free to adjust illumination and pattern pose. This method turns out to be very effective, and the whole calibration process can be finished within minutes. It only requires two people, each with a telephone handset: operator 1 is required to monitor the utility running while operator 2 standing before the camera hardware with the chessboard. The telephone link has to be established so that operator 2 can hear the audio feedback.

We use a standard pattern of 8-by-11 inch for intrinsic calibration. The typical practice is to obtain 5 poses by rotating around the pattern's horizontal axis, and another 5 poses by rotating around another axis. 10 frames is enough for reliable intrinsic computation. The same process can be done multiple times for averaging/verification.

For extrinsic calibration, we need a 4X chessboard pattern (32 by 44 inch) to be placed in the camera's view. We use the first approach, which require the determination for intermediate system S_1 by selecting one of the 4 *inner corners* as origin (Refer to Fig. 2).

Start the utility and capture a single snapshot for the pattern on the floor. Under Aware Home's geometry, it's simply not possible for the native OpenCV corner detector to reliable detect corners and we have to do it manually. Exit the program, manually locate the image coordinates for the 4 *inner corners* (origin, X and Y direction, and the last corner) (using IrfanView or even Windows Painter). The function `CCameraModel::doManualCornerExtraction` will use linear interpolation as initial value and then refine the corner coordinates with the OpenCV function `cvFindCornerSubPix`. Given 48 correspondence between image coordinates and world coordinates, the homogenous matrix $T_{2\leftarrow 1}$ can be computed.

Tape-measure and determine $T_{1\leftarrow 0}$: the translation and rotation between S_1 and the global world S_0.

$T_{2\leftarrow 0} = T_{2\leftarrow 1} \cdot T_{1\leftarrow 0}$. Refer to `getCameraModel` in module `dcomRemote` for this computation.

This manual work is only required once for each cameras.

3.2.3 Current status

Currently all 27 fixed cameras have been calibrated. Intrinsic, extrinsic and distortion parameters can be found in `visionConfig.ini`

3.2.4 Discussions

In the future, we could be smarter by directly clicking on the captured image to streamline the extrinsic process. Bigger chessboard pattern or better and reliable corner detector will be also better solution, but we simply have not explored those possible

options. Jean-Yves Bouguet's camera calibration toolbox for Matlab also provides links for other improved solutions.

3.2.5 Properties in configuration

category	the category for this group, always `cameraModel`
resourceID	the resource identification within this category for this instance
rotationZ	the rotation between S_0 and S_1, used to compute $T_{1\leftarrow 0}$
cameraOrigin	the origin of S_1 in global world S_0
focus	the camera focus in pixel
principal	the camera's principal point in pixel
undistort	the camera's distortion parameter
extTransVec	the camera's translation vector in mm
extRotVec	the camera's rotation vector
extRotMat	the camera's rotation matrix
alignment	the alignment coefficient

3.3 cameras

This category represents the physical concept of a camera, or the video-capturing device, used by the DCOM module `dcomCapture`. All properties are quite self-explanatory.

category	the category for this group, always `camera`
resourceID	the resource identification within this category for this instance
hostMachine	the host machine for this instance
deviceID	the device ID for the video capture card for this instance
cameraModel	the camera model associated with this instance

3.4 avi files

This category represents the video file, used by the DCOM module `dcomCapture`.

category	the category for this group, always `aviFile`
resourceID	the resource identification within this category for this instance
hostMachine	the host machine for this instance
aviFileName	the local file directory and name for this instance
undistortNeeded	a flag indicating whether additional unwarping is needed
startPosition	the start frame number when requesting frames
cameraModel	the camera model associated with this instance, possibly used for unwarping

3.5 image sequences

This category represents image sequence, used by the DCOM module `dcomCapture`.

category	the category for this group, always `imageSeq`
resourceID	the resource identification within this category for this instance
hostMachine	the host machine for this instance
baseFolderName	the local directory for this image sequence
baseSeqName	the local base name for this instance
suffix	the image type for this instance
startFrame	the start frame number when requesting frames
endFrame	the end frame number when request frames
cameraModel	the camera model associated with this instance, possibly used for unwarping

3.6 blobSensor

In essence, this category represents the DCOM module `dcomBSensor` for performing background substraction for arbitrary video streams. Typical one such instance is established on the same machine of the associated video stream source (camera/AVI/image sequence, etc) to avoid transmission overhead. However, this is not a hard constraint and `blobSensor`s can be used to process frames from any machine, for example due to debugging purpose.

category	the name for this category, always `bSensor`
resourceID	the resource identification within this category for this instance
hostMachine	the host machine for this instance
source	the video source for providing frames for this instance, can be any instance in either `camera`, `aviFile` or `imageSeq` category
trainFrame	the number of frames used for background training;

3.7 clique

The cameras are grouped for each room (typically 4 or 6 cameras per room) based on their physical proximity. The camera group has special semantics because in our system we make full use of their significant overlapping for visual fusing. This category specifies what `blobSensor`s (in turn what cameras/AVIs) are responsible for providing blobs for each instance of clique.

category	the name for this category, always `clique`
camCount	the camera count for this instance of clique
contains	the list for blob sensors within this clique, separated by colon. This implicitly corresponds to the video sources within this instance of clique
gridDist	the distance between grids in centimeter, used to discretize the floor space for this instance
projectionHeight	the height of the projection plane for this instance
cliqueArea	the rectangle (x, y, width, height) for the instance of clique. Note not all grids need to belong to this clique, exclusion and unreliable area can be specified with respect to this rectangle, making the clique non-rectangular
cliqueAreaExclude	a list of rectangle that are excluded from this instance of clique
cliqueAreaUnreliable	a list of rectangle that are within this clique, but deemed unreliable. Examples are those areas near the wall or other furniture where it's unreliable to consider projection

3.8 blobTracker

Clique-level tracking task is accomplished in each DCOM module of `dcomTracker`.

category	the name for this category, always `bTracker`
resourceID	the resource identification within this category for this instance
hostMachine	the host machine for this instance
source	the associated clique for this instance

3.9 cliqueManager

Assuming there are some overlapping between cliques so that target locations can be used for transferring labels, the cliqueManager is designed to assign for each target a global targetID so that house-wide perception is possible. We have not fully implemented this feature in our system yet.

category	the name for this category, always `cliqueManager`
resourceID	the resource identification with this category for this instance
hostMachine	the host machine for this instance
contains	a list of cliques that this instance manages
writeToDatabase	a flag indicating whether the global target should be archived into database

4 Debugging application

To facilitate debugging for the whole system, we designed a MFC-based application which essentially served as a client to connect and verify each DCOM modules. Through this interface, the system/application developer can have better interaction with the underlying system and possibly have a deeper understanding of how the system works. I will describe this debugging GUI in detail in this sections.

The string array `g_resourceName` in the file `visionDebug.cpp` defines all available entities that can be accessed from this debugging GUI. Having detailed definitions in the global configuration file `visionconfig.ini`, these entities ranges from cameras (`cameraB0`, `cameraB12`, etc.), avi video files (`aviFileCalib6`, e.g.), image sequences, `blobSensor`s (`bSensorB0`, etc.), clique trackers (`bTrackerB0`, etc.) and clique manager (`cliqueManager0`).

4.1 Connecting to remote video sources

A new view for any remote camera can be created by selecting the item `File|New Camera View...`. If the remote camera has not been started, this will start the camera (the `dcomCapture` server, to be more exact), otherwise the running instance of DCOM server is connected. The new view will continuously display rectified frames (using associated camera model) transmitted from remote camera server (`dcomCapture`). The remote server is shut down when the last connection in the whole system has been closed. The local camera will follow the same semantics in terms of activating and accessing.

Refer to class `CDocCamera`, especially `CDocCamera::workerThread ()` for details on how to manipulate frames after retrieved at local ends.

A new view for any remote video file can be created by selecting the item `File|New AVI file View...`. If the remote DCOM server `dcomCapture` for this video file has not been started, this will start the server, otherwise the running instance is connected. The new view will continuously display frames from the video file across network. Rectification can also be performed for each frame on either end. The remote server is shut down when the last connection has been closed. The local video file will follow the same semantics in terms of activating and accessing.

Refer to class `CDocAVI`, especially `CDocAVI::workerThread ()` for details on how to manipulate frames after retrieved at local ends.

There is currently no similar manu item for image sequences. However, the implementation is quite straightforward.

The system is quite flexible that the unwarping can be performed either at video server side or at the client side.

4.2 Connecting to remote blobSensors

A new view for any remote `blobSensor`s can be created by selecting the item `File|New BSensor View...`. If the remote DCOM server `dcomBSensor` for the specified instance has not been started, this will start the server, otherwise the running server is connected. The new view will continuously display extracted blobs generated from the specified instance of `dcomBSensor` no matter it's remote or local. The server is shut down when the last connection has been closed.

Refer to class `CDocBlobSensor`, especially `CDocBlobSensor::workerThread ()` for details on data processing.

When the remote server of `dcomBSensor` is required to get started, the server will automatically start all underlying `dcomCapture` servers as defined in `visionConfig.ini`. Be aware this is an automatic chain reaction and in case of failure, consider the possibility of failure from `dcomBCapture`.

4.3 Connecting to remote clique visual trackers

A new view for any remote `blobSensor`s can be created by selecting the item `File|New BTracker View...`. If the remote DCOM server `dcomBTracker` for the specified instance has not been started, this will start the server, otherwise the running server is connected. The new view will continuously display computed local trajectories generated from the specified instance of `dcomBTracker` no matter it's remote or local. The server is shut down when the last connection has been closed.

Refer to class `CDocTracker`, especially `CDocTracker::workerThread ()` for details on data processing.

When the remote server of `dcomBTracker` is required to get started, the server will automatically start all underlying `dcomBSensor` servers as defined in `visionConfig.ini`. Be aware this is an automatic chain reaction.

Note: this interface has not been fully tested, a safer way to try this feature is `CVisionDebugApp::OnFileTracker`, where you can understand how blobs are extracted from synchronized cameras, and fused to provide reliable tracking within one clique.

4.4 Connecting to remote cliqueManager

This is still under development.

4.5 Synchronized video capturing

Our visual fusing algorithm needs the capability of synchronized video capturing. Ideally all concerned cameras (or other forms of video resources) needs to provide frames at the same time instance (synchronization) for accurate geometrical reasoning.

The dialog class `CDlgSyncCapture` is responsible for gathering user's input, performing video capturing and storing frames. Internally `CDlgSyncCapture::syncCaptureThread ()` uses class `CModuleSynchronizer` for all synchronization work.

4.6 More about configServer

The activation/shutdown of `dcomConfigServer` is transparent. The `dcomConfigServer` is only designed to reside at a pre-defined host (as defined in the file `dcomEntities.h`) at the proper time, the issue arising is that how can other process locate this only instance (most possibly across network). Our solution is to use moniker in COM/DCOM terminology. A moniker is a resource identifier that can uniquely locate a running instance (note: not a class of servers) in a networked environment. The authors are strongly encouraged to find more details about moniker in MSDN. The flip side of this approach is that we are somehow forced to live in Microsoft world.

Some details for this approach: after activating the `dcomConfigServer`, the system will write the moniker (resource identifier) to a network-accessible file (the file `mainCliqueManager.guide`) at a well-known location (the location has to be accessible by all server/client machines). Other applications/DCOM modules can access the file for the moniker. With the moniker, they are able to locate the unique instance of `dcomConfigServer` currently running in the system.

For details, refer to `startLocalConfigAgent` and `stopLocalConfigAgent` in `dcomRemote` library.

5 Deployment, debugging, and maintenance

5.1 System deployment

5.1.1 Microsoft DCOM support

Both Windows 2000 and Windows XP fully support COM/DCOM Technology. However, for serious development purpose, you need to make sure the underlying include files and libraries are up-to-date. First all latest service packs for both the Windows system and the Visual Studio have to be applied. On top of those patches, you also need to download the latest Microsoft SDK (core SDK) from Microsoft website, and direct Visual Studio to use the latest SDK instead of the old include files and libraries that ships with the original Studio CD-ROM. (Select `Tools->Options->Directories...` in Visual Studio).

5.1.2 deployment and updating

System deployment is an essential step towards making the system running over distributed resources. Currently all libraries (except the debugging GUI) is deployed in all vsensor machines (from `vsensor1` to `vsensor8`) since all those machines can possibly act as providers for some DCOM services.

We use `vsensor8` as the major development machine: the system will generate libraries and executables into local `lib` directory, which is also shared across network and can be accessed by synonym `\vsensor8\system`.

On the desktop of each vsensor machine, there is a .BAT icon named `Update VSENSOR`, which simply copy the latest binaries from the publishing directory `\vsensor8\system` to their local directory `c:\vtracking\bin` and perform necessary registerations. This manual updating should be performed every time when there is a change in code/interface in

vsensor8. Each machine should have this directory (c:\vtracking\bin) properly set in system global variable PATH so that Windows knows how to locate them.

Also currently c:\vtracking\bin\system of each machine from vsensor1 to vsensor8 has all required DLLs for running COM/DCOM and OpenCV package. These libraries do not required to update every time. Each machine should also have this directory properly set in global PATH variables so that Windows knows how to find these DLLs in case of need.

In the future, we need to be more careful about the system deployment issue, and especially we need to separate *Release Version* from *Debug Version*. While *Debug Version* is for system developers to further expand the core function for this system, the *Release Version* is for application developers who want to build their own applications on top of this system, and definitely they prefer a more stable system.

5.1.3 Accounts

A distributed system will inevitably run into security considerations. You can not simply let users to access any computing resources from an unauthorized location, and this is especially true for video cameras. We do not focus too much on security issues and primarily rely on Microsoft COM/DCOM's in-built security mechanisms.

In order for this distributed system to work, we manually established an account vision on each vsensor machine. All machines have to be actively logged exactly in account vision in order to provide/request services. COM/DCOM do provide mechanism for one machine to provide remote services with different accounts, which requires somehow embedded password in the function call. Interested system developers can explore that option in future. However, at this moment, we restrict the user to stick to vision account.

Due to security management, currently Windows requires periodic changing of password for every 3 months, and expired password do cause problems. Here is the symptom and solution:

Tip: In Microsoft COM/DCOM, HRESULT hr is used as return value by most functions to indicate the status of previous operation. The text description of this return value can be found by typing "hr,hr" in Visual Studio's Variable Window. It contains valuable information about COM/DCOM debugging and helps greatly to reduce debugging pain.

One of the most annoying error message for hr in our system development is E_ACCESSDENIED when you are trying to establish an instance or connect to a remote instance. While this error can have some other causes, mostly it's due to the expiration of user account "vision". Simply reset the password to default for the target machine will solve this issue.

5.1.4 Adding new machine

Here is the checklist for adding new machine into this system:

1. setting up system, and possibly Visual Studio for development work;
2. ensure network runs well, add the machine into network;
3. ensure this machine is able to access mainCliqueManager.guide on the network;
4. modify visionConfig.ini to include any hardware/software pieces that will run on this machine;
5. set directories to store binaries for possible DCOM server/clients. set script to update from publication directory (vsensor8\system);

5.2 Debugging

The COM/DCOM debugging is very complicated because it concerns with both servers and clients. Especially when client/server terminates/crashes for some reasons, the developer might have to manually shut down all related processes. One of the most frequent mistakes: when shutting down the GUI client, the developer often forgets to shut down all remote server processes and the configServer. This will cause the subsequent invocation of remote servers to fail.

Tip1: sometime it's hard to tell whether a crash is due to algorithm failure or COM/DCOM infrastructure. In this case, simply use CVisionDebugApp::test() to perform an isolated blackbox test and unit test. Since this is done locally without intervention from COM/DCOM issue, problems might be easily located.

Tip2: given KVM switch, it's sometime hard to debug distributed DCOM modules on different machines. One work-around is to change the visionConfig.ini to make both the server and client application running on the same machine.

Start one instance of Microsoft Visual Studio, load server and set breakpoint accordingly. Start another instance of Visual Studio on the same machine for the client and set breakpoint. (It's OK that class information might not be available for the second instance of Visual Studio due to sharing conflict). Running the server will put the server into waiting state, waiting inter-process invocation from clients, and the breakpoints in the server will be automatically reached when the client invoke the server.

5.3 System extensions

Refer to directory `summerProject` for codes, documentations on various system extensions. This work was done with a group of 4 undergraduate student during the summer of 2004.

5.3.1 PTZ camera wrapper

We performed initial investigation on how to integrating PTZ cameras into our vision system, primarily for active vision purpose. This requires the capability to communicate with camera control base and direct the camera to point to a specified location in global world system S_0. It differs significantly from traditional fixed cameras since in this case the communication is bi-directional. In the project CD-ROM, the project `summerProject/TestCameraServerGUI` for manipulating PTZ camera. Refer to separate documentation for details.

5.3.2 Database interface

We performed initial investigation on interfacing with Microsoft SQL Server, primarily for archiving trajectories. The original code is in `visionWin_lib`, and documentation is also available in directory `summerProject`.

5.3.3 Applications

Based on the idea presented in [2, 3], we performed experimentations on a number of applications.

5.4 Testing videos

We've captured a number of videos using either the debugging application or the video capture utility that comes with Osprey Video card/Windows system. Those videos can be used for performance test for various vision modules.

Since our visual fusing algorithm require synchronization (time stamp alignment), we did synchronized video capturing using the cameras in second floor kitchen (`cameraB4` to `cameraB7`). These videos include:

- `caseA_cameraB*` are the videos used for our WACV'05 paper [4]. Four cameras are used to capture 4 people enter/exit the kitchen randomly and walking around;
- `caseB_cameraB*` are captured by 4 kitchen cameras for WACV'05 paper [4], Only 1 person is walking around to simplify data association;
- In `cameraB*_sitting`, 4 people sits and walks around the kitchen;
- `cvpr_camB*` is the synchronized videos for stress test. The testing scenarios are described in section `Stress test` of previous Technical Report

Acknowledgments

The support of resources from Aware Home Research Initiative (AHRI) at Georgia Institute of Technology to make my work possible is gratefully acknowledged.

Appendix

Appendix A: Delivable CD-ROMs

Here is the structure of the CD-ROM media:

module	codes for individual modules
doc	documentations in both PDF and HTML format
lib	binaries and libraries for the system, directory for publication
publications	publications (conference papers and Technical Reports)
summerProject	experimental features and utilities
testVideos	videos used for system debugging
testVideos/polygons	the polygon output after visual fusing, one file per frame
videos	videos for `Finding lost object` application
misc/floormap	The floor map for Aware Home (jpeg and Microsoft Visio format)
misc/extrinsic	working images for extrinsic calibration
misc/alignment	working images for alignment
misc/calibratoinPattern	the chessboard pattern
misc/visionConfig.ini	a copy of the global configuration file for the whole system, specified in variable `defaultLocalConfigFileName`
misc/vidCap	Video capturing utility from MSDN
misc/o100NTsdk	SDK from Osprey on how to perform video capturing
misc/STLFilt	STLFilt software
misc/WinCVS120	WinCVS software
misc/vs6.chm	MSDN Samples
misc/calib	camera calibration utility
misc/Tortiglioni	work by previous students on Aware Home Visual Tracker, depreciated

Appendix B: Misc materials

Something that system developers might need to know:

1. password for account `vision` for each machine;
2. password for CVS located on `jetson.cc`;

For obvious reason, this information can only be found physically with the project CD-ROMs.

Together with the project CD-ROMs, there is a file folder, which contains a) working sheet for camera calibrations; b) tutorials for COM/DCOM, STL, ATL, etc.

For more information about this project, please contact: Zhonghao Yang (yangzh@gmail.com)

References

[1] Eric Martinson, Ronald C. Arkin, Noise maps for acoustically sensitive nativation, Proc. of SPIE, Vol. 5609, Oct. 2004.

[2] Rodney E. Peters, Richard Pak, Gregory D. Abowd, Arthur D. Fisk, Wendy A. Rogers, Finding lost objects: informing the design of ubiquitous computing services for the home, GIT-GVU-04-01.

[3] Truon, Khai. N., Abowd, Gregory D., Brotherton Jason A. Who, what, when, where, how: Design issues of capture and access applications, Proceedings of the Inernational conference: Ubiquitous computing (UbiComp 2001), Atlanta, Georgia, Sep, 2001, pp. 209-224.

[4] Zhonghao Yang, Aaron Bobick, Visual integration from multiple cameras, *IEEE workshop on applications for computer vision, 2005*