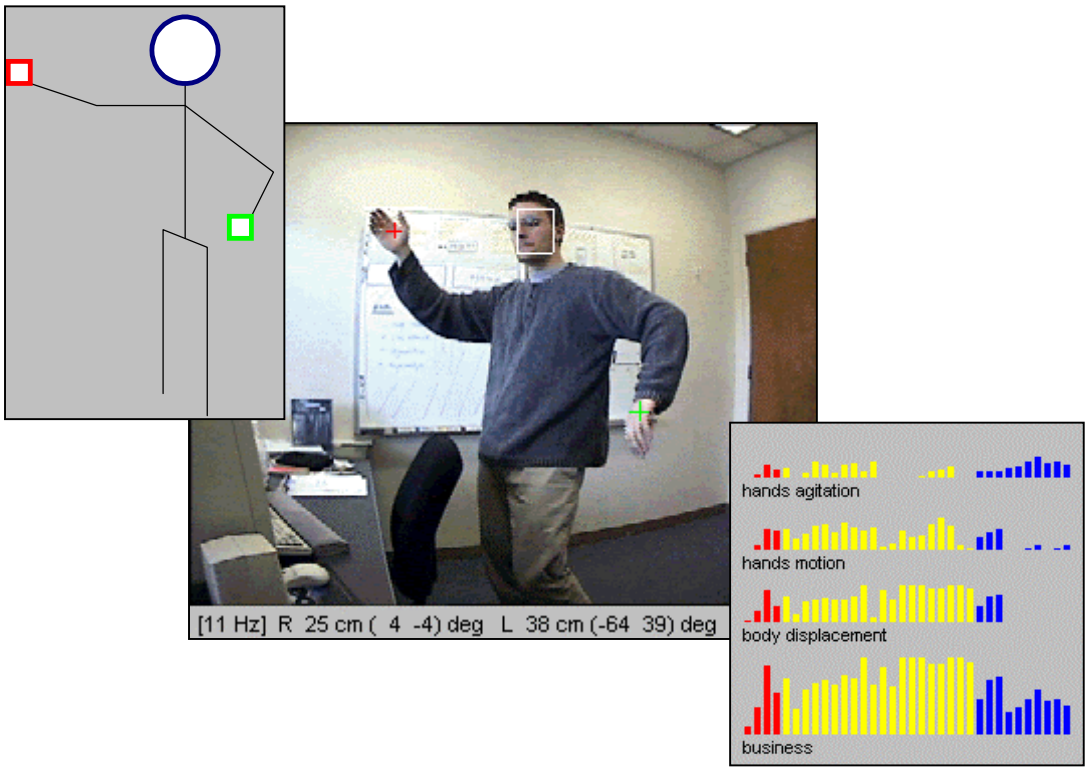
Diploma Thesis
academic year 1999-2000

Vision-based Sensor Fusion for Active Interfaces: H.O.T. – Human Oriented Tracking



Student	*Sébastien Grange*	
Supervisors	*Prof. Reymond Clavel*	EPFL
	Dr. Luc Julia	SRI
Assistants	*Dr. Charles Baur*	EPFL
	Terry Fong	EPFL
	Didier Guzzoni	SRI

Table of content

List of figures

List of tables

Abstract

Sensor fusion can be used to build a robust tracker and to analyze human activity for automatic input monitoring. The goal of this project is to develop new sensor fusion techniques using vision sensors (stereo cameras, panoramic cameras). These techniques will enable creation of active interfaces for intelligent environments, for example smart meeting rooms.

This project describes the implementation of a vision-based people tracker, which is used to create a geometric and dynamic model of a person. The tracker uses sensor fusion from two input modalities, namely color images and stereovision, to locate particular human features. A model of the human pose is then built from the tracking results, and human movements are segmented and parameterized. The model is finally used to monitor the person's activity.

The purpose of activity monitoring is primarily to allow for better human-computer interaction by giving the computer knowledge about the human. A benefit from activity modeling is to provide the human with appropriate feedback, at the right moment, and using the best available output modality.

The result is a software library, called H.O.T. (for Human Oriented Tracking), which can be used as a reusable interface between a human-computer interaction application and a human user interacting with the machine. H.O.T. is specifically designed for smart spaces applications, but can be used in many other fields involving human monitoring, such as mobile robotics and computer- assisted teleoperation.

1. Introduction

One of the new challenges in interface design for Human-Computer Interaction (HCI) is to make the interaction seemingly natural. Recent technology advances in computing hardware have lead to cheap, powerful computing devices that can be embedded virtually anywhere. Therefore, it is now possible to build intelligent environments that can sense the human presence and provide extended functionality.

One of the ways to enhance HCI applications is by giving the machine more knowledge about the human being it is interacting with, such as the task he is performing, his current center of interest, the mood he is in, … Most of this information is used by humans when communicating with one another, to regulate the interaction and to efficiently perform complex tasks. The purpose of this activity monitoring is primarily to allow for better human-computer interaction by giving the computer the means to perceive how busy the human is. A benefit from activity modeling is to provide the human with appropriate feedback, at the right moment, and using the best available output modality.

This project aims at creating a reusable software layer that could be used with any HCI application. This software layer can provide the application with information about the human's behavior, and lead to more meaningful interaction. It is specifically designed for Intelligent Environment applications, but can be used in many other fields involving human monitoring, such as mobile robotics and computer-assisted teleoperation.

The software library developed in this project, named H.O.T. (for Human Oriented Tracking), is implemented as three independent layers. First, a sensing layer is dedicated to vision-based tracking of human beings using sensor fusion from color and stereovision modalities. On top of this sensing layer, a geometric and dynamic model of the human pose and actions is built. Finally, information about the human's activity is extracted and integrated over time to perform meaningful activity monitoring.

This report is divided into several sections. Section 2 explains the context in which H.O.T. is meant to operate. Current research that is relevant to this project is described in section 3, with an emphasis on Intelligent Environments, while section 4 describes the H.O.T. concept. The following sections (5 and 6) give more insight on H.O.T. implementation, and present the current results. Finally, the future possible development of H.O.T. is discussed in section 7.

2. Context

2.1. Human computer interaction

The Human-Computer Interaction (HCI) domain is a vast and complex research field that is highly interdisciplinary. It involves knowledge ranging from computer science to psychology and ethnology, and has many potential applications:

- personal computers
  interaction with computers as we know them today is traditionally limited to the keyboard-mouse paradigm; however, researchers are trying to enhance the way we interact with desktop computers. As a result, speech recognition technology, smart pointing devices such as gaze-tracking vision-based systems, and force feedback mechanical systems are being developed to widen the interaction possibilities, and make them more intuitive and natural.
- ubiquitous computing
  technology is now allowing computing power to be available in small, flexible devices that can be invisible and integrated everywhere; as a result, desktop computers may soon become obsolete in many environments where smaller, distributed devices will take over. Smart clothes, smart vehicles, smart appliances are examples of this new technology. Interaction with these smart devices should not require such cumbersome means like keyboards. Therefore, new technology and concepts have to be developed to allow for natural communication between a human and his smart devices.
- smart spaces
  an extension of ubiquitous computing is smart spaces. Smart spaces, or Intelligent Environments, are places where many computing devices and interaction modalities are combined to support and assist humans in every-day tasks. Examples of smart spaces are smart house (see [4]), smart office, smart kitchen, smart car… Again, managing the interaction between an omnipresent sensing machine and a human is not straightforward, and concepts have to be developed to create a natural interaction between a Smart Space and a human user. The human should not feel overwhelmed by the computing force surrounding him, and the machine should still provide appropriate services without explicit commands through a control panel.
- military
  although not the most noble research field for human-computer interaction, the constant need armed forces show for technology is driving the research effort. New technologies are being engaged on the battlefield, such as robots and “smarter” communication systems. Interaction modality with the machines, along with the design of the command interfaces, has a crucial impact on the capacity of such devices to complete their mission. Interaction has to be both straightforward and accurate, while performed in the fast, discrete and explicit way that military operations require.

All these application fields for HCI (and the many more that were not mentioned above) often have to deal with input from several sources (e.g. “put that there”, a voice command and a human gesture, pointing to a certain object). Natural interaction, from a human perspective, requires multiple input sources to be processed together, compared, and used in a complementary or exclusive way. There is a lot of research about how to combine input from several sources to extract a meaningful command or information; its purpose is to develop *multimodal* interfaces that can interpret human behavior in a way that’s as close as possible to the way we humans understand each other. Multimodality is a very complex field with abundant literature, and is beyond the scope of this project. However, multimodality is part of the framework behind this project, which consists of using sensor fusion to monitor and interpret human actions.

A good introduction to multimodality, numerous resources and experiments, along with many references can be found in [14] and [6].

2.2. Project specifications

This project aims at using vision-based sensor fusion to create an active interface in the HCI context. It consists of three main parts:

- human tracking
  using vision-based sensor fusion, successfully detect and track a human being in a given volume of space
- human modeling
  while tracking a human being, extract a model of his pose and actions
- activity modeling
  using the pose and actions modeled, interpret human activity in a useful way

The idea is to create a reusable software layer (name H.O.T. for Human Oriented Tracking) that will act as a buffer between an HCI application and the external world, as illustrated in figure 2.1. The components highlighted in gray are the main components of H.O.T.

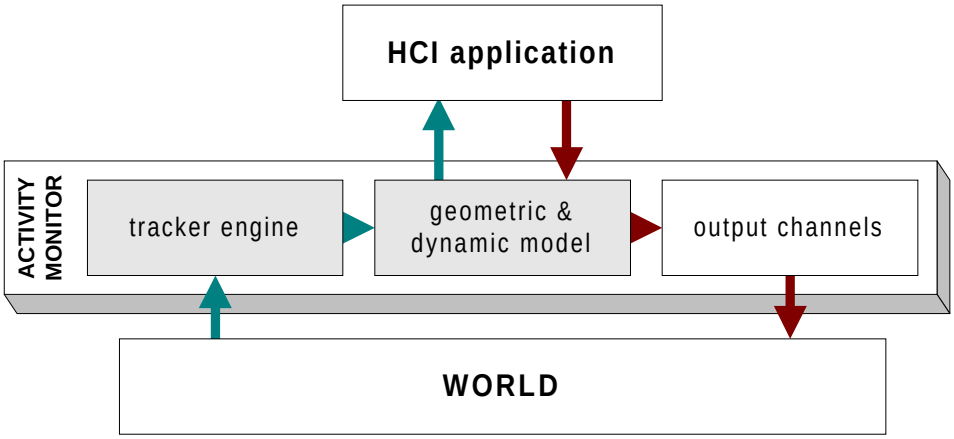


figure 2.1 – the H.O.T. concept

As figure 2.1 shows, H.O.T. will have two main functions:

- input processing
  the model built by the tracker is available to the HCI application
- output processing
  the HCI application uses the data collected by H.O.T. to deliver information to the user at the appropriate time and using the appropriate output devices

H.O.T. is typically intended to work in a smart space architecture. It is designed to be combined with another application that tracks people in a room, such as the one David Beymer is currently developing at SRI. Once a person is located in a sensitive area where he is likely to perform a specific task, H.O.T. takes over the people tracker and monitors the person's activity in more detail. This way, the visual tracking in a room is divided into two modes ("keep track" and "interact"), which seems to make sense in an Intelligent Environment where people moving around do not require the same amount of attention from the computer as a person trying to perform a specific task at a specific location.

Another possible application of H.O.T. is mobile platforms such as robots or teleoperated vehicles. H.O.T. can give these devices a perception of the human in front of them, as well as the basic tools to interpret the person's actions and activity.

2.2.1. What is activity

H.O.T. is designed to monitor people's activity. Activity is here taken in its broad meaning, namely a measure of how active a person is. H.O.T. is not designed to recognize a specific activity (i.e. sitting, walking, typing, eating…), but rather to characterize the current actions the person is undertaking, and integrating them over time. A typical result of H.O.T. activity monitoring is "the person has been moving around a lot for the past 2 minutes", or "the person was very still for the past 5 minutes", or even "the person was doing something delicate for ten minutes, then moved to this other location and is now moving objects around".

This way, H.O.T. monitors how busy the person seems to be through several visual parameters. These parameters can be used as an indirect measure of concentration or physical activity of the user. This monitoring allows making assumptions about the availability of the human for different output types (typically audio vs. visual) and at different times.

2.2.2. Why monitor activity

There are many applications that can take advantage of a human monitoring interface. HCI applications that aim at developing natural, intuitive interaction with a human being certainly require some level of visual interpretation of the person's activity (just think about how difficult it is to guess somebody's actions while blindfolded). The idea is to give computers some kind of "consciousness" of the user's behavior to allow for better interaction. Activity monitoring, as defined in this project, is very versatile, and its possible uses depend on the purpose of the application that requires it. Here are some typical application examples:

- output redirection
  in a Smart Space context, the most appropriate output channel the machine should access to provide the user with information can be derived from activity monitoring. For instance, if the user is physically busy, or located away from any display devices, an audio output is more appropriate than a visual output. On the other hand, if the user is on the phone or talking to somebody, visual feedback is preferable.

- output timing
  again, in a Smart Space context, HCI applications can use activity monitoring data to decide on the most appropriate moment to deliver a certain piece of information, based on the person's activity and the message priority. When the person is busy, only high priority messages should be delivered. For low priority messages, the application should wait until the person is less busy to deliver the information.

- command interfaces
  many devices can be controlled at least partially through visual inputs. All applications requiring pointing (such as white boards, map displays for teleoperation interfaces, …) or gesture tracking (virtual reality, …) can take advantage of activity monitoring. Not only does H.O.T. provide gesture segmentation and parameterization (see section 4), it can also be used (at some level) to determine if the person is trying to interact with a particular device, or just performing another activity. This information can be used to switch the device input control on or off, therefore avoiding clumsy artifacts that are common in vision and gesture-based control interfaces.

- augmented reality, virtual reality and games
  an extension of the command interface application is augmented reality and virtual reality systems (including games). A measure of the activity of the person can make a reactive virtual environment much more sensitive to the mood of the user; for instance, the application can determine if the user is bored, or extremely attentive to a certain part of the virtual world, and act in consequence by adjusting the intensity of the interaction, or changing its location.

2.2.3. Design hypothesis

Software design

Given the specifications described above, there are several technical design constraints to derive that will affect the design of the H.O.T. system:

- real-time
  obviously, since H.O.T. deals with both instantaneous activity measurement and fast gesture segmentation, it has to be a real-time application; the software will be optimized to minimize the cycle time of the vision processing
- single person tracking
  H.O.T. works in conjunction with another application that tracks several people at the same time and can provide H.O.T. with localization information; therefore, H.O.T. only needs to track one human being at a time
- person faces the camera
  as a collaborating people tracking application (or other reasonable assumptions relative to the performed task) can retrieve the user's orientation in space, H.O.T. can always use input from a camera approximately facing the user. The reasonable assumption that the human being faces the camera simplifies the geometric model.
- no background subtraction in tracking
  even though this commonly used visual tracking technique makes tracking very reliable, it is not applicable here since H.O.T. can be used on mobile platforms as well as in fixed places.
- unconstrained environment
  because H.O.T. is meant to be versatile and portable to a wide range of applications, it should be fairly environment insensitive. The only acceptable constraint is that the light conditions should not vary dramatically during H.O.T. operation.
- flexible
  H.O.T. should be flexible enough to be easily enhanced with more features at a later time (when more computing power is available, or a new tracking modality should be integrated to add a new feature to the human model)
- portable
  as much as possible, H.O.T. should not be platform specific, nor hardware specific, so that it can be used with different systems, and integrated easily into existing systems

3. Related research

There are a lot of projects that deal with human tracking and monitoring, whether directly focusing on the topic or including it in wider systems capability. Here is a selection of some relevant ones; the reader should keep in mind that the list of projects detailed below is far from exhaustive.

3.1. Smart spaces research

Smart spaces, also called intelligent environments, are spaces in which ubiquitous computing is used to enhance functionality. A smart space is therefore smart relatively to the purpose it serves: a smart office will offer support for office activities, and a smart kitchen will provide smart kitchen appliances.

As this project is intended for HCI applications, smart spaces are very likely to be one of the domains in which H.O.T.-like systems will be used. Smart spaces involve human tracking and monitoring at different levels, depending on spatial and temporal constraints. It is currently the object of a strong research effort, and is likely to be a part of everyday life in the near future – how important this part is going to be is a discussion way beyond the scope of this project. However, smart spaces oriented research faces several problems that directly address the specifications of this project.

3.1.1. MIT Intelligent Room

A famous example of smart space is the Intelligent Room developed at MIT by Michael Cohen [21]. The room is actually a smart conference room, and is equipped with an array of computer-controlled devices. These include steerable video cameras, VCRs, LCD projectors, lights, curtains, video/SVGA multiplexers, an audio stereo system and a scrollable LCD sign. The room can also generate infrared remote control signals to access consumer electronic items. A skeletal view of the conference area is shown in figure 3.1:

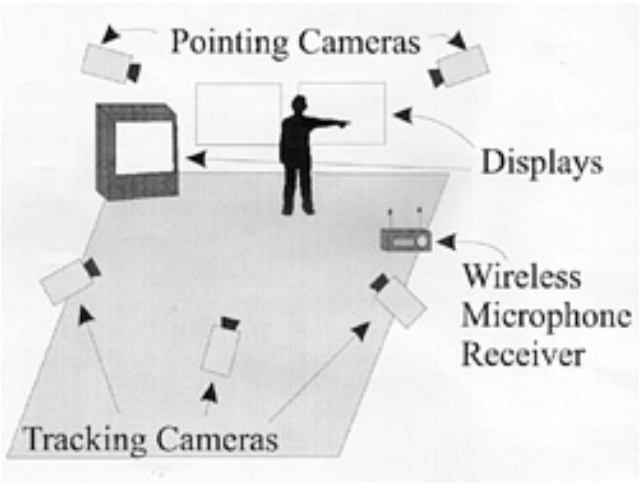


figure 3.1 – conference area in the M.I.T. Intelligent Room

The primary purpose of this project is to explore how Intelligent Environments should be designed. The concept of Intelligent Environment relies on newly developed technology that allows computing to be both user-friendly and invisible to the user. Therefore, computing can be ported to new areas.

A typical scenario that illustrates the Intelligent Room capability could be (from [21]):

> *The user walks into the room and lies down on the sofa after shutting the door. [the computer] sees this, dims the lights, closes the curtains, and then puts on Mozart softly in the background. [the computer] then asks "what time would you like to get up?"*

The room's intelligent system, called HAL (after the computer in the movie "2001: a space odyssey"), can interact with people through speech synthesis and the visual displays in the room. HAL uses speech recognition technology to get voice commands from the user, and keeps track of the people in the room by integrating input from several tracking cameras. An illustration of the tracking system is presented in figure 3.2.

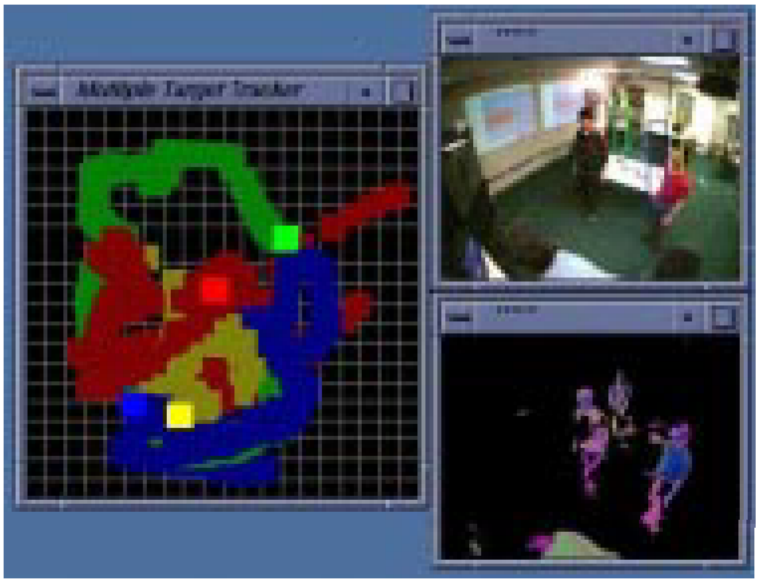


figure 3.2 – HAL people tracking debug interface

HAL tracking system locates and follows up to four people in the room. It also remembers their trajectories. However, as said in [21], although dynamic person tracking can seem important, the vast majority of the tracking system's output is never used. Few applications need real-time trajectory information at the room scale. A very important piece of information is rather the location of the person when he stops moving: close to a particular device, or on a piece of furniture.

HAL doesn't push visual analysis of the people in the room any further than location tracking. The only place where a person can visually interact with the room is one of the room walls. Two dedicated cameras track the person's finger if it comes closer than 4 inches to the surface of the wall (see figure 3.1). This feature is used as a pointing device to interact with maps displayed by the LCD projectors. This system is highly specific and limited to this single pointing application; moreover, it doesn't model the human pose.

H.O.T is the next step; once a person is located in a sensitive area, higher-level tracking can be performed that also focuses on the person's activity. Helpful and meaningful interaction can then be achieved, as the goal the person is trying to achieve is known to the computer – or at least limited to a couple of possibilities associated with that location in the room.

The Intelligent Room is a good example of the context within which H.O.T is designed to work.

3.1.2. Microsoft EasyLiving

Another research aiming at the creation of an Intelligent Environment is EasyLiving, presented in [5]. This ambitious research wants to provide a flexible, easy to deploy ubiquitous computing architecture to integrate in living spaces such as houses and offices.

The basic idea behind EasyLiving is shown in figure 3.3:

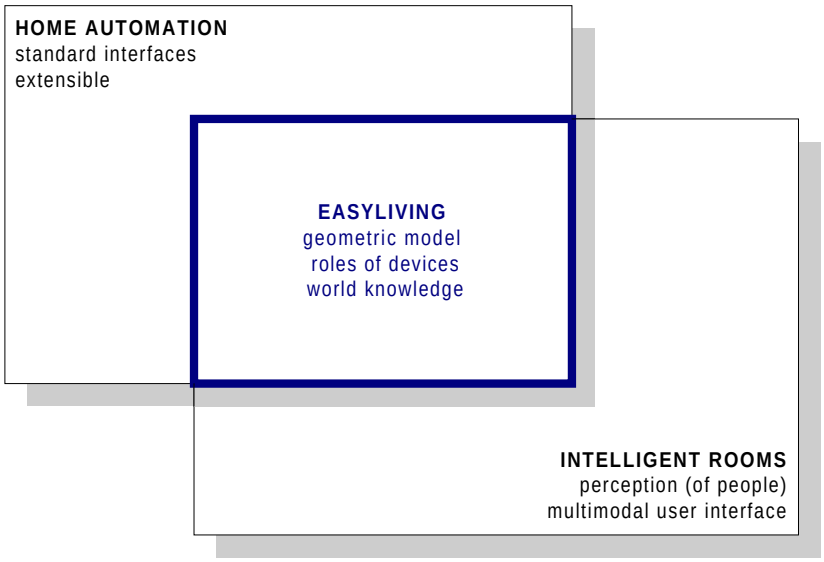


figure 3.3 – the Microsoft EasyLiving concept

The goal of EasyLiving is to provide casual access to computing. It is designed for every-day living spaces and uses a flexible distributed computing architecture to make it easy to expand and install in new environments with little effort (figure 3.4).

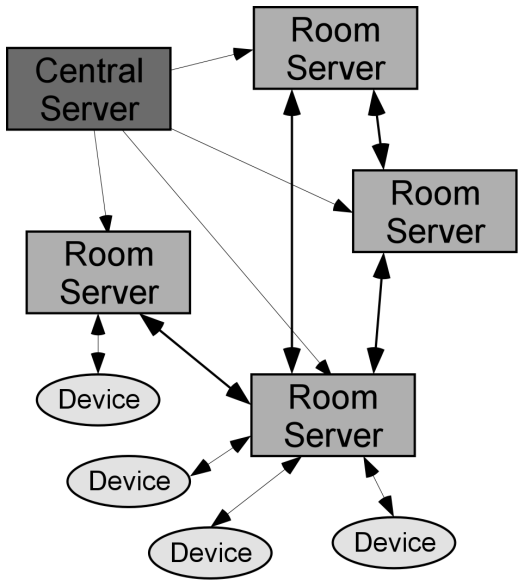


figure 3.4 – EasyLiving modular architecture

More specifically, the tracking system in the current EasyLiving demonstration room makes use of cameras combining 3 modalities (stereo vision, color vision and infrared vision) to detect and track people. Several such cameras are placed in different locations in the room. Using inclinometers mounted in each camera and image processing techniques, the system will be able to perform self-calibration of the stereo systems and retrieve each cameras position within the room without any human support. This feature makes the system very flexible and easy to extend.

The input from the cameras is used to build a geometric model of the world. This model is a high-level description of the objects and their relative position and orientation in the world that can be accessed by every application that needs it. New sensing modalities, such as active badges or pressure sensors, could be added to make the geometric model more accurate; these changes would be transparent to the applications, as these only query the geometric model and do not rely on a specific sensor output.

Figure 3.5 shows the geometric model of the demonstration room coupled with the people tracker. The gray dot represents the position of the person in the world.

figure 3.5 – EasyLiving people tracking debug interface

When entering the room, the people tracker automatically tracks a person. Using a fingerprint reader mounted in the wall, the human being can identify himself, thereby allowing the system to provide more specific services.

Currently, the system in the EasyLiving demonstration room performs simple tasks such as turning the TV on with the user's favorite settings when he sits on the sofa, or by playing the favorite music of an identified person.

Again, people tracking is reliable, but currently no further analysis of the person's activity is performed. For instance, the system is going to react in the same way (that is, by turning the TV on) whether a person sits on the sofa looking towards the TV or if he sits on the sofa and starts reading a book.

3.1.3. Kismet

Kismet is a robot that has been developed at MIT's Artificial Intelligence Lab to model a mother-infant interaction. The purpose of this experiment is to understand learning in a social context. Kismet ("fate" in Turkish) is an infant-robot, who responds to a caretaker's actions and intentions. This research is making use of concepts from psychology, ethnology, and developmental psychology, as well as the applications of these fields to robotics as outlined in [23]. Kismet is part of a larger project designed for social interactions with humans [24].

In a mother-infant relationship, the infant displays a variety of facial expressions and sounds that the mother interprets as clues about the internal state of her child; she then reacts in the appropriate way to ensure the well being and happiness of the infant. For example, when he appears content she tends to maintain the current level of interaction, but when he appears uninterested she intensifies or changes the interaction to try to re-engage him. This way, the infant can regulate the intensity of the interaction, and the mother can provide the best learning environment possible.

Kismet is a robotic platform that can display facial expressions. It is equipped with animated facial features such as eyes, eyebrows, eyelids, mouth, lips and ears. Kismet also owns a camera that allows him to keep track of the caretaker to evaluate the kind of interaction the caretaker is trying to provide and to evaluate the intensity of such interaction. Kismet's robotic platform is shown in figure 3.6.

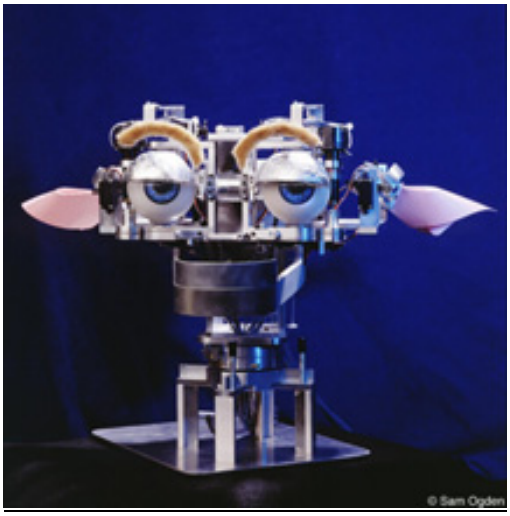


figure 3.6 – Kismet's mechanical head

Kismet distinguishes between several kinds of interactions. In the "social stimuli" category, Kismet puts every interaction involving a face-to-face contact with a human or with the face of a plush cow that the caretaker shows him. Among the "non-social stimuli", one finds Kismet's favorite toy (an orange cubic plush) being agitated in front of him, and a human hand being waved at him. The motion generated by the object gives Kismet a measure of the interaction intensity. The identification and tracking of each object is performed by image correlation on dedicated hardware.

When interacted with, Kismet reacts by changing his internal “mood” and displays reactions on his “face”, mimicking human expressions (see figure 3.7). For instance, Kismet will get bored if nobody interacts with him; he will be happy if a toy is waved in front of him, but will grow scared or disgusted if interaction becomes too intense. The visual expressions displayed by Kismet are clues to the caretaker on how to modify the interaction so that the robot remains in a happy, receptive mood. The caretaker and Kismet are working together to keep Kismet’s emotional state within certain boundaries.

figure 3.7 – Kismet’s attitudes

The aspect of Kismet that is relevant to H.O.T. is the *direct interaction* that takes place *both ways* between the human and the machine. The robot perceives the human behavior and reacts to it; similarly, the human takes notice of the robot’s expression, and modifies his behavior accordingly. Kismet’s system organization is shown in figure 3.8.

This process involves direct visual attention to humans and human activity. The response Kismet displays is a translation of the human activity (namely, waving one’s head before Kismet’s eyes or waving a toy) into human intention (the level and characteristics of interaction). This intention is then further interpreted into feelings (loneliness, happiness, surprise, fear, …) that the interaction would provoke in an infant.

By using an approach similar to the Kismet project, but with further activity analysis, one could achieve better seemingly natural interaction with machines. For instance, if Kismet were to be aware not just of a face and a toy, but of the caretaker's body pose, actions, apparent center of interest and actual enthusiasm in the relationship, a much more complex range of interaction could be achieved. Therefore, a better model of learning in a social context could be built, leading to better human machine interaction.

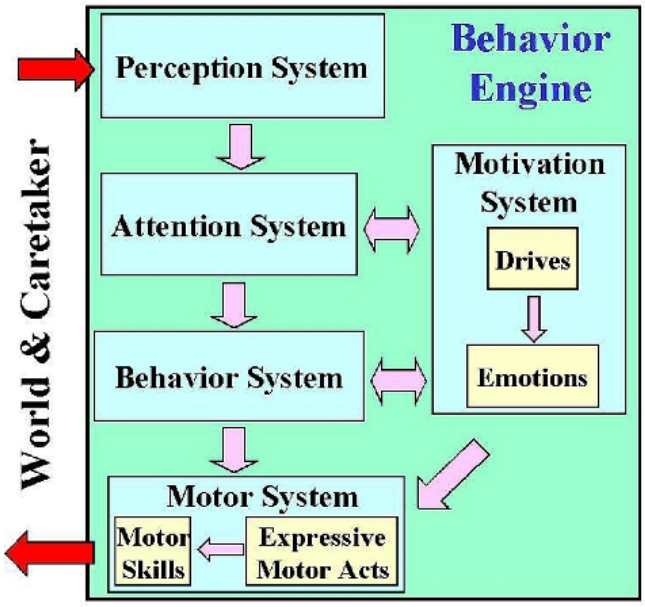


figure 3.8 – Kismet's system architecture

3.2. Related problems

Using sensor fusion to track human activity asks for solutions to complex sub-problems. The two main tasks required to perform activity monitoring are people tracking and human pose modeling.

The current commercially available motion capture systems are either based on magnetic or optical trackers that require the subject to wear a special suit with markers on it, or even to be attached to the system by wires.

There are good people trackers that don't model humans, and good modeling methods that can only track under dramatically constrained environments. The idea behind H.O.T is to merge a simple version of each to make a reliable, powerful tracker, with some level of pose modeling, to then achieve gesture tracking and activity measurement for appropriate output modality selection.

3.2.1. People tracking using sensor fusion

People tracking is a common problem. In [10], disparity map from a SVS stereo pair is combined with background subtraction and shape correlation in intensity image (gray levels) to track people. There are several limitations to this system. First, it is relatively sensitive to occlusions and performance drops when the number of people to track augments. Also, using non-adaptive background subtraction does not allow the system to perform its tracking task on a mobile platform, such as a mobile robot, as reliably as it does with a static camera. Finally, the SVS stereo module used in this case only supports intensity images; thus, color filtering is not an option, even though useful data could be obtained through color processing.

Closer to this project is the work presented in [11], which uses a combination of range data, color data and face pattern recognition to track humans. The system not only tracks multiple users and locates their heads, it also tries to identify known users over several time scales, using color histograms and face correlation. An illustration of the different modalities used can be seen in figure 3.9.

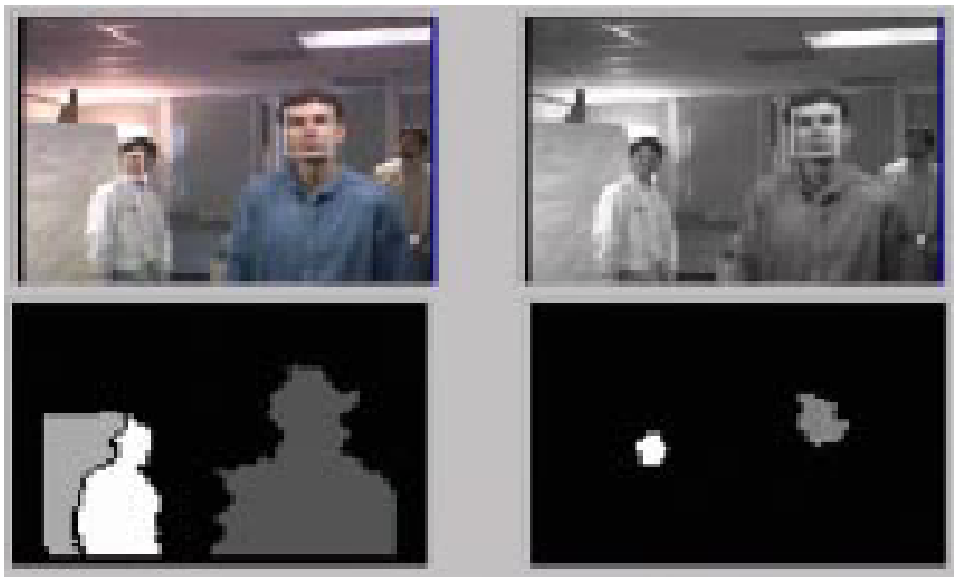


figure 3.9 – vision-based people tracker
input image, face pattern detection output,
stereo range data and flesh color filtering

This sensor fusion scheme achieves great performances, even in crowded environments, and can locate heads with remarkable accuracy. However, several characteristics make this system different from the one presented in this project:

- Woodfill's system requires two SGI workstations and one PC, and uses a dedicated PCI board for stereo computation (for an average performance of 12Hz)
- it also requires training, as it uses a neural network library for face pattern recognition
- no particular features, besides the head, are identified in the person tracked
- no modeling of the human posture is made
- no interpretation of human behavior is made

This system is a very reliable person tracker, but does not do anything else other than pure tracking. The H.O.T. project differs from Woodfill's tracker in two main areas:

- performing reliable tracking on standard hardware using more selective integration techniques and performance self-evaluation
- modeling the human posture, which involves feature extraction, localization and analysis

3.2.2. People posture modeling

Another part of this project consists in performing some level of modeling of the human body. The modeling should be accurate enough to allow meaningful posture description, while simple enough to be performed in real-time.

Many works have dealt with real-time human body modeling so far, ranging from complex 3-D models as in [12] to simpler geometric descriptions. This section presents two relevant examples.

Stick model

The tracker in [9] is intended to track and model any articulated objects made of sticks. It follows a non-redundant parameters modeling approach, where each segment is defined with respect to a reference point in a unique, deterministic way. Using a disparity map, it extracts blobs that are statistically mapped on a predefined, articulated structure. Range data allows the system to deal with occlusions much better than other 2-D based trackers.

To initialize the system, the subject is expected to assume a certain pose based on marks in the image, so that the different regions in the disparity map are assigned to different body parts and the initial model can be built. The joints are also assumed to be at certain fixed positions in the image plane during initialization. Therefore, the system is extremely sensitive to initialization parameters, and requires refinements such as taking body measurements and combining these with joint refinements.

Also, because it is based only on range data and a crude statistical model, the system has difficulties estimating a rotation about an axis parallel to the image plane. For instance, the model shown in figure 3.10 cannot capture the head pose very well (namely, the degree of rotation around the vertical axis).

figure 3.10 – stick-based human body modeling

Even though the technique is general enough to theoretically model a complete human body, this system has only been implemented on a 4-sticks model of an upper right human torso (figure 3.10). Because of its sensitivity to initialization and the fact that it only uses intensity image combined with range data, this tracker can as yet only run under very limited conditions.

Blob model

In [7], a system is presented that builds and tracks a blob-based model of the human body. The tracking ability of the system is based on adaptive background subtraction and is thus severely limited:

- it only tolerates one person in the image
- it doesn't differentiate people from objects
- background needs to be static (or to change very slowly over time)
  therefore, only a fixed camera can be used

Nevertheless, the interesting part of the research is that a model of the human body is constructed using morphological growth of connected areas to obtain a set of connected blobs (see figure 3.11).

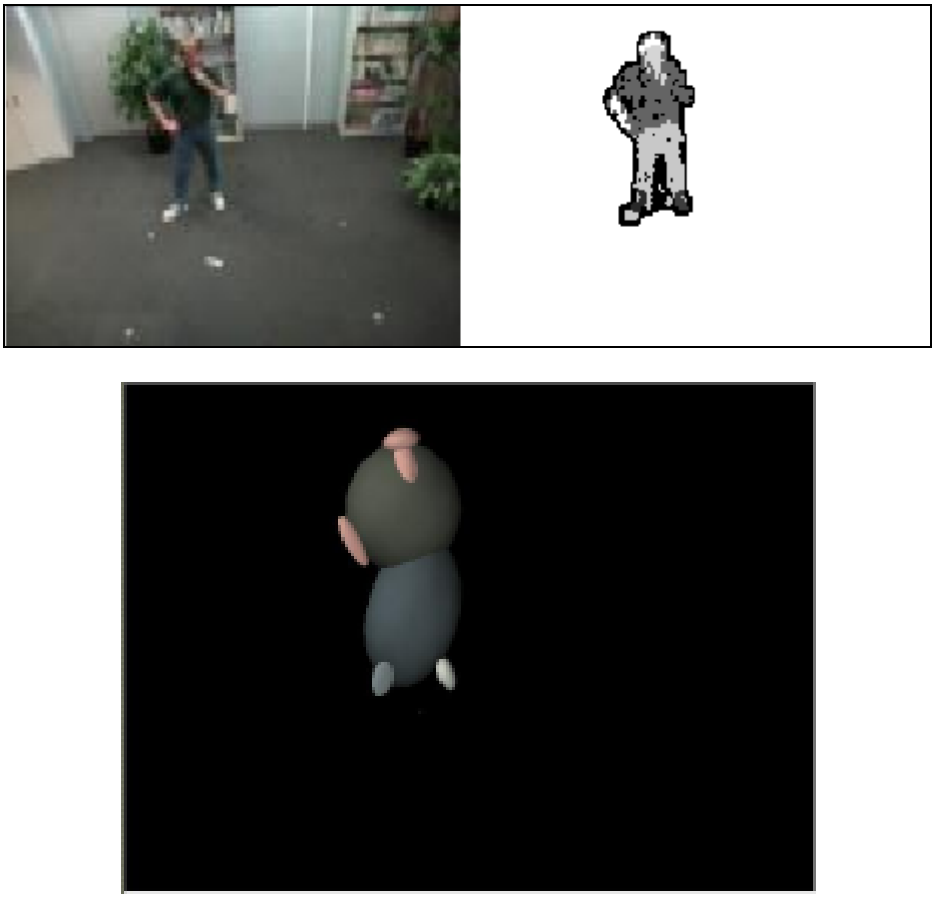


figure 3.11 – Pfinder: human figure extraction and blob-based model

This model has been used in an ASL (American Sign Language) recognition application, using hidden Markov modeling applied on the blobs representing the hands [8].

While this model building is very powerful in itself, it is difficult to use in a natural human-computer interaction application, mainly because of the constrained environment it requires. However, it is a famous example of real-time people modeling performance.

An attempt to make this tracker work in 3D has been made in [13] using a stereo system. The two cameras are not used to calculate a dense disparity map, but rather to estimate 2-D blob parameters in each image by running the 2-D tracker described above on the two image sources. *Spfinder* (for *Stereo Pfinder*) has been used in a small, controlled desk-area environment to track the head and hands of a human subject. However, the constraints on the environment remain the same as the 2-D model. Tracking and modeling results from the Spfinder are shown in figure 3.12.

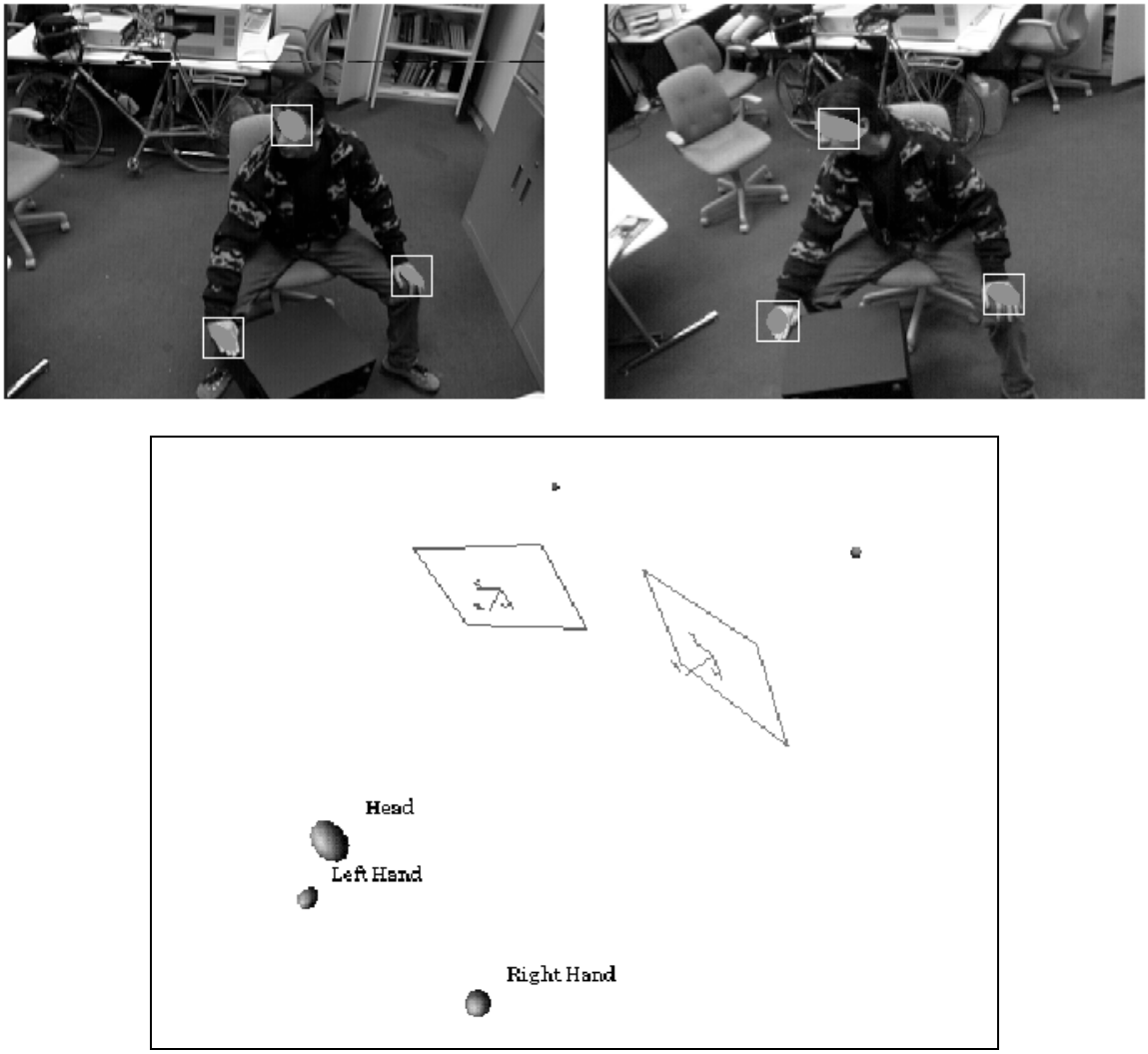


figure 3.12 – Spfinder tracking and modeling result

Again, this blob-based model building approach depends heavily on the segmentation technique used (in this case background subtraction). Therefore, it might be more difficult to implement in an unconstrained environment and without using background subtraction (which are two requirements for H.O.T) can be shaky and imprecise.

4. H.O.T. system design

4.1. Overview

H.O.T. architecture is based on the successive integration of information over 3 layers, as illustrated in figure 4.1.

Each layer has its own input and output, independently from its link with the two others. As a result, the tracker can track objects manipulated by the user without affecting the human model, the human model can be used by other applications and the activity layer can be consulted without much care about how it's been updated.

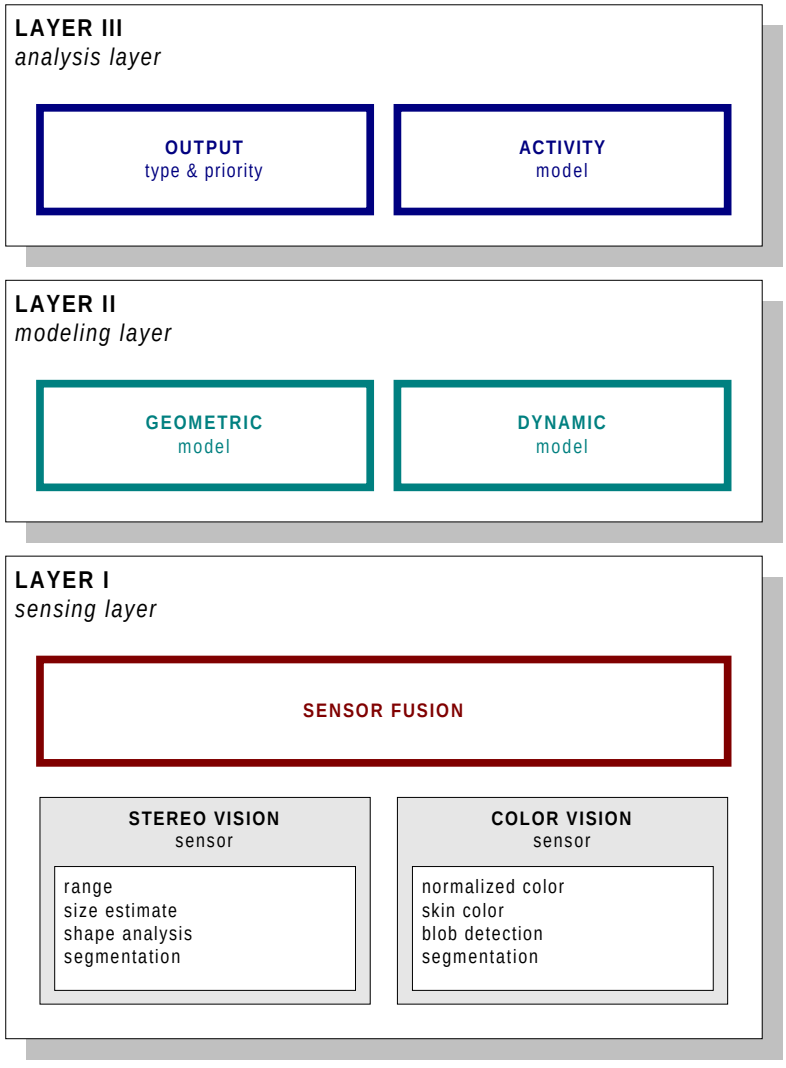


figure 4.1 – H.O.T. integration layers

Layer I: tracking

The first layer could be described as the sensing layer. It combines all of the available inputs and processes them so that they locate and describe a human feature. The sensing modalities used in this project are described in section 4.2.

The role of this layer is to actually fuse the information properly so that each output can be attached to a feature of the human model and update its state regularly. For instance, the color and stereo camera outputs are attached to three basic components of the human model (namely the head and the hands), and update their position and dynamics at each iteration of the tracking program. A sound-source localization, using a microphone array, could be fused with the visual tracking system to correctly associate a segment of speech to the correct human being, and then attached to a speech module of the human model.

Layer II: human model

The second layer integrates all of the output of the first layer and produces a model of the human beings pose and movements. It does not only involve a geometric model of the human body, but also a dynamic model representing the motion of the person. The positions of key features of the human body are updated through layer I, and layer III then uses the resulting model to build an activity model of the person.

The models developed for this project are described in section 4.3.

Layer III: activity

The final layer offers the highest abstraction. Its task is to integrate all the features of the human model (both spatial and temporal), and to characterize the activity of the human, as defined in section 2.2. In the vision based application developed in this project, this layer would take into account the pose of the human user and information about motion and gestures over time in order to qualify the general activity of the person. This data is available through layer II only, which makes the activity model totally independent from the sensing modality (layer I).

The activity model is described in section 4.4.

Interfaces between layers

As each layer is independent of the others in its function, but still has to communicate with them, interaction between layers has to be designed carefully. Interface design is described in more detail in section 4.5.

4.2. Sensor fusion-based tracking engine

4.2.1. Multiple sensor sources

The sensing layer consists primarily of the tracking engine. Sensor fusion is performed on two sources of data to achieve efficient tracking. In this section, the two modalities used for tracking are described in detail. Section 5 will then present the actual tracking algorithm that makes use of the sensor data.

Color data

Color is a powerful, easy-to-use way to track objects. It offers the possibility to filter the image and to keep only the parts that match a certain tone. This makes a correlation-based search much faster and more accurate. It can also be useful to identify objects or object states.

Several color spaces can be used to filter color information. The choice mostly depends on the properties of the objects to track, along with the computational effort the transformation between color spaces requires and the memory space each color vector occupies. Commonly used color spaces are given in table 4.1, along with the dimension of the space.

color space	dimension	components
RGB	3	R red intensity G green intensity B blue intensity
normalized color	2	NR normalized red NG normalized green
gray	1	intensity
YUV	3	Y luminance U chrominance V chrominance (note: YUV is used in PAL TV broadcasting)
HLS,HSI, *HSB,HSV*	3	H hue L luminance S saturation B brightness V value I linear-light quantity

table 4.1 – commonly used color spaces

For more information concerning color spaces and manipulation, see [25].

The specifications of the hardware related to color sensing used in the development of this project are listed in appendix A.

human tracking specific interest

There are two interests in using color data to track human beings. The first major advantage is that skin color occupies only a small portion of the normalized color space (figure 4.2). Surprisingly, this property does not depend on the race of the person. While the luminance of the skin can vary dramatically between people, the chromatic information is fairly constant [26].

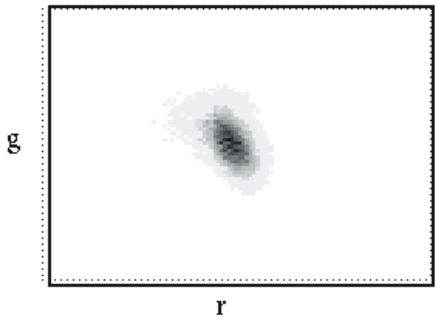


figure 4.2 – skin color domain in the normalized color space

Figure 4.3 shows a color image, and the filtered mask using a normalized skin color filter. White pixels represent elements of the original image with a color similar to skin color.

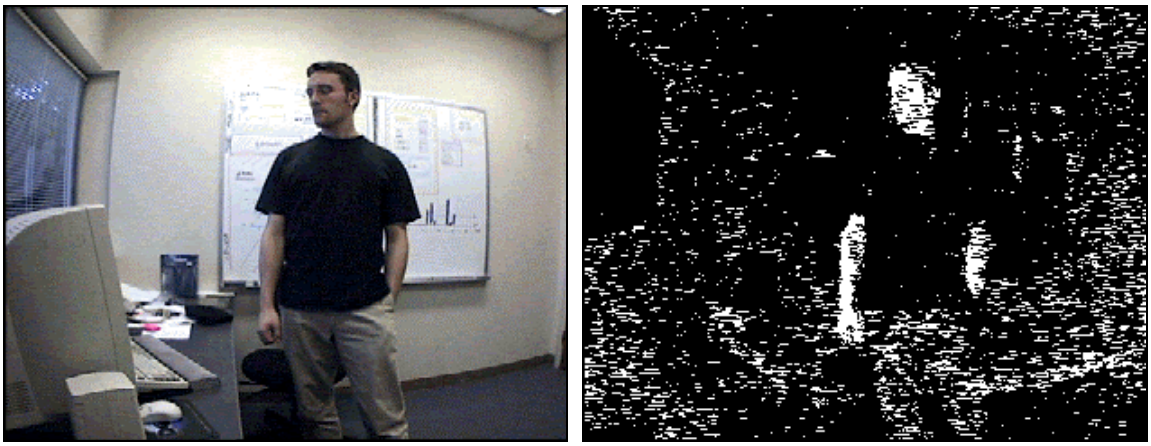


figure 4.3 – normalized skin color filtering

The second advantage of using color is user recognition. It has been shown [22] that a “color signature” using color indexing can be retrieved from a color image, and used to identify a human being in images. H.O.T. will not include this feature in its first implementation, but provides all the tools to perform this identification task.

limitations

One limitation of color based skin segmentation is its sensibility to changes of light. If the illuminant changes (e.g. due to time of day, weather, …), the color spectrum of the objects in the world will change accordingly, affecting the color filtering. While color works well in environments with controlled lighting conditions, it is however less reliable in places where the light changes over time. How sensor fusion can counter this limitation will be described later in this section.

Depth data from stereo

Stereovision is a technique that takes two images from a camera pair and uses pattern matching to retrieve the depth information. The two cameras are pointing in the same direction and are separated by a fixed, known distance (the *baseline*). Stereovision is somehow similar to human vision, even though the latter is far more complex.

A schematic of a stereo system is given in figure 4.4. First, two images are grabbed from the cameras (I); then the images are scaled and optical aberrations are corrected (C). The pair is then sent to the pattern matching processor (D). The result is a disparity map (M), which is finally filtered for errors (F).

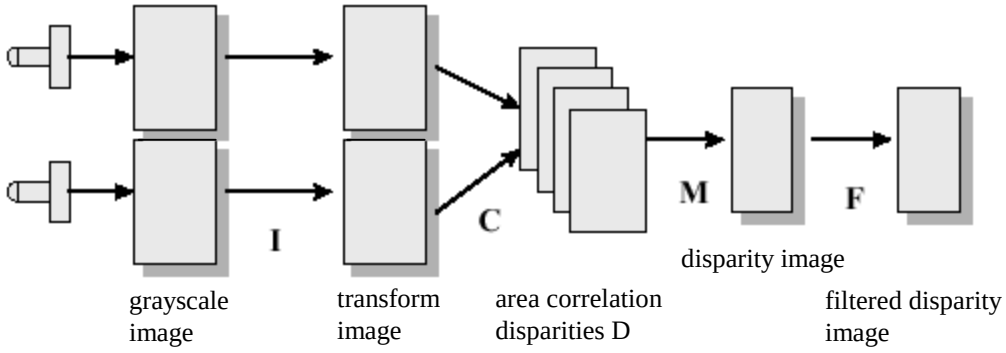
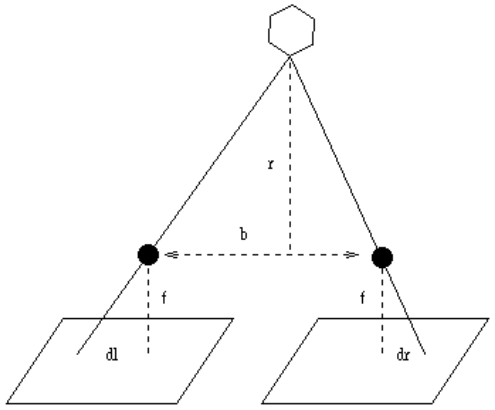


figure 4.4 – stereovision system

The output of a stereo system is a disparity map. Disparity is a measure of the shift of the pattern between the two images. It is related to depth through the expression:



b: baseline

d: disparity

f: focal length

r: depth

$$r = \frac{b \cdot f}{d} \text{ where } d = dl - dr$$

figure 4.5 – calculating depth from disparity in stereovision

A complete description of the theory and implementation of the SVS system can be found in [18]. The specifications of the hardware used for this project are listed in appendix A.

human tracking specific interest

There are three features of stereo data that can be used to track humans. The first interest is that stereo images can retrieve shapes, independently from color. As shown in figure 4.6, objects in the foreground can be distinguished from the background. Not only can one segment objects from the rest of the image, but one can also analyze the shape of these objects even if they are made of several components with varying colors.

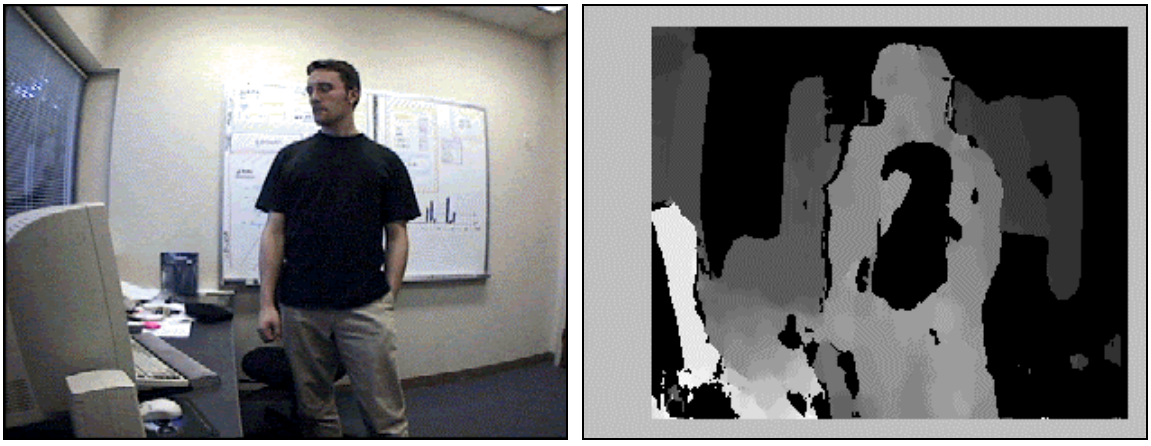


figure 4.6 – stereo processing
the darker each pixel is, the further away from the cameras plane;
black pixels indicate areas where stereo processing failed.

The second interest of stereovision is that it can be used to estimate real-world sizes. As the distance of an object is known, one can extrapolate its real size from its size in the image. This information can then be used to locate objects of a certain size, or help determine the nature of an object.

The last benefit is that stereo information can be very useful in tracking occluded objects. Using range data, it is possible to detect if an object is *behind* another, and to compensate for that occlusion.

limitations

There are several drawbacks when using range as a tracking modality.

First, in order to find matches in the stereo image pair, a sufficient level of detail has to be present; therefore, stereo processing only works with textured surfaces.

Sensibility to calibration is also an issue. For accurate range measurement, the stereo system has to be carefully calibrated. It then becomes sensitive to mechanical disturbances.

Also, depth measurement range is not unlimited, and depth resolution is not linear. Parameters can be adjusted to correctly measure range data in a given “slice” of volume (the *horopter*); objects outside these limits will be given unpredictable disparities that are not usable. Finally, and because of the reasons mentioned above, disparity processing can be a noisy process.

Summary table

Advantages and limitations of each sensor modality with respect to different tracking issues for 3D tracking are shown in table 4.2.

feature	stereo	color
depth estimation	+	-
skin detection	-	+
sensitivity to texture	-	+
sensitivity to light condition	+	-
size estimate	+	-
shape analysis	+	+
re-identification	-	+

table 4.2 – comparison between H.O.T. sensor modalities

Color-based skin segmentation limitations can be sensibly countered by fusion with depth data. As depth allows for shape and size analysis, it can reject false positives from the color module. Therefore, a more tolerant skin color model can be used, which is less sensitive to light condition changes. Similarly, shape analysis is generally not sufficient to identify a human feature. Skin color detection is an efficient way to provide this information and to discriminate non-human objects from people features.

Figure 4.7 shows a simple fusion result, where only skin-colored objects in the foreground are picked. When compared with figure 4.3 and 4.6, the amelioration that sensor fusion brings is straightforward. A more complex sensor fusion scheme is described in section 5.3.

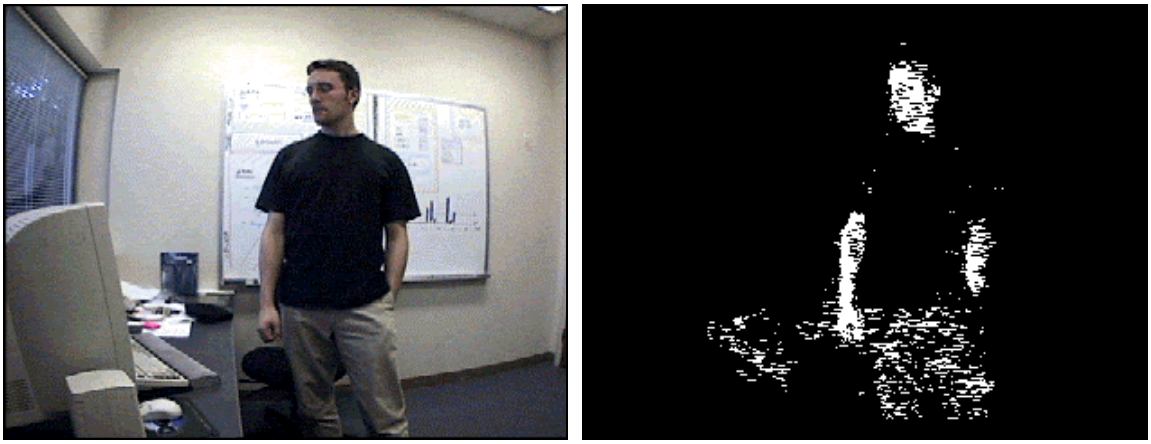


figure 4.7 – skin color and depth data fusion

Additional input (not used in H.O.T.)

Because the model developed for this project (see section 4.3) is not specific to the type of sensors involved in keeping it up to date, other useful sensing devices could be considered and integrated into the system. Their role can be either to make tracking more robust, or simply to enhance the quality and complexity of the human model.

IR camera

An infrared camera could be used to discriminate “living” components from inert ones. This sensing modality could make tracking more robust in environments where both color and range information experience problems. For instance, rooms with skin-color like walls that may confuse the color-based tracker, or low-quality images that do not display enough details for the stereo system to locate the person in depth, could take advantage of infrared input to successfully detect human features.

sound source localization

Using sound in general could lead to an interesting enhancement of H.O.T. Using a microphone array, one could detect the origin of speech to confirm (or discriminate) the localization of a human head. Models have been developed to provide sound source localization. Simple models deal with source heading only [1], while more complicated ones allow for azimuth and elevation estimates [2].

Moreover, the same input modality could be used in the human model, making it more complete and allowing for more activity monitoring. Ultimately, tracking and speech processing could be coupled to lead to multimodal human-machine interaction. It has been shown [3] that fusion of the data from both sound and vision systems can make tracking of the active person more robust (by allowing identification of the speaker). Sound in itself is also a very strong clue as to what the user is being attentive to, and what his/her intentions are.

laser range-finder

Finally, it is now possible to find small, low-cost laser range-finder devices that could be used to locate the person more accurately. This sensing modality could make up for the lack of accuracy and the resolution loss typical to stereovision.

4.3. Human body and actions modeling

4.3.1. Overview

The human model developed for this project has been designed with two main concerns in mind: simplicity and extensibility.

Simplicity

There are several reasons why the model should be simple. First of all, the accuracy of the sensing system used to update the model should be somehow correlated to the model itself. In particular, a vision-based tracker as developed in this project cannot be nearly as accurate as mechanical or active sensing systems currently used for many virtual reality applications (like in [19]), mostly because of the real-time constraint. Therefore, a complex articulated model would be inappropriate and could only lead to a more error-corrupted representation.

Also, the purpose of this model is to represent human activity in a general way, as defined in section 2.2. For this purpose, it is not necessary to know the exact pose of all the components of a human body.

The model developed for H.O.T. is then mathematically simple (9 parameters), and matches the computing power of the platform used.

Extensibility

There are two main reasons to make the model easy to extend.

First, the model is designed independently of the sensing modality (that is, it is not specific to the type of sensor used to update it). As a result, it can be used by other applications, or with other tracking modalities. Therefore, the model has to provide general parameters that can be updated by different sensor inputs; it should also leave room for more parameters that new sensing modalities would bring. For instance, if one decides to use an infrared-based visual tracker with a microphone array, the model should not only tolerate geometric updates from the infrared tracker, but also allow it to be extended easily to integrate sound.

Also, as computing power increases exponentially over time, it might soon be possible to track more features of the human body than the current hardware performances allow. For this reason, the design of the model has to take this possibility into account, and make sure that new features can be easily included.

4.3.2. Objects

The model developed for H.O.T. is object-oriented. Only certain features of the human body are being paid attention to, and the model is built around these objects position and behavior. For a complete description of an object, see section 5.3.

The advantage of using an object-based model is that it is a very modular way to “build” the human body. Given the computing power available, the sensing modalities used to update the model and the purpose of the model, one can add more objects, or remove some. Possible objects in the human body can be:

- head
- arms, hands
- torso
- legs, feet

In this project, the model has been given three objects, namely the head and two hands. The reasons for this choice are:

- three objects is the maximum amount of objects the system can follow on the available hardware (see section 5.1)
- head and hands are sufficient to detect actions such as pointing, grabbing, …
- these objects are sufficient to extract a meaningful measurement of human activity

4.3.3. Pose

Once the sensing layer updates head and hand positions, the modeling layer builds a simple mathematical model of the pose. The model is illustrated in figure 4.8. From the features positions, two sticks are extracted, assimilated to the right arm (in red) and the left arm (in green).

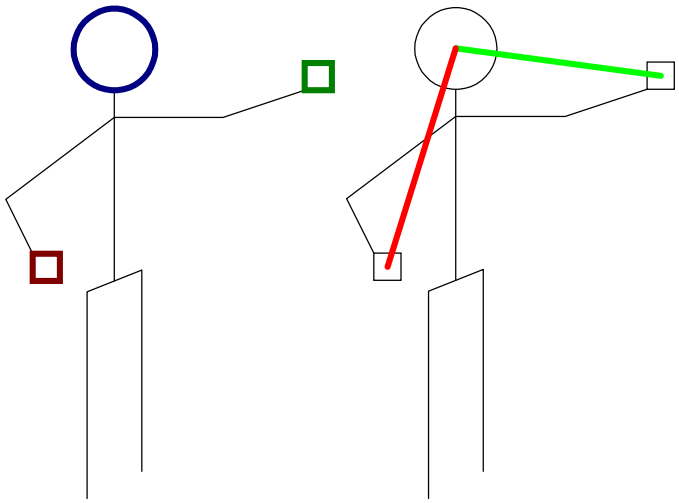


figure 4.8 – human model tracked features and parameterized elements

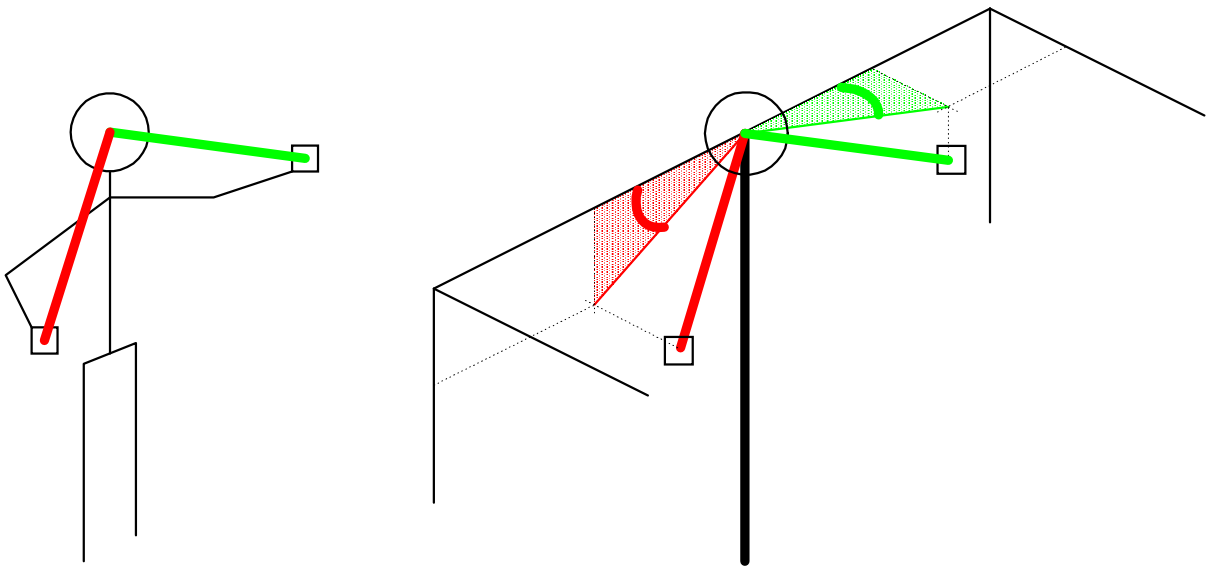


figure 4.9 – human model projection angles

Figure 4.9 shows the projection angles of the arm sticks. The angle shown in red (right arm) is the projection angle in the vertical plane, while the one shown in green (left arm) is the arm projection angle in the horizontal plane. Figure 4.10 illustrates the final model with its 9 parameters.

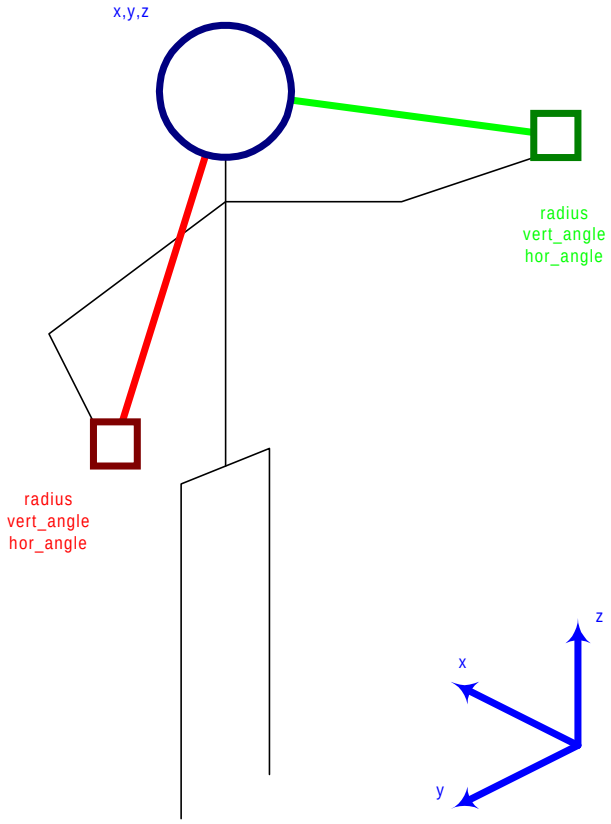


figure 4.10 – human geometric model

Table 4.3 presents the parameters of the human model:

parameter	description
x, y, z	Location of the head
radius (left, right)	Distance from head to hand
vert_angle (left, right)	Angle of projection of arm in the vertical plane
hor_angle (left, right)	Angle of projection of arm in the horizontal plane

table 4.3 – human model parameters

Figure 4.11 shows images with the geometric model objects in overlay. The white square is the head, the red cross designates the right hand and the green one locates the left hand. The status bar under each image states the model hands parameters (for the **R**ight and **L**eft hand) in the following format:

```
[radius] cm ( [vert_angle], [hor_angle] ) deg
```

figure 4.11 – human geometric model examples

4.3.4. Motion modeling

The object-oriented approach makes it easy to associate movements with tracked objects. Typically, *movement objects* will be attached to *physical objects* tracked by the vision system. There are two main questions to face when trying to define a motion in general, a human movement in particular:

- how to segment a movement from others
- how to encode a movement

These questions have been answered from the human movements tracking point of view, as this section will illustrate. However, it seems that the conclusions are generic enough to be applied to many other categories of motions, including vehicles and mechanical structures.

Motion segmentation

People can easily interpret other human beings gestures. Not only can they interpret the intention of a gesture without any effort, but they can just as easily integrate several disconnected movements to understand a complex activity. For example, we interpret lifting a bottle, pouring some of its contents into a cup and putting it back on the table as one action. On the other hand, numerous sports require simultaneous execution of several movements with different parts of the body, even though it appears as one general pattern. The way human beings segment and learn movements takes the motion “meaning” into account.

As computers don’t have this ability to segment actions according to the intention they represent, the only simple way they can distinguish one movement from another (in the temporal domain) is based on motion. One could think of two ways to use quantity of motion to segment movement:

- velocity
  movements of an object can use moments of zero velocity as boundaries; for example, lifting a bottle, then holding it still while pouring its contents into a cup, finally putting it back on the table, would be segmented in 2 different movements (lifting it, and putting it back)
- heading changes
  in the same way an object null scalar velocity can be a movement segmentation boundary, changes in the heading of the motion can indicate the start of another action; taking the example of the bottle again, the bottle travels towards the cup, then back, defining again 2 different movements.

While the heading based segmentation would theoretically allow for faster, subtler movement decomposition (the segmentation routine could extract several movements from a sequence where the object doesn’t stop, such as a mobile robot taking turns), it contains in itself several drawbacks:

- in the case of a curve trajectory with no noticeable, abrupt heading changes, there would (virtually) be an infinity of movements; the integration and interpretation in real-time of numerous short segments could be a problem.
- if the tracker system is not of great accuracy (as real-time systems usually aren’t, due to model simplification constraints), the risk of producing non-negligible amounts of false positives and misses of heading changes is real.

Therefore, movement segmentation based on velocity seems like the most appropriate solution. However, ignoring the dramatic heading changes would be a mistake. The next section explains how a simple movement model using velocity segmentation can take into account heading changes information.

Motion representation

In a first approximation, one could consider that human beings use two different kinds of movements:

- fast, “direct” movements
- slow, “meaningful” movements

The first category groups together transition movements (stretching one arm to grab an object, walking towards a precise location), while the second one considers movements that require skills, or whose trajectory has a special meaning (like drawing on a white board, or dancing).

The major difference between these two categories is the trajectories of the hands or the head. Fast, direct movements are generally very close to a straight line (the faster, the more straight); on the other hand, slow gestures that require concentration or manipulation of a particular tool are in general anything but linear.

On top of distinguishing between two different kinds of movements, the motion model should be accessible not only once the movement is finished, but also should provide information while the movement is being executed. This way, the computer could have real-time indications about the action undertaken, and could maybe anticipate the user’s needs.

To keep the model simple, few parameters should be used. Once a movement is started, the motion object tries to interpolate them onto a single line, using a simple version of the Kalman filter. If the trajectory cannot be fitted into the model (i.e. too many points deviate from the linear approximation), the movement is considered to be of the “meaningful” kind.

Moreover, some human gestures that are not “meaningful” are repetitive, for instance grabbing something, or going somewhere and back, or simply waving at somebody. The model should take into account the periodicity of certain gestures, and parameterize the trajectory of the repetitive gesture instead of considering each period to be a new movement.

The best way to describe a motion according to this approach is to remember the movements starting and ending points, along with parameters representing the spatial orientation of the linear motion. For computational efficiency, the 3 projection angles of the motion segment in the world frame have been chosen, as described in the next section (see figure 4.12).

At any moment during the movement, the Kalman filter computes the best fit for the projection angles of the object trajectory in the world frame. This data is permanently available to the rest of the program, along with a flag indicating whether the movement can be considered roughly linear or if it is more complex.

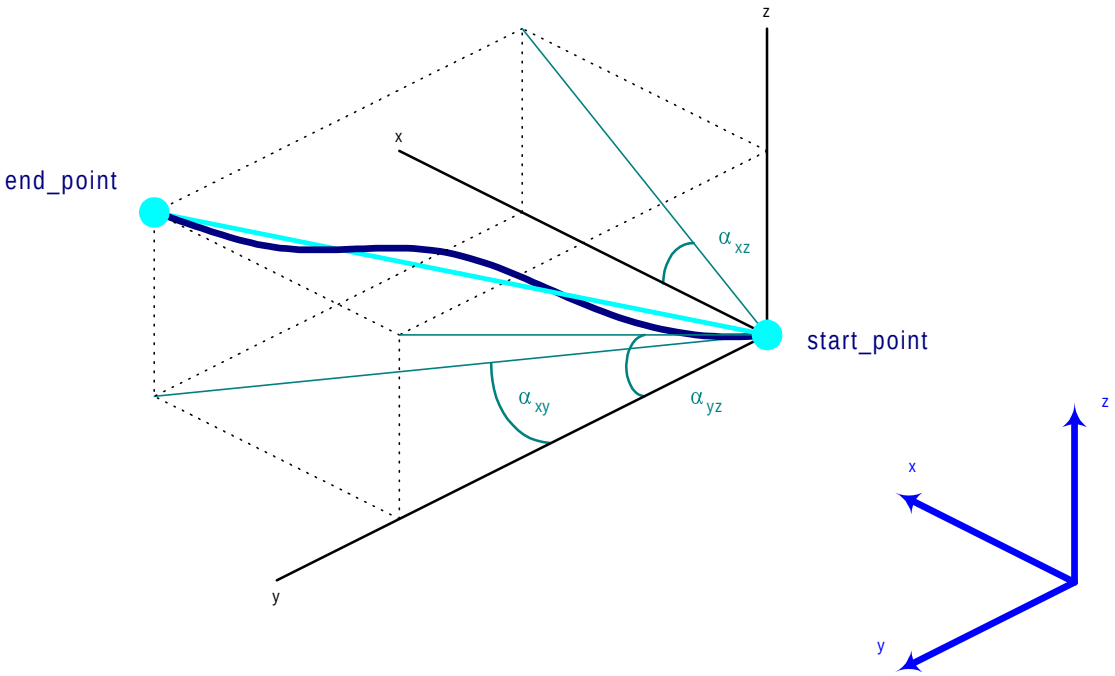


figure 4.12 – linear motion model

Table 4.4 summarizes the parameters of the movement model:

parameter	description
start_point	location of motion origin
end_point	location of motion termination
α_{xy}	trajectory angle projection in the XY plane (with respect to the Ox axis)
α_{xz}	trajectory angle projection in the XZ plane (with respect to the Ox axis)
α_{yz}	trajectory angle projection in the YZ plane (with respect to the Oy axis)
linear	linearity indicator (yes/no)

table 4.4 – linear motion model parameters

Figure 4.13 illustrates the movement segmentation and parameterization. Red lines show linear motions of the right hand, green lines are segmented linear motions of the left hand. Orange lines stand for non-linear movements. In the image sequence, the human being depicts a triangle with his right arm, pausing for about half a second between each segment; he then brings his arm back to his left shoulder depicting an “S” (i.e. not linear) curve.

figure 4.13 – motion segmentation examples

Kalman filter

In order to have a precise estimate of the motion parameters during the movement execution, the motion object implements a Kalman filter.

The Kalman filter is a set of mathematical equations that recursively computes the solution of the least-squares best-fit method. It was described by R.E. Kalman in 1960 (see [15]). A detailed mathematical description can be found in [16] and [17]. The properties of this recursive method make the Kalman filter very popular:

- it doesn't require memorization of past states
- it can estimate past, present and future states of a system
- it doesn't require an exact model of the system behavior

There is a lot of documentation about the Kalman filter available. Shortly, the digital filter uses a linear model of the state equations of the system to predict the future state. Once this new state is reached, it processes the noisy measurements and updates its parameters to match the least-squares criterion. The process is illustrated in figure 4.14.

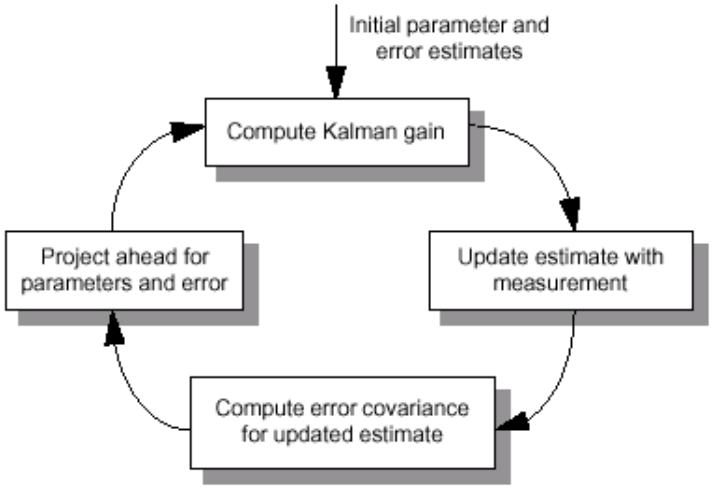


figure 4.14 – the Kalman filter

Kalman filter has several nice features that make it appropriate to use for movement parameterization:

- unlike simple averaging, the Kalman filter optimizes the likelihood of the model to be correct
- because of its structure, it tolerates a certain amount of error in both the process and the measurement. Therefore, the model can tolerate not only the inaccuracy of the vision system, but also the small deviations human movements have from a straight line.
- no memorization of the previous waypoints is necessary; the computational simplicity of the model makes it particularly suitable for a real-time implementation.

During motion execution, if the input from the vision system diverges from the Kalman filter estimates, the movement is considered non-linear. For what was defined before as "meaningful" movements, it is not the motion itself that is of interest, but each position of the user's hand (tracking, ideally, should be continuous). Thus, there is no need for a model of the movement anymore, and direct processing of each waypoint can be substituted to the Kalman filter.

4.4. Human activity characterization

4.4.1. Overview

Extracting the exact activity the human subject is performing is not an easy task. Many systems, such as the one described in [20], require much more accurate (non passive) sensors and tremendous computing power to model the human pose and gestures. Even then, the identification of the user's actions is not always successful.

Other projects use neural networks or other heavy computing methods to identify known actions (i.e. sitting, walking, grabbing an object, …), like the motion templates described in [27]. Instead of trying to identify each action as a human action, this project aims at measuring the amount and the quality of activity of the user in a way that is close to how human beings perceive each other's activity. For instance, the computer should know if the user is involved in a physical activity, or a mental activity, or both.

The purpose is to then use this data to deliver information to the user in the appropriate way, and at the right time.

4.4.2. Activity model

Based on the geometric model presented in section 4.3, there are three parameters that can be extracted to measure the activity of a human being. Table 4.5 presents the activity model parameters.

parameter	description
hand_agitation	hand shape variation over time measure of concentration
hand_motion	hand motion in space measure of physical activity
body_motion	head motion in space measure of displacement

table 4.5 – activity model parameters

Hand agitation

Hand agitation (as opposed to hand motion) is a measure of how active hands are. This parameter is independent of hand motion in space, and relies only on both hand shape variations over time. Depending on the context, hand agitation can give the machine an indication of how concentrated the human is. For instance, in an office in general, hand agitation can mean typewriting, or writing.

Hand motion

Hand motion is a measure of hand displacement in space, independently of how much each hand is agitated. These parameters can be related to physical activity that doesn't involve displacement.

Body motion

Head displacement in space can be assimilated to body motion. This measure gives a rating of how much and how fast the person is moving in a given portion of space. In conjunction with hand motion, it is an indication of physical activity. No body displacement can indicate either no activity, or an activity that requires more concentration.

Temporal integration

Applications that make use of activity monitoring typically do not need an accurate record of all events over time. It is not very useful to know how busy a person was 3 hours, 16 minutes and 24 seconds ago. Rather, how busy the person was 3 hours ago, on average, can be useful. However, it can be interesting to know what happened during the past 10 seconds. H.O.T. offers 3 levels of integrations, namely:

- short term integration
  activity is quantified every second and results are kept in a short term result array
  typically in the order of 10 seconds
- medium term integration
  after a given number of short term iterations, H.O.T computes the average of the short term values and stores it in a medium term result array
  typically in the order of 10 minutes
- long term integration
  the same process is repeated again after a given number of medium term iterations
  typically in the order of 1 hour

Figure 4.15 shows the typical integration steps:

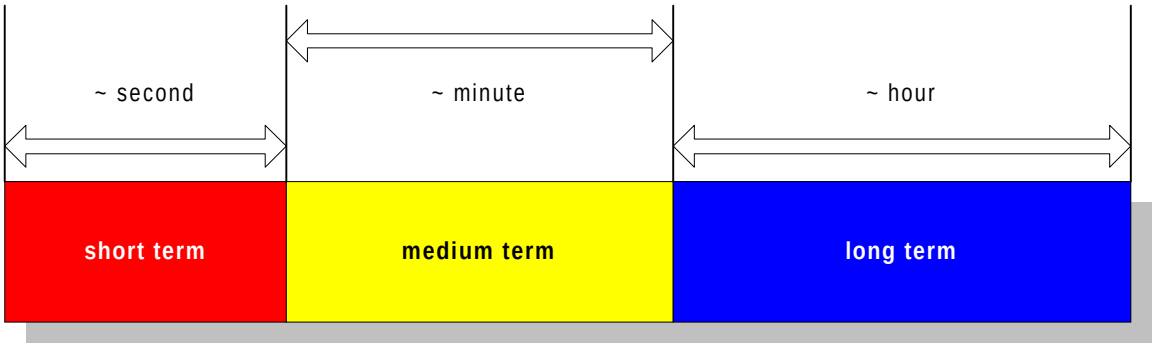


figure 4.15 – activity monitoring temporal integration periods

Activity classification

Without being too specific, it is possible to extrapolate certain characteristics about the activity the human displays based on the combinations of the parameters described above (table 4.6). These characteristics are purposely very general. When using H.O.T. in a specific context (i.e. at a specific location, or in relation to a particular device or action), more specific and useful information can be deduced from these basic activity indicators.

hand_agitation *without body_motion*	**activity description**
with hand_motion	• high level of concentration • medium physical activity • priority threshold: high
without hand_motion	• medium level of concentration • low physical activity • priority threshold: medium
hand_motion *without hand_agitation*	**activity description**
with body_motion	• low level of concentration • high physical activity • priority threshold: low
without body_motion	• medium level of concentration • medium physical activity • priority threshold: medium

table 4.6 – activity parameters combinations

The interest and versatility of the activity model depends heavily on the human model used. In the case of H.O.T., the simple geometric model doesn't allow for many nuances to be detected between different activities. Moreover, important modalities that would bring significant information to the activity model, such as speech and environmental noise, are not included in the human model.

However, the three parameters described above are sufficient to give clues about the level of activity the human being is displaying, and to demonstrate that meaningful monitoring is possible.

Examples

Figure 4.16 illustrates the activity monitoring of a human being in an office. After working at his computer for a couple of minutes, he then walks around the office to get documentation. These different activities are summarized in the graphs displayed on the H.O.T. control window.

The graphics below the images represent each parameter (3 top graphics), along with a general activity estimation (bottom graphic). Red elements show the short-term measurements (4 seconds in this case), yellow represents the medium-term (20 periods of 4 seconds), and blue displays the long-term integration results (20 periods of medium term measurements). The integration shown in figure 4.16 lasted for about 15 minutes.

The activity monitoring graphics clearly display how the hands of the human being are the only active parts while he sits at his desk typing. Similarly, walking around the office translates into body motion and hand motion, with less agitation.

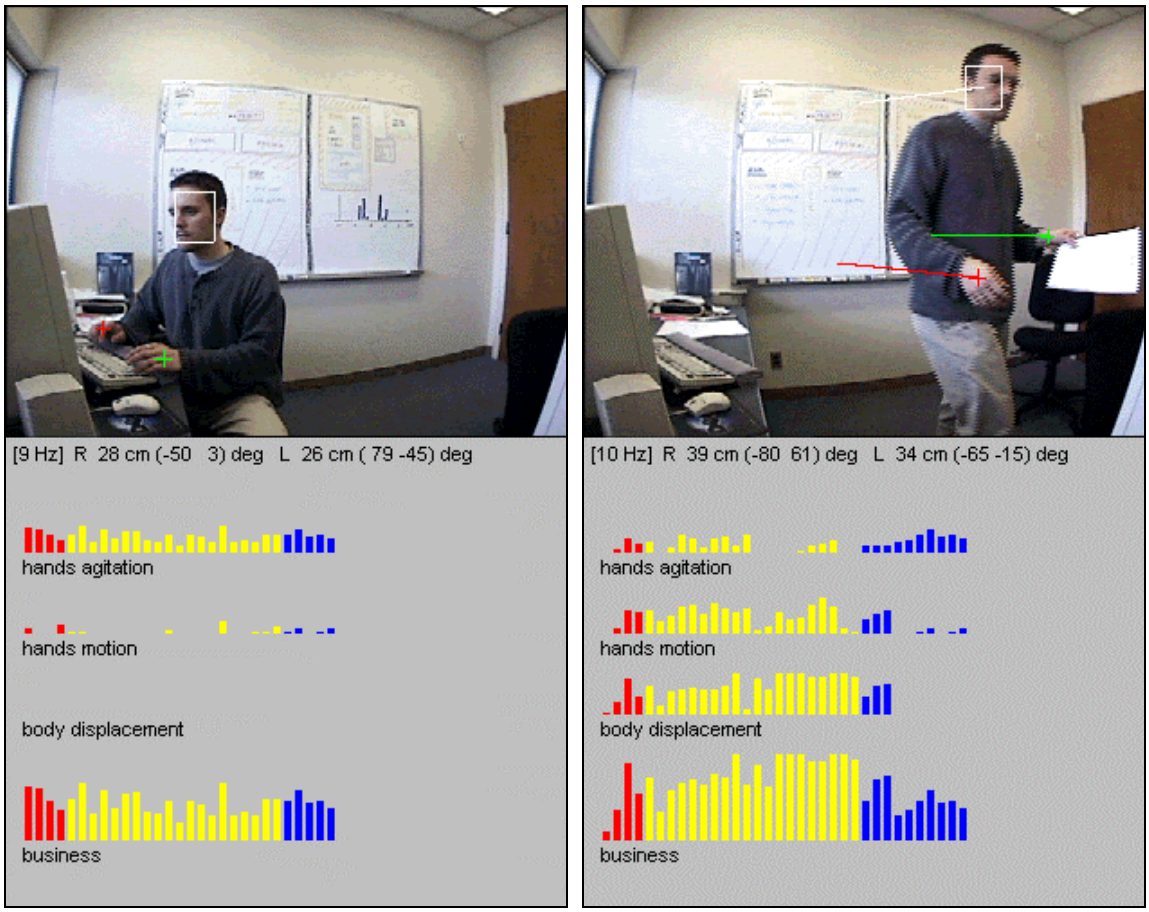


figure 4.16 – example of activity monitoring

4.5. Linking the tracker and the model

4.5.1. Importance

The tracking layer described in section 4.2 is meant to work independently of the nature of the objects it is supposed to track. For instance, one should be able to track a new object in the model without having to redesign the tracker. Similarly, the model should not need to be redesigned if changes occur in the tracker parameters or tracking modalities.

However, in the case of the human model, the three objects to track are not independent from one another. Typically, a hand can (hopefully) never be further away from the head than an arm length. There are geometric constraints in the model that lead to assumptions when it comes to locating and tracking its object components. Therefore, one must provide a way to make the tracker aware of these geometric constraints without making it specific to the model used.

The difficulty is to provide an elegant solution to give the tracker access to the model data without compromising the independence of the two. The tracker must provide a way for each object to be updated easily, and be given a way to consider model constraints when required.

4.5.2. Interface design

Object integration

In general, every object that has to be tracked (independently of whether it belongs to a model made of several objects) must be interfaced with the tracker. The way H.O.T. is designed is to include in the object data an element that contains all of the information the tracker requires, called a tracker object. Figure 4.17 illustrates this concept in the case of an object belonging to the human model:

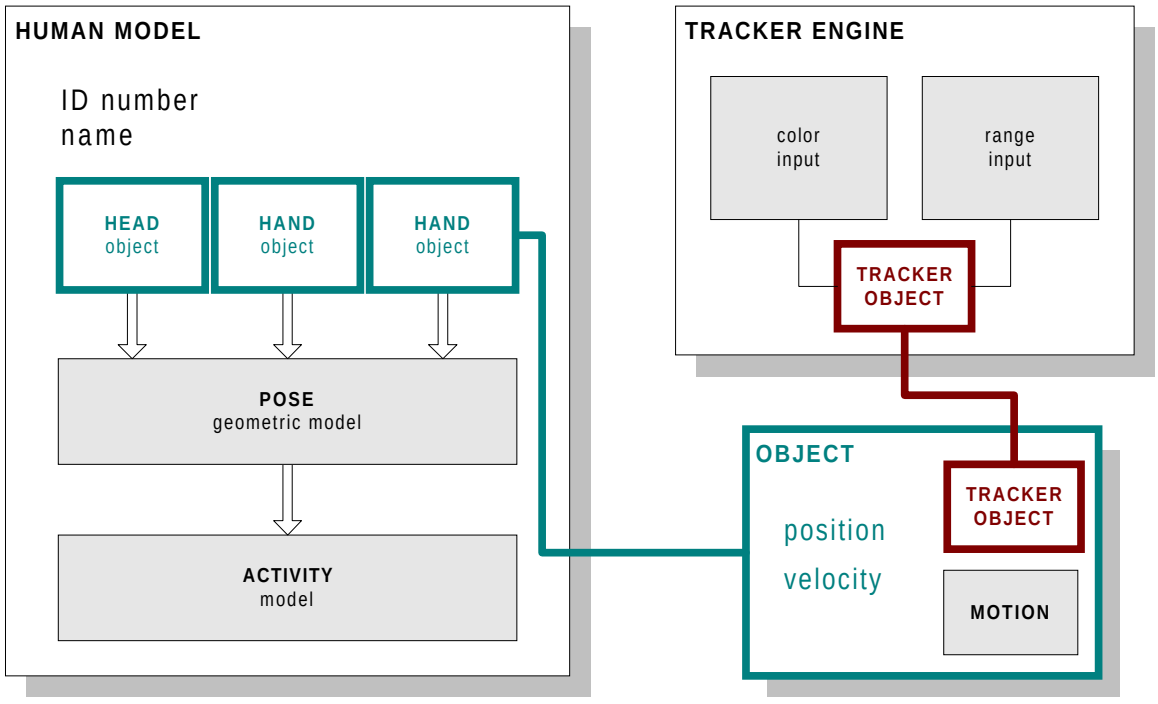


figure 4.17 – H.O.T model-tracker interface

Information required by the tracker typically include:

- color information
- size information
- last known position
- status (lost, occluded, …)
- etc.

When tracking an object, the tracker is provided with the tracker object only, updates the state of this tracker object, and returns the location of the object to the model.

Model integration

The solution proposed here to integrate model constraints into the tracking layer is to create *modules*. A module is a software piece that plugs into the tracker, but comes with the model. It is both model and tracker specific. Each model provides one module that contains tracking methods taking into account the geometric constraints between the objects that compose it.

For more details, see section 5.2 about the tracker implementation.

5. Implementation

This section is intended to give more insight into the way H.O.T. is implemented. It describes the technical aspects of H.O.T. and the software architecture behind it. The complete software interface is provided in appendix B.

5.1. Hardware

The hardware platform used to develop the H.O.T. application is:

- 1 PC Pentium III 700MHz, 128 RAM
- 2 frame grabbers ImageNation PXC200.
- 2 CMH color CCD cameras with wide angle lenses

Cameras used for color and stereo processing are described in appendix A.

5.2. Nomenclature

The tracking library and the human model library are two different software libraries. They each respect the following nomenclature scheme:

library	library name	public/global prefix	private/internal prefix
tracking library	trackLib	`tlFunctionName()` `tlVariableName`	`_tlFunctionName()` `variable_name`
human model library	hmLib	`hlClass.functionName()` `hlVariableName`	`hlClass.functionName()` `hlVariableName`

table 5.1 – H.O.T. software libraries nomenclature

In the following sections, code excerpts are printed in `Courier` font.

5.3. Tracking library

The tracking layer is implemented as a C library. As described in section 4, there are two different kinds of objects the tracker is supposed to deal with:

- standard objects that do not belong to any particular model
- objects that do belong to a model and that require model parameters to be taken into account when performing the tracking operation

This section describes the localization and tracking principle for standard objects. Section 5.4 will cover constraint localization and tracking in the case of the human model.

For more information about library modularization and implementation, see appendix B.

5.3.1. Overview

When tracking standard objects that do not have any particular spatial or temporal constraints from belonging to a model, the tracking engine uses two criterions to refine the object size and shape:

- color
  the normalized color of the object is used to filter input images and extract possible locations
- temporal depth continuity
  when several objects of the same color are detected in the tracking area, the one that is closest to the previous locations in terms of depth is picked

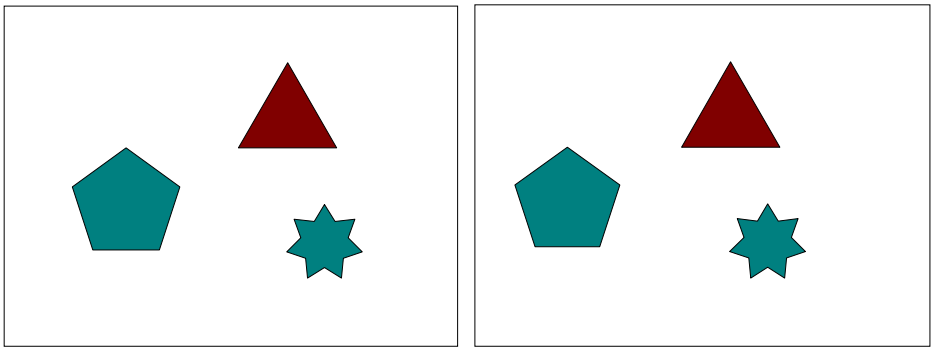
The tracker performs sensor fusion from its two input sources, namely color data and range information. The fused data is represented in a histogram holding the filtered color blobs area versus disparity values. Figure 5.1 describes the fusion process.

The tracker works in two phases:

- object localization
  first, the objects the tracker should track are localized in the image; this stage involves heavy processing of big portions of the images where the object is expected to be found, and is therefore slow.
- object tracking
  once the object is located, only a small area of the image is then processed; this area is determined by the object last known position as well as the object velocity. By processing small portions of the image only, high refresh rates can be obtained.

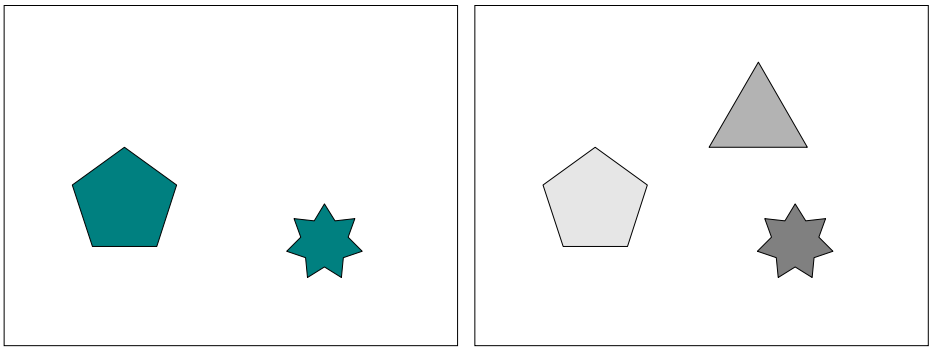
If the object gets too occluded, or the confidence the tracker has in object detection drops below an acceptable threshold, it then tries to localize the object again until it is reacquired.

RAW INPUT
two color images are grabbed from a “left” and a “right” camera



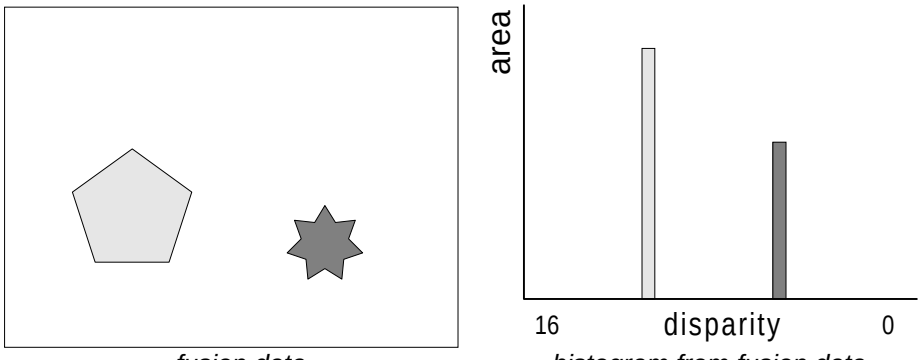
left input image *right input image*

IMAGE PROCESSING
left input image is filtered for a specific normalized color
a disparity map matching the left input image is processed
(darker values indicate lower disparity)



color filtered image *disparity map*

SENSOR FUSION
filtered color data is superimposed on range data
histogram is built from both data



fusion data *histogram from fusion data*

figure 5.1 – sensor fusion process

5.3.2. Standard localization

The tracking library provides a general `tlLocateStdObject()` routine. This function looks for an object with a certain color in a given volume of space. If several objects are detected, the one that is the closest to the camera is picked. The algorithm is presented in figure 5.2.

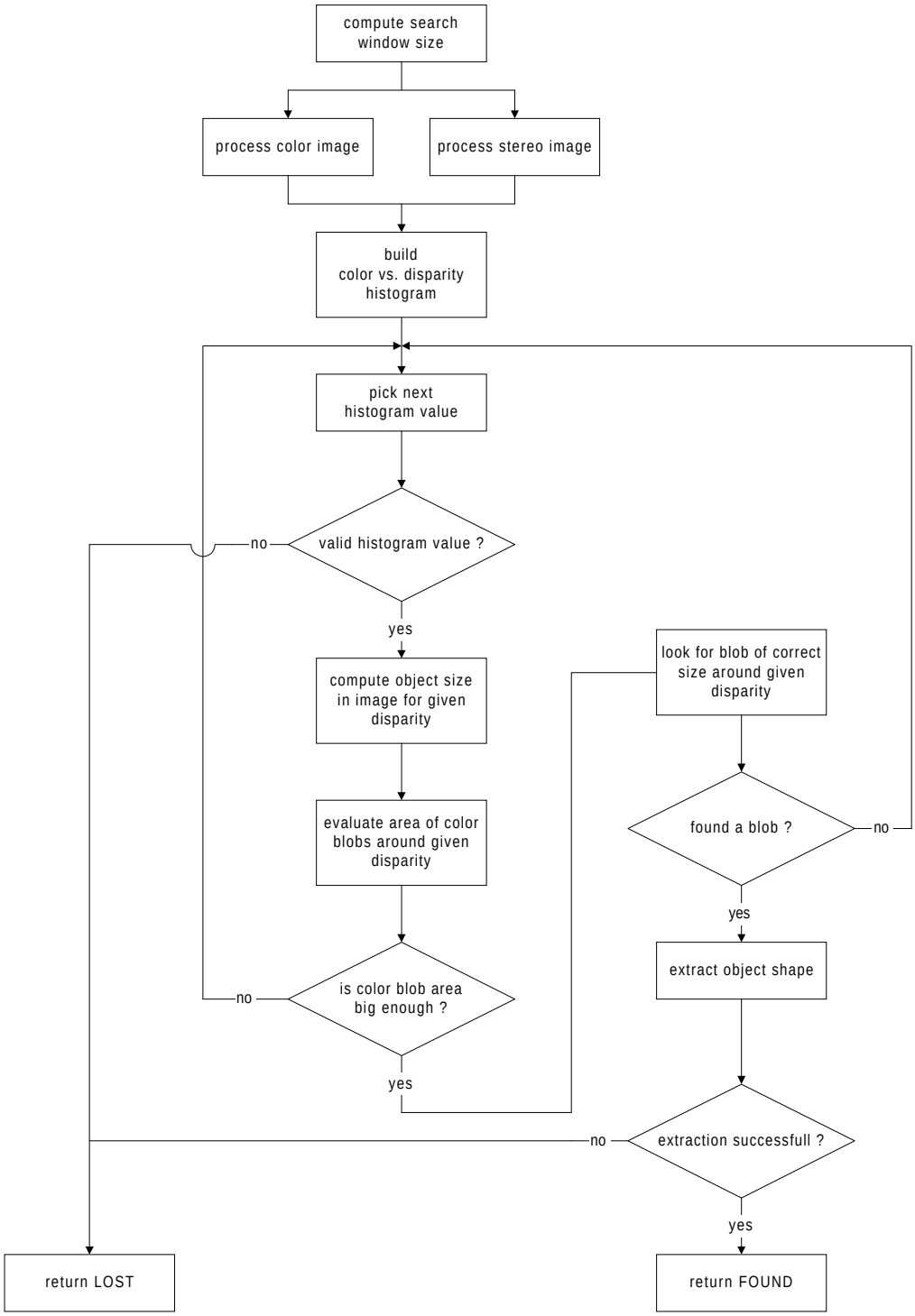
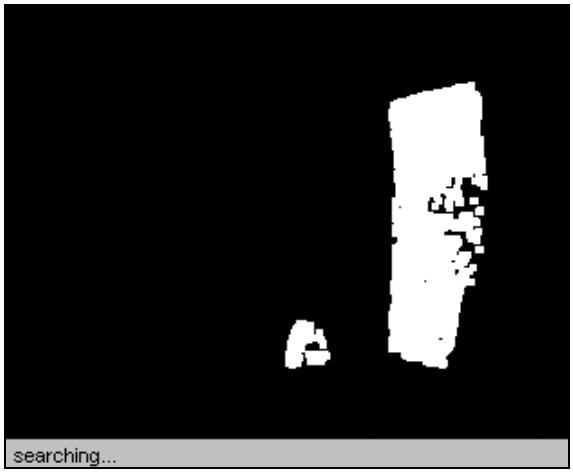


figure 5.2 – standard object localization algorithm

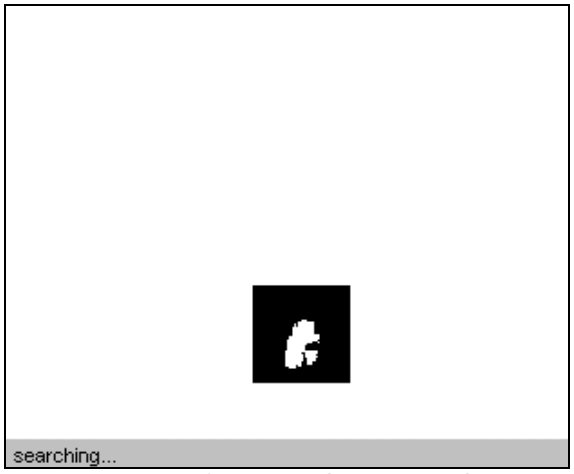
The sequence in figure 5.3 shows the image processing steps to locate an orange juice bottle. Note how range data allows successfully discriminating the door and correctly locating the bottle.

original image

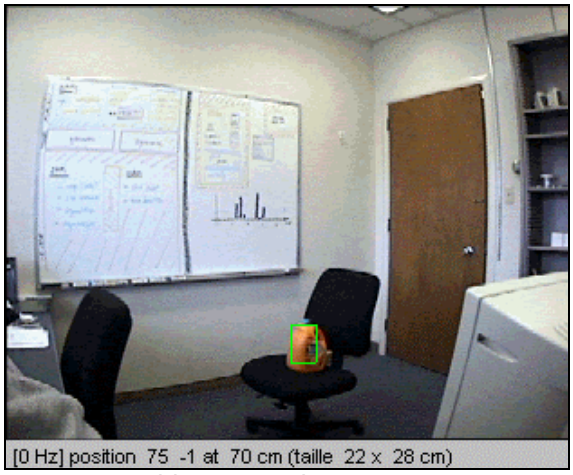
normalized color filter



morphological filter



foreground segmentation



object extraction

figure 5.3 – image processing sequence for object localization

5.3.3. Standard tracking

Once the object is located, the images are only processed in an area (the *tracking window*) surrounding the expected position of the object in order to limit processing time. Using this technique, a refresh rate around 19Hz (fluctuating with object size) is achieved when tracking a single object. The schematic in figure 5.4 shows the algorithm used to track standard objects.

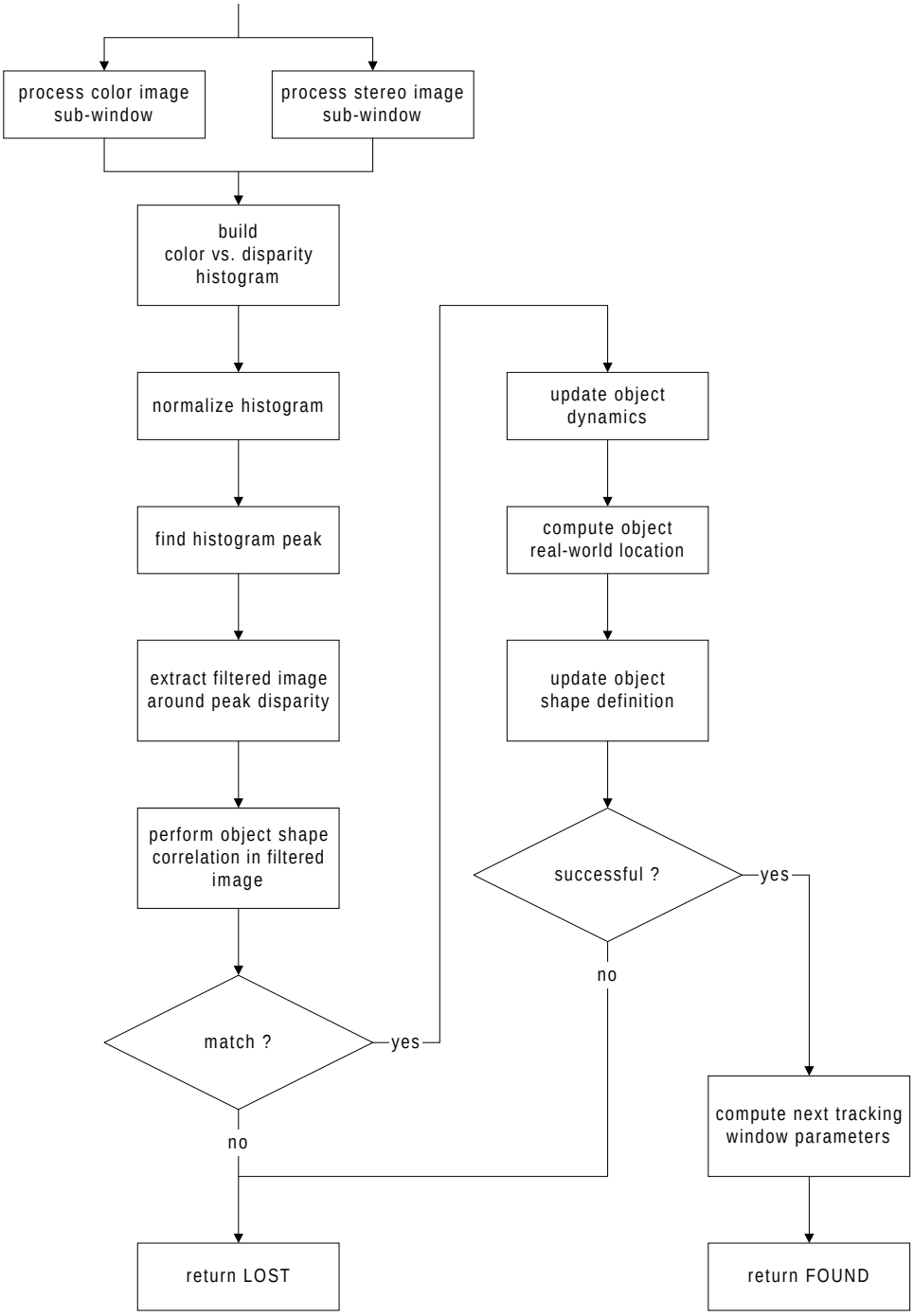
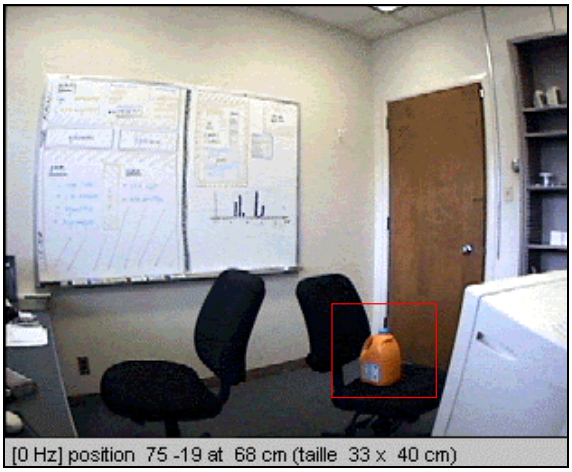
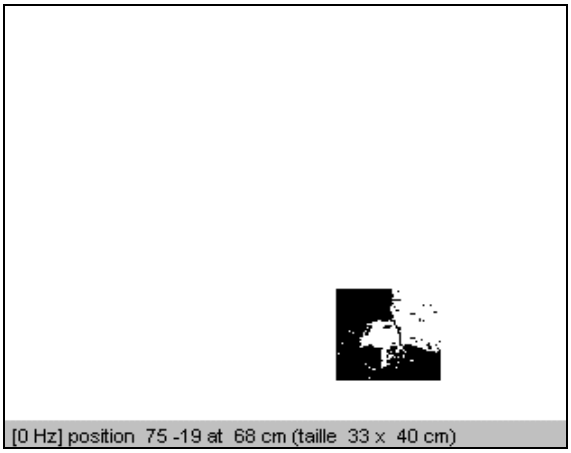


figure 5.4 – standard object tracking algorithm

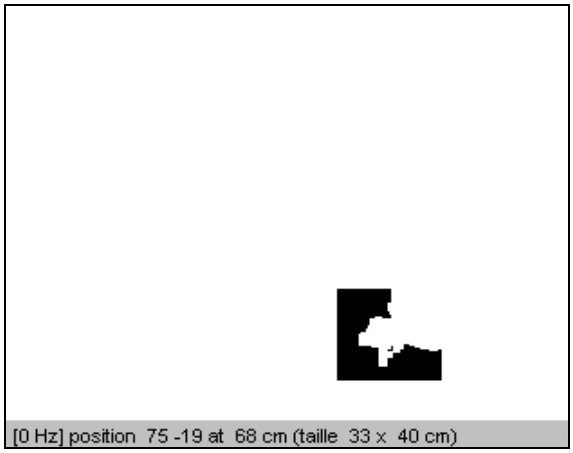
A sample sequence of images from the orange juice bottle object tracking is shown in figure 5.5. Only a sub-window of the image (red square) is processed to achieve high refresh rates. Again, note how range data is extremely useful in segmenting the bottle from the background.



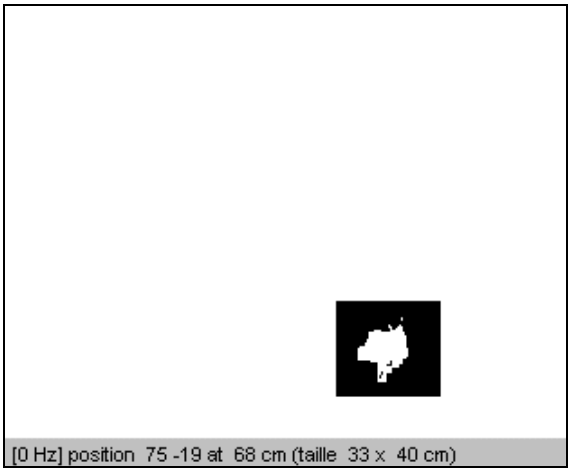
original image (with tracking window)



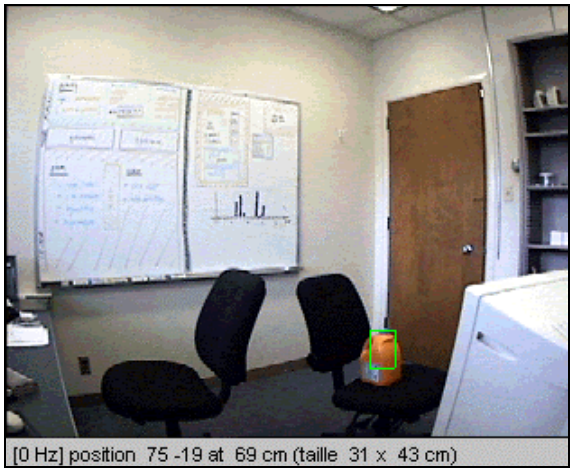
normalized color filter



morphological filter



foreground segmentation



object localization

figure 5.5 – image processing sequence for object tracking

5.4. Human model library

5.4.1. Overview

The human model is implemented as a C++ object oriented library. Its organization is shown in figure 5.6 in UML (Unified Modeling Language) format. For simplicity, only public instances and methods are shown.

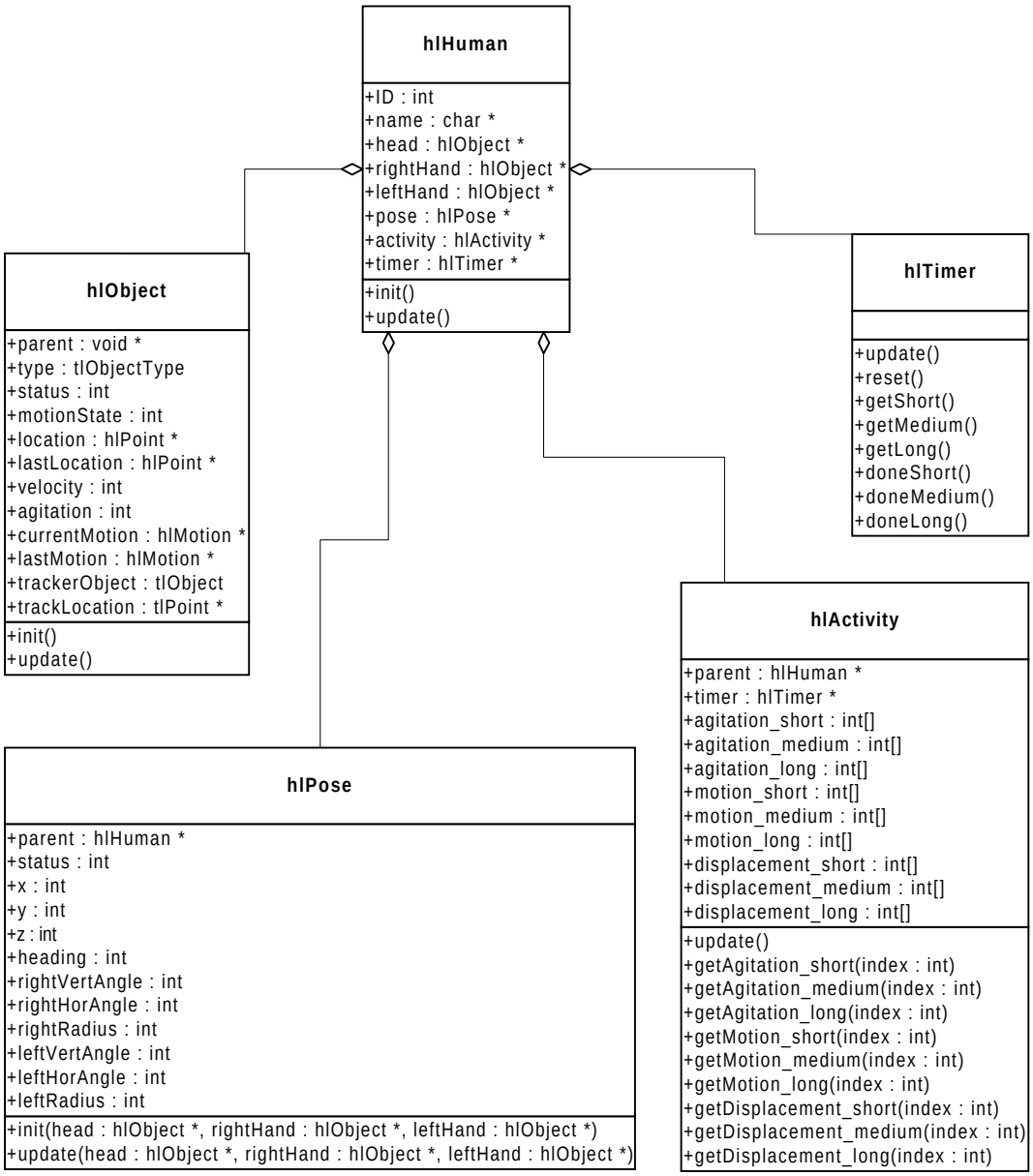


figure 5.6 – UML schematic of the human model library

For more information about each class, see software interfaces in appendix B.

5.4.2. Object class

The object class is designed in a way that makes it very versatile. In order to be flexible, the object implementation has to be:

- tracker independent
- model independent

As described in section 4, the objects of the modeling layer have to be independent from the tracking layer, so that other tracking modalities can be added without having to redesign the object.

Similarly, the object design has to be independent from the model so that it can be used to quickly build another model, or be used by itself without any constraints from a model.

Therefore, objects are assigned a *type* that signals the tracking layer what kind of objects it is dealing with. The type list is implemented by the tracking library to allow for more constrained tracking. Object types recognized by the tracking layer so far are given in table 5.2 below.

object type	description
STD	standard object (does not belong to any model)
HL_HEAD	head object from the human library
HL_R_HAND	right hand object from the human library
HL_L_HAND	left hand object from the human library

table 5.2 – H.O.T. object types

Except for the `STD` type, which specifies that no particular tracking rules are available for the object, all other types labels are composed of a model specific prefix (`HL_`) that identifies the model library used. It is then possible to track the same type of object modeled separately in two different libraries, and using different rules. When an object calls the tracking routines, the tracking library checks the object *type* tag and calls the appropriate routine. For non-standard objects, specific tracking routines are located in the model specific modules mentioned in section 4.5.

The human model library provides modules with tracking routines that are specific to both its geometric constraints and the tracker modalities. Section 5.5 describes constrained tracking in more detail.

5.4.3. Updating the human model

The human model is easy to use in any software. It only needs to be initialized once, and then updated regularly.

Initialization occurs when the `human.init()`method is called; the library then performs the following tasks:

- initialize the timer
- initialize each of its objects (head and hands)
  - create and initialize tracker objects
  - initialize monitoring values
- initialize pose model and activity model

Once the model has been initialized, the state of the human model is updated by calling the `human.update()`method. When asked to update, the human library performs the following tasks:

- update the timer
- update each object
  - call object tracking routines
  - update object dynamics
- update pose model
  - compute pose static parameters
  - update pose dynamics
- update activity model
  - compute current activity
  - perform activity integration

This architecture frees the application using H.O.T libraries from dealing with the tracking and the modeling themselves.

5.5. Interface between libraries

The tracking library has been designed in a modular way so that it can integrate modules corresponding to specific models, on top of the tracking routines for standard, unconstrained objects. These modules have access to all of the tracking engine resources, as well as to the models data structure themselves. They create the links that allow a model to impose constraints on the tracker, while the tracker accesses the model elements that will help it find the object. Typically, a module consists of a set of tracking functions, each one specific to one type of object composing the model. Section 5.5.1 describes shared data structures, while the following sections explain the human model modules in more detail.

5.5.1. Data structure

Both libraries need to share a convention to pass geometric dimensions along. *hmLib* requires *trackLib* to implement two structures, namely a *point* structure and a *box* structure, used to reference respectively a location and a volume in the world coordinate system. These two structures have to provide world frame reference members respecting the following syntax:

```
typedef struct {
    int wx;
    int wy;
    int wz;
    ...
} tlPoint;

typedef struct {
    int wx;
    int wy;
    int wz;
    int wwidth;
    int wheight;
    int wdepth;
    ...
} tlBox;
```

Using this convention, it is possible to use another tracker without affecting the human model code. Note that other members, used by the tracking library only, can be added to the above-mentioned structures; the model library only stores these without ever accessing any member that is not listed above.

Also, each object composing the human model has to contain a data structure holding all the data specific to itself that the tracker requires, along with the result of the tracking action. These two instances of the object class are:

```
tlObject *trackerObject;
tlPoint  *trackLocation;
```

For the exact implementation of the tracker object, see appendix B.

5.5.2. Constrained localization

Principle

To locate an object that belongs to the human model, several geometric constraints can be added to the standard localization routine presented in section 5.2. The goal is to make object localization more robust by rejecting false positives.

The following set of rules is used in human model objects localization, along with the standard localization algorithm:

- head position must be above a certain height from the ground
- a torso must be found below the head
- head size and size ratio must be within certain boundaries
- hand positions respectively to the head must be within certain boundaries
- hand size must be within certain boundaries
- no hands can be found if no head is found

The following constants are used in object localization:

```
/* human model dimensions [cm] */
TL_HEAD_WIDTH                  15
TL_HEAD_HEIGHT                 20
TL_TORSO_WIDTH                 30
TL_TORSO_HEIGHT                60
TL_ARM_LENGTH                 180
TL_HAND_WIDTH                   3
TL_HAND_HEIGHT                  3

/* localization constants */
TL_HL_LOCATE_HEAD_HEIGHT      150
TL_HL_LOCATE_HEAD_DISP          4
TL_HL_LOCATE_HEAD_THRES        90
TL_HL_LOCATE_TORSO_DISP         4
TL_HL_LOCATE_TORSO_THRES       70
TL_HL_LOCATE_HAND_DISP          4
TL_HL_LOCATE_HAND_THRES        95
```

Implementation

The tracker module that implements the localization for the human model library objects is implemented in the `tlModule_hmLib.h` and `module_hmlib_locate.c` files. The routines list and interface are given in appendix B.

5.5.3. Constrained tracking

Principle

Again, specific rules can reinforce the standard object tracking routine if the object tracked belongs to a geometric model. The goal is to make tracking both faster and more reliable. Also, tracking parameters such as image processing filter values and tracking window size can be set for each object with custom values that are more appropriate than the generic ones.

The following set of rules is used in human model objects tracking:

- head size cannot change suddenly
- distance between hand and head cannot be greater than a limit value
- two objects cannot be at the same location (one must be occluded)

The following tracking parameter values are specific to the human library model:

```
/* human model dimensions [cm] */
TL_HEAD_WIDTH                  15
TL_HEAD_HEIGHT                 20
TL_TORSO_WIDTH                 30
TL_TORSO_HEIGHT                60
TL_ARM_LENGTH                 180
TL_HAND_WIDTH                   3
TL_HAND_HEIGHT                  3

/* tracking window extra size */
TL_HEAD_TRACKWIN_X             15
TL_HEAD_TRACKWIN_Y             10
TL_HAND_TRACKWIN_X             20
TL_HAND_TRACKWIN_Y             20

/* morph filters coefficients */
TL_HEAD_ER_COEFF                2
TL_HEAD_DIL_COEFF               6
TL_HAND_ER_COEFF                2
TL_HAND_DIL_COEFF               4

/* tracking constants */
TL_HL_TRACK_HEAD_DISP           2
TL_HL_TRACK_HEAD_THRES         50
TL_HL_TRACK_HAND_DISP           2
TL_HL_TRACK_HAND_THRES         30
```

The other difference with standard object tracking is the way the object is considered lost. If the tracking fails, it first considers the object occluded and tries to locate it again in a slightly bigger sub-window at the next iteration of the tracking program. If nothing is detected again, only then is the object considered lost. This scheme makes the tracker faster and more robust.

Implementation

The tracker module that implements the localization for the human model library objects is implemented in the `tlModule_hmLib.h` and `module_hmlib_track.c` files. The routines list and interface are given in appendix B.

5.6. Software flexibility

Portability

The code for trackLib and hmLib has been written in ANSI C and C++ on a Windows platform. ANSI C and C++ *should* (with all appropriate reserve) compile on every platform.

Platform specific elements are limited to the user interface and have been reduced to a minimum (as they are only intended to help debug the application using the libraries during the development process); they are insulated in separated files.

Hardware specific elements, namely the frame grabber drivers and access routines, are implemented for the ImageNation PXC200 board and compatible. Again, source code for the hardware module is insulated in a separated file.

Provided that the platform specific and hardware specific modules are rewritten with respect to the existing nomenclature and conventions, the libraries are easily ported to another platform, or run on different hardware.

Model flexibility

use with another tracker

Since the implementation of the human model is independent from the tracking modality, it can be used with any arbitrary tracker that returns the geometric position of objects, given that the interface does not change. Any tracker that respects the types and calling convention described above can virtually be used to update the model.

model extensions

As different tracker engines may use different or more modalities, such as sound and speech processing, or can process more objects at the same time, a more complex model could be used which would be more accurate and feature more parameters. The current model design and object-oriented implementation easily allow more features to be added without affecting the existing model parameters. The object-oriented implementation of the model is particularly adapted to modular features management.

6. Results

6.1. Tracking results

head tracking

figure 6.1 – head tracking result

frame rate	19 Hz
description	the white square locates the human being's head tested with 10 humans from different nationalities very reliable

human tracking

figure 6.2 – human tracking result

frame rate	approximately 13 Hz
description	the white square locates the human being's head the red cross or square indicates the right hand the green cross or square indicates the left hand tested with 3 human beings hand localization needs fine tuning few false positives for hand detection looses track when hands move fast because of low refresh rate

human tracking with motion

figure 6.3 – human tracking with motion result

frame rate	approximately 12 Hz
description	the white square locates the human being's head the red cross indicates the right hand the green cross indicates the left hand the red line indicates a linear movement from the right hand the orange line indicates a non-linear movement tested with 3 human beings motion segmentation works well motion modeling works well

6.2. Modeling results

model position

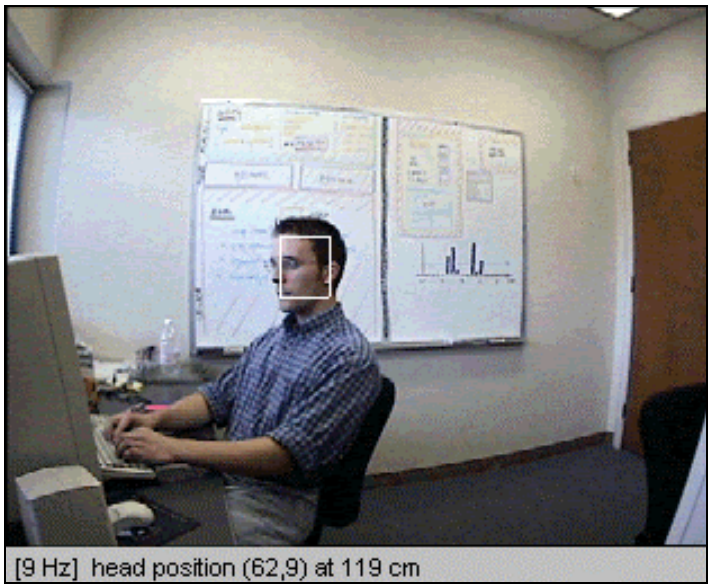


figure 6.4 – human modeling result (position)

The position of the body is derived from the head position.

The position is given in the (X,Y,Z) world frame as shown in figure 4.10. In figure 6.4, head position is (62,9,119) [cm].

model parameters

figure 6.5 – human modeling result (pose)

The model parameters are extracted as described in section 4.3.3.

In figure 6.5, model parameters are:

right hand

radius	39 cm
vert_angle	-62 deg
hor_angle	12 deg

left hand

radius	39 cm
vert_angle	-70 deg
hor_angle	-8 deg

activity monitoring

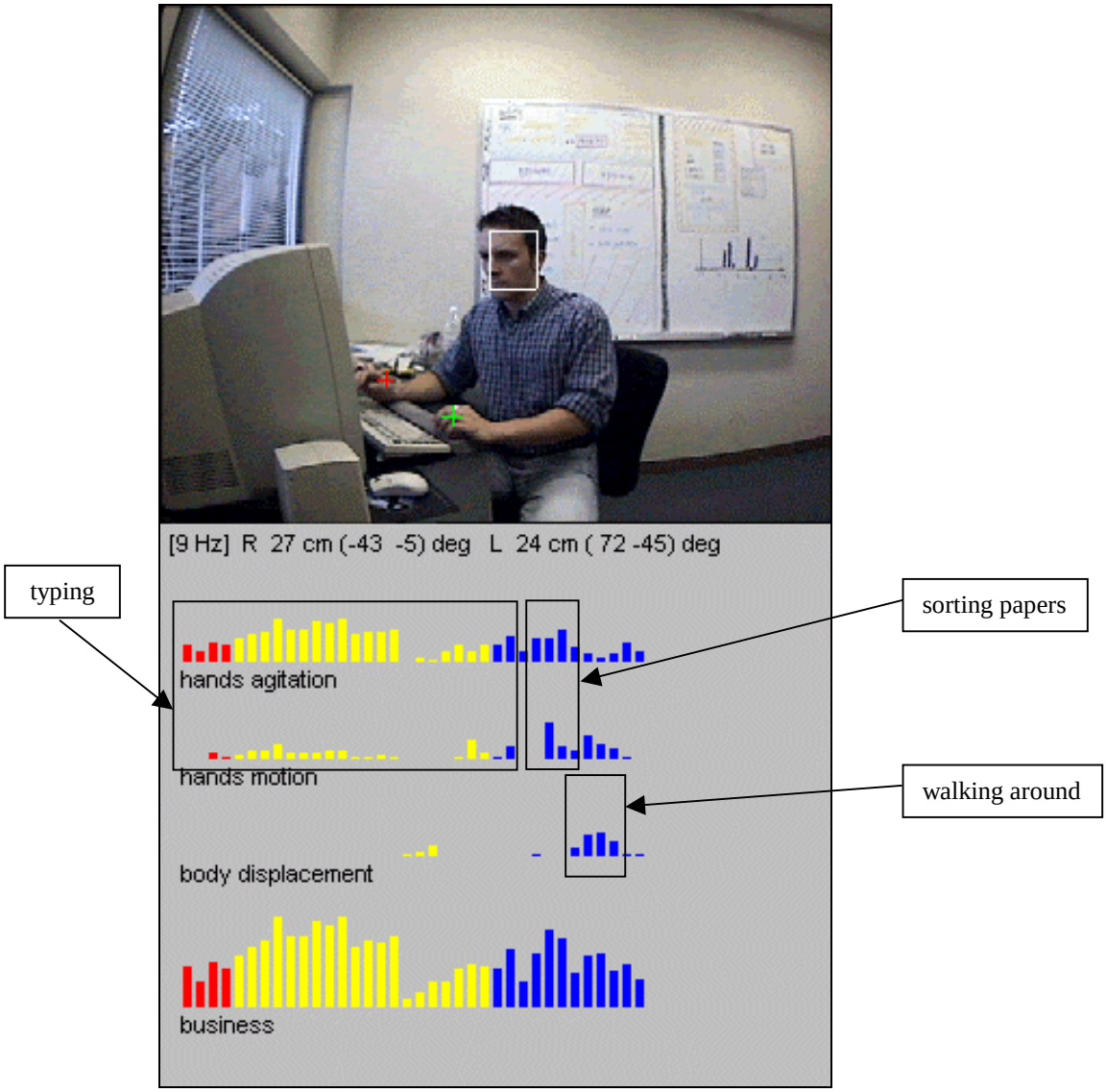


figure 6.6 – human activity monitoring

The model parameters are extracted as described in section 4.3. Figure 6.6 is the result of 15 minutes of monitoring.

6.3. Demo application

A simple HCI application that demonstrates H.O.T. capabilities is currently under development. It will consist of a virtual whiteboard the human being can interact with. Both the geometric model and the activity monitoring features of H.O.T. will be used.

The virtual whiteboard application is a whiteboard that can be projected on any surface (screen displays, walls). Interaction with the whiteboard requires a pointing device, and several different input modalities, such as voice recognition, pen writing or others.

Activity monitoring, along with the position of the human being in the room, will give the application hints about whether the person is trying to interact with the whiteboard or not. The geometric model is then used to extract a pointing vector from the user's right arm, which can be used as a pointing device on the virtual board.

Finally, movement modeling will be used to provide several command features; for instance, if the user makes a horizontal movement with his left hand while pointing at the board, the board will be stored and a new, empty board will appear. A horizontal movement in the opposite direction will restore the previous board on display. Similarly, vertical movements of the left hand can be used to turn the input mode on or off. For example, if the user makes a vertical movement while his right hand points at the board, the computer could turn speech recognition on, and type the spoken text at the location the human is pointing to. When a downward movement is detected, speech recognition is turned off, or a drawing tool is enabled that draws named objects at the pointing location.

Figure 6.7 shows a simple application of the geometric model, where H.O.T. output is used to point at a virtual whiteboard. The top window shows the tracking result, while the bottom window simulates a 4m by 3m wall behind the camera the user is pointing at. The red and green squares on the "wall" are the places the user is designating.

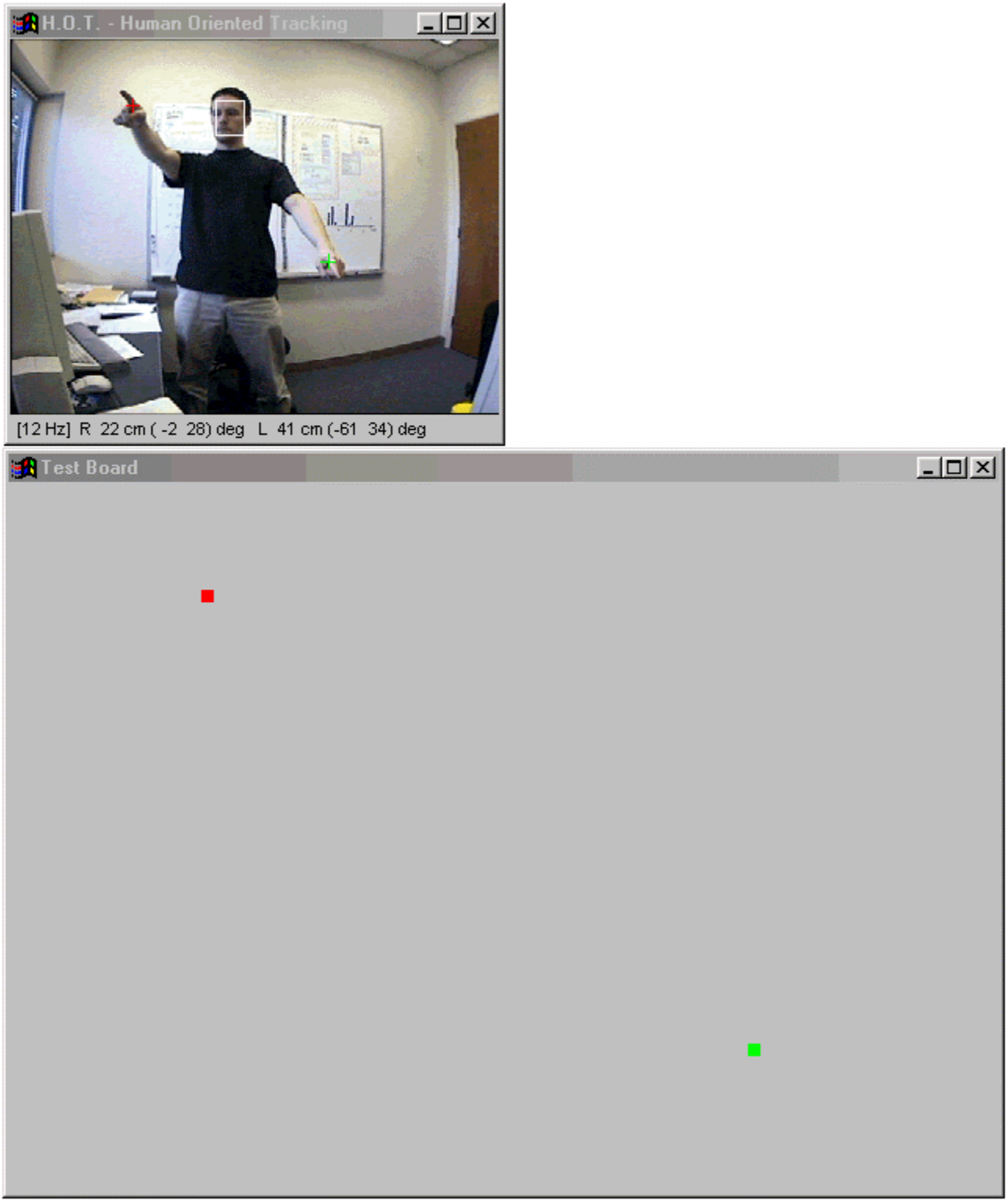


figure 6.7 – using H.O.T. to point at a virtual whiteboard

7. Prospect

7.1. Tracker engine

There are many improvements that can make the tracker more reliable. However, most of them would require more computing power. For the current available hardware, the version of H.O.T. described in this report appears to be the optimum trade-off between robustness and fast tracking.

Infrared sensor modality

As mentioned earlier, an interesting sensing modality when tracking humans is the infrared light sensor. Low-cost infrared sensing devices can be found that allow discriminating “living” blobs from inert ones in an infrared image. This feature would make H.O.T. more robust when tracking a person.

Disparity continuity analysis

For certain human model features, in particular the hands, a more complex localization technique would consist of “tracing” the disparities back to the torso. By doing so, one would virtually locate the arm of the person. The advantages of this technique would be:

- less false positive hand detection
- more reliable left/right hand distinction
- arm localization (would make it easy to enhance the geometric model)
- shoulder axis localization (would help in estimating the person’s orientation in space)

Better object shape correlation

Currently, each object is localized by a simple, binary correlation. With more computing power, one could perform a smarter correlation based on least square differences. Localization would then be more accurate, and chances to loose the object would drop.

Smarter adaptation

One of the first improvements to the tracker engine would consist of making it more adaptive to environment changes. In particular, changes in light conditions have a considerable effect on colors, therefore affecting the color tracker. Simple color adaptation can be performed within certain boundaries. However, smarter adaptation could prove to be tricky, as it also involves hardware filters on the cameras and the frame grabber boards.

7.2. Model

Several features could be added to the model to make activity monitoring more meaningful.

Human identification

As mentioned in section 4.2, color histogram techniques can be used to identify humans that come and go in images. This feature could be an interesting add-on to H.O.T. and would be relatively easy to implement, given the existing sensing modalities. However, human identification, especially in the smart spaces field, is a sensitive issue for obvious privacy reasons, and should be carefully thought about before being used.

Gaze tracking

Research has already been performed in the field of gaze tracking. By including a gaze feature in the human model, H.O.T. could provide an HCI application with the apparent point of interest of the human being, or at least, his direction of gaze. Gaze is a very useful element in attention monitoring, and could be of great interest for vision-based command interfaces.

In the same spirit, but at a higher-level it could be interesting to integrate facial expression analysis to the human model, in order to get some insight about the user's mood. Facial expressions play an important role in the way humans perceive each other's state of mind, and research also exists to have computers visually interpret them.

Sound analysis

Another very useful feature that should be added to the human model is sound. Once sound processing is added to the tracking layer, there are multiple uses for audio input:

- distinguishing sounds from speech
  sounds can carry useful information, but of a significantly different nature than speech; distinguishing between the two is a first important step when integrating sound into H.O.T.
- speech localization
  speech localization could first make the tracker more reliable by confirming the head localization; it could also be used to determine if the person speaking is the one H.O.T. is tracking at the moment or another human being.
- speech processing
  ultimately, speech processing could be performed to process voice commands, or to spot keywords that correlate to human activity.

8. Conclusion

As technology makes ubiquitous computing possible and affordable, making seemingly natural human-computer interaction becomes a growing need. Intelligent environments are being developed, where human actions and tasks are to be facilitated by the machine. To achieve meaningful interaction, the computer must have some perception of the human's position and actions.

This project resulted in H.O.T. (for Human Oriented Tracking), a reusable software layer that is meant to be used as a buffer between an HCI application and a human being:

- provide information about the human
  H.O.T. uses vision-based sensor fusion to build a geometric and dynamic model of the human being, available to the application.
- provide information to the human
  H.O.T. also monitors how busy the human being is; this activity monitoring can be used to deliver information to the person at the right time and in the appropriate way.

H.O.T. is implemented as 3 independent layers, namely:

- a sensing layer
  a tracking engine using color vision and stereovision is used to locate and keep track of human features (the head and the hands)
- a modeling layer
  a geometric model of the human body is built in real-time; each movement of the features tracked are segmented and modeled as well
- an activity monitoring layer
  a 3-parameters model of human activity is extracted and integrated over time

H.O.T. performances have been evaluated in an office environment. It offers very reliable head tracking, and reliable hand tracking in most conditions. Meaningful activity monitoring was achieved, even though limited by the simplicity of the model and the available processing power. A sample application was described that would make use of H.O.T. features.

Future enhancements to H.O.T. could include more sensing modalities, such as infrared vision and sound processing, and an enhanced model of the human being including speech recognition and more detailed activity monitoring. Currently, performances are limited by the hardware capabilities.

9. Acknowledgements

I would like to thank Dr. Charles Baur and Mr. Reymond Clavel, who made this Californian experience possible. My thanks also to Didier Guzzoni who helped me in more ways than one during my stay, and to Terry Fong for his support and guidance.

There are several people at SRI I am grateful to, in particular Kurt Konolige for his expert advice on this project's orientation, and David Beymer who helped me with the stereo device. I also wish to thank Luc Julia, manager of the CHIC! group at the time of my stay, for giving me freedom in my direction on this project.

Finally I would like to thank my family and friends, here and abroad, for their permanent, unconditional support throughout my studies.

Menlo Park, 25th of February 2000

Sebastien Grange

10. References

[1] M.S. Brandstein, J. E. Adcock, and H. F. Silverman, "A Practical Time-Delay Estimator for Localizing Speech Sources with a Microphone Array", Computer Speech and Language, vol. 9, pp. 152-169, 1995.

[2] K. D. Martin, “Estimation Azimuth and Elevation from Interaural Differences”, on-line downloadable publication, MIT Media Lab, E15-401, Cambridge, MA, 02139, 1995.

[3] S. G. Goodridge, “Multimedia Sensor Fusion for Intelligent Camera Control and Human-Computer Interaction “, dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering, Raleigh, NC, 1997.

[4] J. Achenbach, “FUTURE PERFECT - Your house is about to get very smart. Ready?”, Washington Post Page H01, Friday October 8 1999.

[5] S. Shafer, J. Krumm, B. Brummit, B. Meyers, M. Czerwinski, D. Robbins, “The new EasyLiving project at Microsoft Research”, Joint DARPA/NIST Smart Spaces Workshop, Gaithersburg, Maryland, July 30-31, 1998.

[6] S. L. Oviatt, “Multimodal Interfaces”, in *Proceedings of Conference on Human Factors in Computing Systems (CHI '96)* (New York: ACM Press), pp. 95-102, 1996.

[7] C. Wren, A. Azarbayejani, T. Darrell, A. Pentland, “Pfinder: Real-Time Tracking of the Human Body”, in *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI’97)*, July 1997.

[8] T. Starner, A. Pentland, “Real-time American Sign Language recognition from video using hidden markov models”, in *Proceedings of International Symposium on Computer Vision (ICCV’95)* (IEEE Computer Society Press), Coral Gables, FL, USA, 1995.

[9] N. Jojic, M. Turk, T. S. Huang, “Tracking Self-Occluding Articulated Objects in Dense Disparity Maps”, in *Proceeding of International Conference on Computer Vision (ICCV’99)*, pp. 123-130, Korfu, Greece, September 1999.

[10] D. Beymer, K. Konolige, “Real-Time Tracking of Multiple People Using Stereo and Correlation”, submitted to *International Conference on Computer Vision (ICCV’99)*, Korfu, Greece, September 1999.

[11] T. Darrel, G. Gordon, M. Harville, J. Woodfill, “Integrated Person Tracking using Stereo, Color, and Pattern Detection”, in *Proceeding of Conference on Computer Vision and Pattern Recognition (CVPR’98)*, pp. 601-609, Santa Barbera, June 1998.

[12] N. Jojic, J. Gu, H. C. Shen, T. S. Huang, “Computer Modeling, Analysis and Synthesis of dressed Humans”, in *IEEE Transactions on Circuits and Systems for Video Technology*, March 1999.

[13] A. Azarbayejani, A. Pentland, “Real-Time Self-calibrating Stereo Person Tracking Using 3-D Shape Estimation from Blob Features”, in *Proceedings of 13th International Conference for Pattern Recognition (ICPR’96)* (IEEE Computer Society Press), Vienna, Austria, August 1996.

[14] L. Schomaker, J. Nijtmans, A. Camurri, F. Lavagetto, P. Morasso, C. Benoît, T. Guiard-Marigny, B. Le Goff, J. Robert-Ribes, A. Adjoudani, I. Defée, S. Münch, K. Hartung, J. Blauert, “A Taxonomy of Multimodal Interaction in the Human Information Processing System”, report of the ESPRIT Project 8579, 1995.

[16] R.E. Kalman, “A New Approach to Linear Filtering and Prediction Problems”, in *Transactions of the ASME – Journal of Basic Engineering*, pp. 34-35, 1960.

[17] R.G. Brown, P.Y. Hwang, “Introduction to Random Signals and Applied Kalman Filtering”, Second Edition, John Wiley & Sons Inc., 1992.

[18] K. Konolige, “Small Vision Systems: Hardware and Implementation”, Eighth International Symposium on Robotics Research, Hayama, Japan, October 1997.

[19] N. Badler, M. J. Hollick, J. P. Granieri, "Real-Time Control of a Virtual Human Using Minimal Sensors", in *Forum short paper in Presence 2(1)*, pp. 82-86, MIT, 1993.

[20] L. Emering, R. Boulic, S. Balcisoy, D. Thalmann, “Real-Time Interactions with Virtual Agents Driven by Human Action Identification”, First ACM Conference on Autonomous Agents'97, Marina Del Rey, 1997.

[21] M. C. Cohen, “Design Principles for Intelligent Environments”, *Proceedings of the 1998 National Conference on Artificial Intelligence (AAAI98)*, 1998.

[22] M. J. Swain, D. H. Ballad, "Color Indexing", *International journal of computer vision*, Vol. 7, No. 1, pp.11 - p.32, 1991.

[23] R. Brooks, C. Breazeal(Ferrell), R. Irie, C. Kemp, M. Marjanovic, B. Scassellati, M. Williamson, "Alternative Essences of Intelligence", *Proceedings of AAAI98*, pp. 961-967, Madison WI., 1998.

[24] R. A. Brooks, C. Breazeal, M. Marjanovic, B. Scassellati, M. Williamson, “The Cog Project: Building a Humanoid Robot”, to appear in a *Springer-Verlag Lecture Notes in Computer Science* Volume.

[25] C. Poynton, “A Technical Introduction to Digital Video”, published by John Wiley & Sons, ISBN 0-471-12253-X, 1996.

[26] J. Yang, A. Waibel, “A Real-Time Face Tracker”, *Proceedings of the 1996 Workshop on Applications of Computer Vision (WACV'96)*, Sarasota, Florida, USA, December 1996.

[27] A. F. Bobick, J. W. Davis, “Real-Time Recognition of Activity using Temporal Templates”, *Proceedings of the 1996 Workshop on Applications of Computer Vision (WACV'96)*, Sarasota, Florida, USA, December 1996.

APPENDIX A - HARDWARE

Color camera

Board Camera Model **CMH524-L25**

specifications	
TV System	NTSC
Image Format	1/4" CCD 380000 pixels
Eff. Picture Elements	768(H) x 494(V)
Scanning System	2:1 Interlaced
Sync. System	Automatic internal/L.L. selection
Scannig Frequency	15.734KHz (H) / 59.94Hz (V)
Resolution	480 (H) TV Lines at center
S/N Ratio	More than 48dB (AGC-OFF)
Sensitivity	3 lux
Gamma Characteristic	γ=0.6 / γ=1.0 selectable by dip switch
Video Output Signal	VBS Composite (1.0Vp-p) and YC-output
Automatic Gain Control	ON/OFF by dip switch (0-20dB)
Electronic Shutter	1/60 - 1/100,000 linear or 1/60 1/100, 1/250, 1/500, 1/1000, 1/2000, 1/4000, 1/10,000 selectable
White Balance	Auto tracking
Power Source	AC24V ±10% (CMH524)
Power Consumption	2.2W (12VDC) or 4.5W (24VAC)
Operating Temperature	-10 ~ +60° C
External Dimension	42(W) x 42(H)mm double board (CMH524)
Input/Output Terminal	24VAC power board: 24VAC input 4-pin plug DC12VDC & L.L. pulse output: 3-pin plug
Dip Switches	AGC/Flickerless/BLC/Gamma (ON/OFF) WB-Fix/AES-DC (AES speed selectable)

table A.1 – color camera specifications

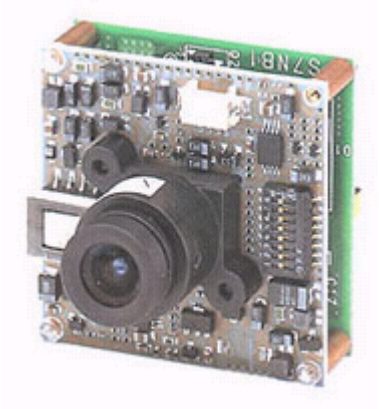
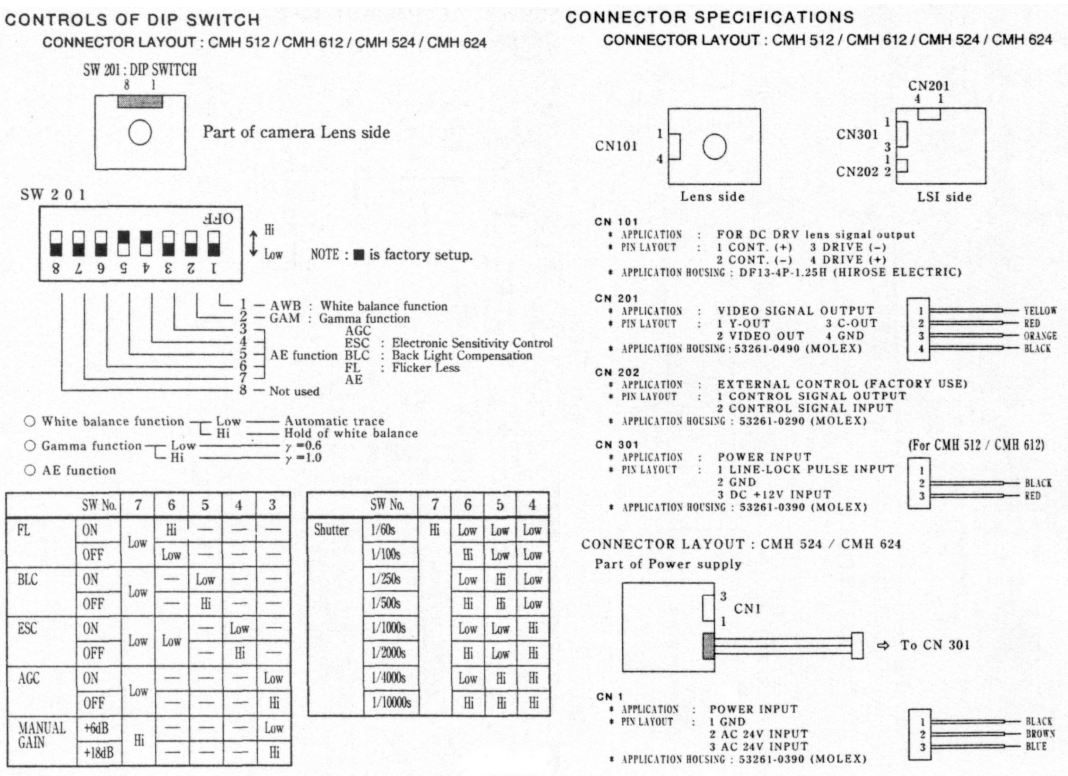


figure A.1 – color camera



	SW No.	7	6	5	4	3
FL	ON	Low	Hi	—	—	—
	OFF		Low	—	—	—
BLC	ON	Low	—	Low	—	—
	OFF		—	Hi	—	—
ESC	ON	Low	Low	—	Low	—
	OFF			—	Hi	—
AGC	ON	Low	—	—	—	Low
	OFF		—	—	—	Hi
MANUAL GAIN	+6dB	Hi	—	—	—	Low
	+18dB		—	—	—	Hi

	SW No.	7	6	5	4
Shutter	1/60s	Hi	Low	Low	Low
	1/100s		Hi	Low	Low
	1/250s		Low	Hi	Low
	1/500s		Hi	Hi	Low
	1/1000s		Low	Low	Hi
	1/2000s		Hi	Low	Hi
	1/4000s		Low	Hi	Hi
	1/10000s		Hi	Hi	Hi

figure A.2 – color camera settings and schematic

APPENDIX B – SOFTWARE INTERFACE

Human model library

file list

header	implementation	description
hmLib.h	-	library header
hlHuman.h	human.cpp	human model class
hlObject.h	object.cpp	model objects class (used for head and hands)
hlMotion.h	motion.cpp	motion class (attached to each object)
hlPose.h	pose.cpp	geometric model class
hlActivity.h	activity.cpp	activity model class
hlTimer.h	timer.cpp	human timer class (used for activity integration)

table B.1 – human model library file list

The following pages are the C++ interface of each software module.

hmLib.h

```
// features to include in modeling
#define HL_DO_FULL            // full modeling (as opposed to head tracking only)
#define HL_DO_MOTION          // include motion detection
#define HL_DO_ACTIVITY        // include activity modeling

// activity requires motion modeling
#ifdef HL_DO_ACTIVITY
  #define HL_DO_MOTION
#endif

// ANSI C includes
#include <iostream.h>
#include <math.h>

// tracker lib
#include "tracklib.h"

// math constant
#define HL_PI 3.141592


// point type
typedef struct {
    int wx,wy,wz;             // world frame coordonates
    int ix,iy,id;             // temp - image frame coord
                              // because invert proj func (world2image) doesn't exist yet
} hlPoint;


// class protos
class hlMotion;
class hlObject;
class hlPose;
class hlActivity;
class hlHuman;

// headers
#include "hlTimer.h"
#include "hlHuman.h"
#include "hlMotion.h"
#include "hlObject.h"
#include "hlPose.h"
#include "hlActivity.h"
```

hlHuman.h

```
// human status list
#define HL_HUMAN_INIT         0      // tracking parameters not initialized
#define HL_HUMAN_OK           1      // tracking works
#define HL_HUMAN_OCCLUDED     2      // human is partially occluded
#define HL_HUMAN_LOST         3      // tracker lost track


class hlHuman
{

public:
    // identification members
    int ID;
    char name[255];
    int status;

    // timer
    hlTimer *timer;

    // visual tracking objects
    hlObject *head;
    hlObject *rightHand;
    hlObject *leftHand;

    // body pose object
    hlPose *pose;

    // activity object
    hlActivity *activity;

public:
    // constructors
    hlHuman( void );
    hlHuman( int newID );
    hlHuman( int newID, char *new_name );

    // destructor
    ~hlHuman( void );

    // update human state
    int init( void );
    int update( void );

};
```

hlObject.h

```
// object status list
#define HL_OBJECT_INIT               0      // object not initialized
#define HL_OBJECT_OK                 1      // tracking is working
#define HL_OBJECT_OCCLUDED           2      // object is occluded
#define HL_OBJECT_LOST               3      // tracker lost track of the object

// motion management constants
#define HL_OBJECT_NO_MOTION          0      // no ongoing motion
#define HL_OBJECT_MOTION             1      // ongoing motion

// position update damping
#define HL_OBJECT_POSITION_DAMP      0.5    // damping for object position changes

// agitation update damping
#define HL_OBJECT_AGITATION_DAMP     0.6    // damping for object agitation changes

// velocity management constant
#define HL_OBJECT_VELOCITY_DAMP      0.3    // velocity computation damping
#define HL_OBJECT_VELOCITY_THRESHOLD 15     // velocity threshold in [cm/iteration]
                                            // to discriminate motion vs. static
class hlObject
{

public:
    // object status and position
    int status;
    hlPoint *location;
    hlPoint *lastLocation;           // last known position

    // dynamic properties
    int velocity;
    int agitation;

    // parent object
    void *parent;

    // motion management
    int motionState;
    hlMotion *currentMotion;
    hlMotion *lastMotion;

    // tracker parameters (from tracking library)
    tlObject *trackerObject;         // tracker object
    tlObjectType type;               // object type
    tlPoint *trackLocation;          // object location in tracker format

public:
    // constructors
    hlObject( void );
    hlObject( tlObjectType type );
    hlObject( void *parent );
    hlObject( void *parent, tlObjectType type );

    // destructor
    ~hlObject( void );

    // update
    int init();
    int update( void );

private:
    // private tracking related methods
    int locate( void );
    int track( void );

    // compute scalar velocity of the object
    void computeVelocity( void );
};
```

hlMotion.h

```
// motion status
#define HL_MOTION_FINISHED    0      // motion is finished
#define HL_MOTION_IN_PROGRESS 1      // motion model is still being updated
#define HL_MOTION_LOST        2      // motion doesn't fit the linear model
                                     // modeling stopped, but motion not finished yet
#define HL_MOTION_RESET       3      // no data available (just initialized or reset)

// biggest number represented by the 'double' type (IEEE standard)
#define HL_DBL_INFINITY       1.7976931348623157E+308

// linear motion consistency constants
#define HL_MOTION_MAX_OFF_ANGLE 15    // maximum angle deviation allowed [deg]
#define HL_MOTION_MAX_OFF_COUNT 3     // maximum number of consecutive deviations
                                      // from linearity allowed
#define HL_MOTION_MIN_ITERATION 10    // minimum amount of iteration before
                                      // checking for consistency

/*
 * KALMAN FILTER
 * constant scalar kalman filter
 */

// motion kalman filter typedef
typedef struct {
    double s, s_;             // state and state estimate
    double P, P_;             // estimate error covariance (a posteriori and a priori)
    double Q, R;              // process and measurement covariances
    int off_counter;          // successive points off the linear model counter
} hlMotionKalmanStruct;

/*
 * below, AXY, AXZ, AYZ represent the motion projection angles in the
 * world frame expressed in [deg]
 */

/* initial estimate covariance
 * represents how much we trust the initial value we give the filter */
#define HL_MOTION_AXY_P 0.1
#define HL_MOTION_AXZ_P 0.1
#define HL_MOTION_AYZ_P 0.1

/* process covariance
 * represents the covariance of the actual system state we are filtering;
 * it also is a nice way to compensate for model inaccuracy
 * (see documentation on kalman filter for details) */
#define HL_MOTION_AXY_Q 0.2
#define HL_MOTION_AXZ_Q 0.1
#define HL_MOTION_AYZ_Q 0.1

/* measurement covariance
 * (see documentation on kalman filter for details) */
#define HL_MOTION_AXY_R 0.3
#define HL_MOTION_AXZ_R 0.2
#define HL_MOTION_AYZ_R 0.2
```

```
class hlMotion
{

public:
    // waypoint data
    hlPoint *startPoint;
    hlPoint *lastPoint;
    hlPoint *endPoint;

    // status data
    int linear;
    int status;
    int iteration;

    // Kalman filter state elements
    int axy,axz,ayz;   // motion projection angles in the world frame

private:
    // Kalman filter variables
    hlMotionKalmanStruct kaxy;        // angle in the Oxy plane (Kalman struct)
    hlMotionKalmanStruct kaxz;        // angle in the Oxz plane (Kalman struct)
    hlMotionKalmanStruct kayz;        // angle in the Oyz plane (Kalman struct)

public:
    // constructors
    hlMotion( void );

    // destructor
    ~hlMotion( void );

    // init motion data
    void init( hlPoint *point0, hlPoint *point1 );

    // update motion data
    void update( hlPoint *point );

    // reset motion
    void reset( void );

    // finish up the motion
    void finish( hlPoint *point );

    // place a copy of the motion data in the motion argument
    void copyTo(hlMotion *dest);

    // debug only - displays a window with the motion parameters
    void showParams( void );

private:
    // initialize single constant kalman filter
    void initialize_kalman_filter( hlMotionKalmanStruct *kalman,
                                   double s_init,
                                   double P_init,
                                   double Q_init,
                                   double R_init );

    // motion specific kalman filter
    void kalman_filter( hlMotionKalmanStruct *kalman, double input );

    // make sure the assumption of linear motion is still valid
    int consistent_model( hlMotionKalmanStruct *kalman, int input );
};
```

hlPose.h

```
// pose status list
#define HL_POSE_INIT   0
#define HL_POSE_OK     1

// dynamic variable smoothing constant
#define HL_POSE_COORD_DAMP           0.5
#define HL_POSE_HEADING_DAMP         0.7
#define HL_POSE_ANGLE_DAMP           0.5
#define HL_POSE_RADIUS_DAMP          0.5


class hlPose
{

public:
    // parent class
    hlHuman *parent;

    // status
    int status;

    // global position
    int x,y,z;                // location of the person in the world frame
    int heading;              // orientation of the person in the world frame

    // model static parameters:
    // angle of proj in vert body plane, in hor plane, and radius
    int rightVertAngle, rightHorAngle, rightRadius;
    int leftVertAngle, leftHorAngle, leftRadius;

private:
    // optimizers
    double sinHeading, cosHeading;

    // past state memorization
    int lastx, lasty, lastz;
    int lastHeading;
    int lastRightVertAngle, lastRightHorAngle, lastRightRadius;
    int lastLeftVertAngle, lastLeftHorAngle, lastLeftRadius;

    // model dynamic parameters
    int dx, dy, dz;
    int dHeading;
    int dRightVertAngle, dRightHorAngle, dRightRadius;
    int dLeftVertAngle, dLeftHorAngle, dLeftRadius;

public:
    // constructors
    hlPose( void );
    hlPose( hlHuman *human );
    hlPose( hlHuman *human, int x, int y, int z );

    // destructor
    ~hlPose( void );

    // public methods
    int init( hlObject *head, hlObject *leftHand, hlObject *rightHand );
    int update( hlObject *head, hlObject *leftHand, hlObject *rightHand );

private:
    // change model heading and pose
    int updateHeading( hlObject *head, hlObject *leftHand, hlObject *rightHand );
    int updatePose( hlObject *head, hlObject *rightHand, hlObject *leftHand );
    int updateDynamics( hlObject *head, hlObject *leftHand, hlObject *rightHand );
};
```

hlActivity.h

```
class hlActivity
{

public:
    // human to be monitored
    hlHuman *parent;

    // agitation meters
    int agitation_short  [HL_TIMER_SHORT];
    int agitation_medium [HL_TIMER_MEDIUM];
    int agitation_long   [HL_TIMER_LONG];

    // motion meters
    int motion_short  [HL_TIMER_SHORT];
    int motion_medium [HL_TIMER_MEDIUM];
    int motion_long   [HL_TIMER_LONG];

    // motion meters
    int displacement_short  [HL_TIMER_SHORT];
    int displacement_medium [HL_TIMER_MEDIUM];
    int displacement_long   [HL_TIMER_LONG];

private:
    hlTimer *timer;           // monitoring timer (imported from human)
    hlPose *pose;             // pose model (imported from human)

public:
    // constructor
    hlActivity( hlHuman *parent );

    // destructor
    ~hlActivity( void );

    // activity update method
    int update( void );

    /*
     * data access
     */

    int getAgitation_short( int index );
    int getAgitation_medium( int index );
    int getAgitation_long( int index );

    int getMotion_short( int index );
    int getMotion_medium( int index );
    int getMotion_long( int index );

    int getDisplacement_short( int index );
    int getDisplacement_medium( int index );
    int getDisplacement_long( int index );

    int getActivity_short( int index );
    int getActivity_medium( int index );
    int getActivity_long( int index );

private:
    // update functions
    int agitationEval( void );
    int motionEval( void );
    int displacementEval( void );
};
```

hlTimer.h

```
// timer constants
#define HL_TIMER_SHORT        4      // seconds of short term monitoring unit
                                     // to compose a medium term unit
#define HL_TIMER_MEDIUM       20     // number of medium term monitoring units
                                     // to compose a long term unit
#define HL_TIMER_LONG         20     // number of long term monitoring unit to memorize
#define HL_TIMER_MOD          HL_TIMER_SHORT*HL_TIMER_MEDIUM*HL_TIMER_LONG


class hlTimer
{
public:

private:
    DWORD refTime;            // reference time
    int shortT;               // short term units
    int mediumT;              // medium term units
    int longT;                // long term units

    char shortFlag;           // end of short term period flag
    char mediumFlag;          // end of medium term period flag
    char longFlag;            // end of long term period flag

public:
    // constructor
    hlTimer( void );

    // destructor
    ~hlTimer( void );

    // data access
    int getShort( void )  { return shortT;  };
    int getMedium( void ) { return mediumT; };
    int getLong( void )   { return longT;   };

    // check for end of period ?
    int doneShort( void );
    int doneMedium( void );
    int doneLong( void );

    // timer management
    void reset();
    void update();
};
```

Tracking library

file list

header	implementation	description
trackLib.h	-	library header
tlCommon.h	common.c	type management and library initialization routines
tlFilter.h	filter.c	image processing
tlStereo.h	stereo.c	stereo processing
tlTimer.h	timer.c	time management
tlLocate.h	locate.c	object localization
tlTrack.h	track.c	object tracking
tlUtility.h	utility.c	miscellaneous utilities
tlModule_hmLib.h	module_hmlib_locate.c module_hmlib_track.c module_hmlib_utility.c	human model library specific module
*tlDisplay.h**	display.c	display management and platform specific drawing routines
*tlGrab.h**	grab.c	frame grabber management

* platform specific modules

table B.2 – tracking library file list

The tracking library also uses *visLib*, an image processing library developed for color tracking with 2D images and available through the GNU public license.

The following pages are the headers of each tracking library module.

trackLib.h

```
/************************ headers ****************************************/


/* debug flags */
#define TL_DEBUG
/* #define TL_DEBUG_STOP */
#ifdef TL_DEBUG_STOP
  #define TL_DEBUG
#endif

/* external C libraries:
   visLib 1.8 for Windows
   svs 1.4 for Windows */
extern "C"{
    #include "vislib.h"       /* image processing and 2D tracking library */
    #include "svs.h"          /* stereo processing library */
}


/************************ constants ****************************************/


/* exit code for event manager */
#define TL_EXIT 2

/* object status */
#define TL_LOST       -1
#define TL_OK          0
#define TL_INIT       -2
#define TL_OCCLUDED   -3
#define TL_BYPASS     -4


/************************ types ********************************************/


/* different objects types */
typedef enum { STD, HL_HEAD, HL_L_HAND, HL_R_HAND } tlObjectType;

/* point data struct */
typedef struct{
    int wx,wy,wz;      /* point coord in the world frame */
    int ix,iy,id;      /* point coord in the image */
} tlPoint;

/* tracking object struct */
typedef struct {
    vlObject *object;         /* visLib object to feed the 2D tracker */
    svsSP *svsParam;          /* parameter structure for the stereo module */
    vlWindow *window;         /* tracking window */
    HANDLE stereoMutex;       /* mutex for multithread control of SVS processing */
} tlObject;

/* volume struct */
typedef struct{
    int wx, wy, wz;                  /* location in the world frame */
    int ix, iy, id;                  /* location in the image */
    int wwidth, wheight, wdepth;     /* world length on the X, Y and Z axis */
    int iwidth, iheight, idisp;      /* image length on the X, Y and disp axis */
} tlBox;
```

```
/*********************** includes ****************************************/


/* library headers */
#include "tlCommon.h"
#include "tlDisplay.h"
#include "tlUtility.h"
#include "tlGrab.h"
#include "tlStereo.h"
#include "tlLocate.h"
#include "tlTrack.h"
#include "tlFilter.h"
#include "tlTimer.h"

/* modules from model libraries to include */
#include "tlModule_hmLib.h"
```

tlCommon.h

```
/* useful macro */
#define TL_ERROR tlShowMsg

/* image constants */
#define TL_IMAGE_CENTER_Y VL_ROWS_DEFAULT/2
#define TL_IMAGE_CENTER_X VL_COLS_DEFAULT/2

/* camera constants */
#define TL_PIXEL_uM    10.0e-4        /* 10.0 microns per pixel */
#define TL_THETA       0              /* camera angle from horizontal in degrees */
#define TL_COS_THETA   1.0
#define TL_SIN_THETA   0.0
#define TL_CAM_HEIGHT  120.0          /* camera elevation from the ground */

/* globals */
extern vlImage *leftImage;
extern HANDLE leftImageMutex;
extern vlImage *rightImage;
extern HANDLE rightImageMutex;
extern vlImage *stereoImage;


/*
 * convenience macros for creating defaults
 */
#define TL_FREE(a)  if(a)     { free(a); a=NULL; }
#define TL_POINT_CREATE()     tlPointCreate(0,0,0)
#define TL_OBJECT_CREATE()    tlObjectCreate(0,0,VL_COLS_DEFAULT,VL_ROWS_DEFAULT)
#define TL_BOX_CREATE()       tlBoxCreate(0,0,0,0,0,0);


/*
 * types management
 */

/* create */
tlPoint *tlPointCreate (int ix, int iy, int id);
tlObject *tlObjectCreate (int x, int y, int width, int height);
tlBox *tlBoxCreate( int ix, int iy, int id, int iwidth, int iheight, int idisp);

/* init */
int tlObjectInit( tlObject *object,
                  int x, int y, int width, int height,
                  int nr_min, int nr_max, int ng_min, int ng_max);
int tlObjectInitSvs( tlObject *object );

/* destroy */
void tlPointDestroy (tlPoint *point);
void tlObjectDestroy (tlObject *object);
void tlBoxDestroy (tlBox *box );


/*
 * tracker management
 */

/* init tracker vars */
int tlTrackerInit( void );

/* shutdown tracker nicely */
int tlTrackerShutdown( void );
```

```
/* normalized skin color values */
#define TL_SKIN_NR_MIN         85
#define TL_SKIN_NR_MAX        110
#define TL_SKIN_NG_MIN         70
#define TL_SKIN_NG_MAX         83


/*****************************************************************************
 *
 * tlProcessImage --
 *   process images:
 *   filters left image with object normalized color values
 *   and create disparity map in the given tracking window
 *
 * INPUT:
 *   trackerObject     struct containing tracked object data
 *
 * OUTPUT:
 *   nrg_pic           binary pic from normalized color filter
 *
 * RETURN:
 *   0 on success.
 *
 * REMARK:
 *   Code is provided to run stereo processing in a different thread
 *   while performing color filtering. For some reason, multithread
 *   works fine when only one object is to be tracker, but stereo returns
 *   a blank image when more objects are called.
 *   Currently, stereo processing is performed sequentially before color
 *   filtering.
 *
 *****************************************************************************/
int
tlProcessImages( tlObject *trackerObject,
                 vlImage *nrg_pic );
```

```
/*****************************************************************************
 *
 * tlProcessImageMorph --
 *   process images:
 *   filters left image with object normalized color values,
 *   perform morphological operations on the filtered image
 *   and create disparity map in the given tracking window
 *
 * INPUT:
 *   trackerObject     struct containing tracked object data
 *   er_coeff          erosion coefficient for morph filter
 *   dil_coeff         dilatation coeff for morph filter
 *
 * OUTPUT:
 *   nrg_pic           binary pic from normalized color filter
 *   bin_pic           morphologically filtered image
 *
 * RETURN:
 *   0 on success.
 *
 * REMARK:
 *   Code is provided to run stereo processing in a different thread
 *   while performing color filtering. For some reason, multithread
 *   works fine when only one object is to be tracker, but stereo returns
 *   a blank image when more objects are called.
 *   Currently, stereo processing is performed sequentially before color
 *   filtering.
 *
 *****************************************************************************/
int
tlProcessImagesMorph( tlObject *trackerObject,
                      vlImage *nrg_pic,
                      vlImage *bin_pic,
                      int er_coeff,
                      int dil_coeff );


/*****************************************************************************
 *
 * tlBuildHistogram --
 *   given a binary image and a disparity map, builds an histogram of binary
 *   pixel versus disparity (i.e. for each disparity, count the number of
 *   active pixels in the binary image)
 *
 * INPUT:
 *   bin_pic           the binary image
 *   disp_pic          the disparity map
 *   window            subwindow to process
 *
 * OUTPUT:
 *   histogram         the result array
 *
 * RETURN:
 *   the disparity with the maximum active pixels
 *   (corresponds to the distance at which the filtered element present the
 *   biggest apparent area)
 *
 *****************************************************************************/
int
tlBuildHistogram( vlImage *bin_pic,
                  vlImage *disp_pic,
                  vlWindow *window,
                  int *histogram );
```

```
/*****************************************************************************
 *
 * tlFindBlob --
 *   filter a given disparity image through a given mask
 *
 * INPUT:
 *   disp_pic                disparity image
 *   bin_pic                 mask image
 *   blob_width/height       blob dimensions
 *   disp +/- tolerance      disparity gap
 *   threshold               the minimum percentage of 1s over 0s for a hit
 *   window                  subwindow to process
 *
 * OUTPUT:
 *   location                the blob location (if any)
 *
 * RETURN:
 *   0 on success, -1 on failure (no blob found)
 *
 * REMARK:
 *   to ensure fast execution, this routine returns as soon as a match
 *   is found; therefore, the location returned might not be the optimum
 *   correlation match; also, there may be other blobs (that could produce)
 *   a better match.
 *
 *   To make this function optimum in terms of correlation match, one would
 *   need to make it perform correlation through the whole winwow and extract
 *   the best match out of all possible results - significantly slower.
 *
 *   To ensure accurate blob localization, use high threshold value.
 *
 *****************************************************************************/
int
tlFindBlob( vlImage *disp_pic,
            vlImage *bin_pic,
            int blob_width,
            int blob_height,
            int disp,
            int tolerance,
            int threshold,
            vlWindow *window,
            tlPoint *location );


/*****************************************************************************
 *
 * tlExtractPlane --
 *   filter a given disparity image through a given mask
 *
 * INPUT:
 *   disp_pic          disparity image
 *   bin_pic           mask image
 *   min,max_disp      plane disparity boundaries
 *   window            subwindow to process
 *
 * OUTPUT:
 *   plane_pic the filtered image
 *
 * RETURN:
 *   0 on success, -1 on failure.
 *
 *****************************************************************************/
int
tlExtractPlane( vlImage *disp_pic,
                vlImage *bin_pic,
                int min_disp,
                int max_disp,
                vlWindow *window,
                vlImage *plane_pic );
```

tlStereo.h

```
/* stereo camera constants */
#define TL_SVS_BASELINE        7     /* 7 cm */
#define TL_SVS_DISPARITY      16     /* number of disparities to compute */
#define TL_SVS_FOCAL          25e-1  /* 25mm */

/* constants */
#define TL_SVS_RANGE          64     /* 4*TL_SVS_DISPARITY (pixel interpolation) */

/* scaling macros
   (empirical, yet accurate !) */
#define TL_SCALE_I2W(dim,disp) (int)( (dim-6)*30.00 / disp )       /* in cm */
#define TL_SCALE_W2I(dim,disp) (int)( 6 + disp*dim / 30.00 )       /* in pixels */

/* global stereo parameters */
extern svsSP *svsParamGlob;


/*****************************************************************************
 *
 * tlSvsInit --
 *   initialize a given svs parameter struct
 *
 * INPUT:
 *   svsP      parameter struct for the stereo engine
 *
 * RETURN:
 *   0 on success.
 *
 *****************************************************************************/
int
tlSvsInit( svsSP *svsParam );


/*****************************************************************************
 *
 * tlVislib2Svs --
 *   image format conversion between visLib RGB and SVS
 *
 * INPUT:
 *   pic       a vislib RGB image
 *   window    a vislib window that defines a subwindow of the image
 *
 * OUTPUT:
 *   dest      a subwindow of unsigned char that's usable by the svs module
 *
 * RETURN
 *   0 on success, (-1) on error.
 *
 * REMARK:
 *   this function requires the dest pointer to be allocated with the
 *   correct size prior to call; it checks buffer size and returns
 *   an error if it doesn't match the window size.
 *
 *****************************************************************************/
int
tlVislib2Svs( vlImage *src,
              vlWindow *window,
              unsigned char *dest );
```

```
/*****************************************************************************
 *
 * tlSvs2Vislib --
 *   image format conversion between SVS and VisLib GRAY
 *
 * INPUT:
 *   src               an array of unsigned char representing image disparity
 *   window            a vislib window defining the subwindow of the original image
 *   updateWindow      a vislib window defining the actual subwindow to update
 *                     (because SVS can't process the borders)
 *
 * OUTPUT:
 *   dest              a visLib image
 *
 * RETURN:
 *   0 on success, (-1) on error.
 *
 * REMARK:
 *   this function does NOT reallocate the image, as it only updates a portion
 *   of the image at a time and can be used in multiple threads to simultaneously
 *   update several subwindows.
 *
 *****************************************************************************/
int
tlSvs2Vislib( unsigned char *src,
              vlWindow *window,
              vlWindow *updateWindow,
              vlImage *dest );


/*****************************************************************************
 *
 * tlSvsCalibrate --
 *   calibrate the stereo input (found optimal external parameters)
 *
 * INPUT:
 *   tracker           tracking data for a given object
 *   leftImage         images from the stereo pair (globals)
 *   rightImage
 *
 * RETURN:
 *   0 on success, -1 on failure.
 *
 *****************************************************************************/
int
tlSvsCalibrate( tlObject *trackerObject );


/*****************************************************************************
 *
 * tlSvs2Display --
 *   produce a pix map of disp scaled for display
 *
 * INPUT:
 *   src       the visLib image containing the raw disp
 *
 * OUTPUT:
 *   dest      a visLib image
 *
 * RETURN:
 *   0 on success, (-1) on error.
 *
 *****************************************************************************/
int
tlSvs2Display( vlImage *src,
               vlImage *dest );
```

```
/*****************************************************************************
 *
 * tlSvsProcess --
 *   extract disparity from stereo input
 *
 * INPUT:
 *   tracker           tracking data for a given object
 *   leftImage         images from the stereo pair (globals)
 *   rightImage
 *
 * OUTPUT:
 *   stereoImage       update a subwindow in the disparity image (global)
 *   window            the window member of the tracker object is updated
 *                     to the actual size of the computed stereo area
 *
 * RETURN:
 *   0 on success, (-1) on failure.
 *
 * REMARK:
 *   under Windows, designed to work as an independent thread using
 *   the 'stereoMutex' of defined for each trackerObject.
 *
 *****************************************************************************/
int
tlSvsProcess( tlObject *trackerObject );
```

tlTimer.h

```
/*****************************************************************************
 *
 * REMARK:
 *   the timer uses Windows timeGetTime() routine
 *   this routine counts the time since Windows was last started;
 *   it undergoes a period of 2^32 milliseconds (which is about 49.71 days)
 *
 *****************************************************************************/


/* timer count */
#define TL_MAX_TIMERS                5

/* timer assigment */
#define TL_TIMER_SYSTEM              0
#define TL_TIMER_HEAD                1
#define TL_TIMER_RIGHT_HAND          2
#define TL_TIMER_LEFT_HAND           3
#define TL_TIMER_GRAPH_DISPLAY       4

/* timer array */
extern int tlTimer[TL_MAX_TIMERS];


/*****************************************************************************
 *
 * tlTimerGet --
 *   get the time in ms since the specific timer was last reset
 *
 * INPUT:
 *   timerID   the ID tag of the timer to be polled
 *
 * NOTE:
 *   every timer needs to be reset once before being used;
 *   unpredictable results may occur otherwise.
 *
 *****************************************************************************/
int
tlTimerGet( int timerID );


/*****************************************************************************
 *
 * tlTimerReset --
 *   set the reference time of the given timer to zero
 *
 * INPUT:
 *   timerID   the ID tag of the timer reset
 *
 *****************************************************************************/
void
tlTimerReset( int timerID );
```

```
/* minimum object size [cm] */
#define TL_LOCATE_WIDTH              10
#define TL_LOCATE_HEIGHT             10

/* regular morphological filter coefficients */
#define TL_LOCATE_ER_COEFF            2
#define TL_LOCATE_DIL_COEFF           4

/* object detection boundaries and threshold */
#define TL_LOCATE_STD_DISP            4            /* typical object disparity range */
#define TL_LOCATE_STD_THRES          80            /* minimum acceptable confidence in
                                                      object localization [%] */

/*****************************************************************************
 *
 * tlLocate --
 *   main object localization function;
 *   this function calls specific localization routines for different
 *   object types - or the default routine for standard objects.
 *
 * INPUTS:
 *   type             object type
 *   box              volume of space in which the object is to be localized
 *   trackerObject    object data
 *   objectParam      pointer to parameter specific to each object type
 *
 * OUTPUTS:
 *   location         object coords are placed in the location var
 *
 * RETURN:
 *   object status (either LOST or OK)
 *
 * REMARK:
 *   box coord used here are in the image frame.
 *
 *****************************************************************************/
int
tlLocate( tlPoint *location,
          tlObjectType type,
          tlBox *box,
          tlObject *trackerObject,
          void *objectParam );


/*****************************************************************************
 *
 * tlLocateStdObject --
 *   deals with localization of object of type STD
 *
 * INPUTS:
 *   box              volume of space in which the object is to be localized
 *   trackerObject    object data
 *
 * OUTPUTS:
 *   location         the object coords are put in the location var
 *
 * RETURN:
 *   the status of the object (either LOST or OK)
 *
 * REMARK:
 *   box coord used here are in the image frame.
 *
 *****************************************************************************/
int
tlLocateStdObject( tlPoint *location,
                   tlBox *box,
                   tlObject *trackerObject );
```

tlTrack.h

```
/* morph filters constants */
#define TL_TRACK_ER_COEFF             2
#define TL_TRACK_DIL_COEFF            4

/* tracking constants */
#define TL_TRACK_HIST_THRESHOLD       0.66   /* histogram normalized peak threshold */
#define TL_TRACK_DISP_RANGE           2      /* object disparity range */
#define TL_TRACK_MIN_THRESHOLD       30      /* minimum acceptable confidence in
                                                object localization in percent */


/*****************************************************************************
 *
 * tlTrack --
 *   main object tracking function;
 *   this function calls specific tracking routines for different
 *   object types - or the default routine for standard objects.
 *
 * INPUTS:
 *   type             object type
 *   trackerObject    parameters for the tracking engine
 *   objectParam      pointer to object type specific parameter
 *
 * OUTPUTS:
 *   location                 coords of the object in both sensor and world frames
 *
 * RETURN:
 *   measure of object shape alteration [%] if found, TL_LOST otherwise.
 *
 *****************************************************************************/
int
tlTrack( tlPoint *location,
         tlObjectType type,
         tlObject *trackerObject,
         void *objectParam );


/*****************************************************************************
 *
 * tlTrackStdObject --
 *   default tracking routine for objects of type STD.
 *
 * INPUTS:
 *   trackerObject    parameters for the tracking engine
 *
 * OUTPUTS:
 *   location         coords of the object
 *
 * RETURN:
 *   measure of object shape alteration [%] if found, TL_LOST otherwise.
 *
 *****************************************************************************/
int
tlTrackStdObject( tlPoint *location,
                  tlObject *trackerObject );
```

```
/* object differentiation constants */
#define TL_MIN_DISP_DIFF       2             /* min disp difference between objects */
#define TL_MIN_DIST           15             /* min dist [cm] between objects */


/*****************************************************************************
 *
 * tlImage2World --
 *   compute a location coordinate in the world frame
 *   given a position in the image
 *
 * INPUTS:
 *   src       source point
 *   dest      point to receive converted coord
 *
 * REMARK:
 *   src and dest can be the same point !
 *
 *****************************************************************************/
void
tlImage2World( tlPoint *src,
               tlPoint *dest );


/*****************************************************************************
 *
 * tlWorld2Image --
 *   compute a location coordinate in the image frame
 *   given a position in the world
 *
 * INPUTS:
 *   src       source point
 *   dest      point to receive converted coord
 *
 * REMARK:
 *   src and dest can be the same point !
 *
 * NOTE:
 *   doesn't work yet !!!
 *   complementary function to svsReconstruct3D doesn't exist yet and
 *   is not straightforward.
 *   Ignore this function for now.
 *
 *****************************************************************************/
void
tlWorld2Image( tlPoint *src,
               tlPoint *dest );


/*****************************************************************************
 *
 * tlBox2World --
 *   update a box world coordinates from its image coordinates
 *
 * INPUTS:
 *   box       box to be updated
 *
 *****************************************************************************/
void
tlBox2World( tlBox *box );
```

```
/*****************************************************************************
 *
 * tlBox2Image --
 *   update a box image coordinates from its world coordinates
 *
 * INPUTS:
 *   box       box to be updated
 *
 * NOTE:
 *   doesn't work yet !!!
 *   complementary function to svsReconstruct3D doesn't exist yet and
 *   is not straightforward.
 *   Ignore this function for now.
 *
 *****************************************************************************/
void
tlBox2Image( tlBox *box );


/*****************************************************************************
 *
 * tlSameObject --
 *   given two objects location, check whether they correspond to
 *   the same physical entity (that is, in the real world) or not.
 *
 * INPUTS:
 *   location1,2      objects location
 *
 * RETURN:
 *   1 if objects are the same, 0 otherwise.
 *
 *****************************************************************************/
int
tlSameObject( tlPoint *location1,
              tlPoint *location2 );


/*****************************************************************************
 *
 * tlDistanceBetween --
 *   compute scalar distance between two points
 *
 * INPUT:
 *   location1,2      extremities of the segment to measure
 *
 * RETURN:
 *   distance [pix]
 *
 * REMARK:
 *   only computes the planar distance in the image
 *   (doesn't take disparity into account)
 *
 *****************************************************************************/
int
tlDistanceBetween( tlPoint *location1,
                   tlPoint *location2 );
```

```
/*****************************************************************************
 *
 * tlRemoveObject --
 *   remove the shape of an object in a binary picture (set all the pixels to 0)
 *
 * INPUT:
 *   bin_pic   binary map
 *   object    the object to remove
 *
 * RETURN:
 *   0 on success, -1 on failure.
 *
 *****************************************************************************/
int
tlRemoveObject( vlImage *bin_pic,
                vlObject *object );


/*****************************************************************************
 *
 * tlCopyPoint --
 *   copies a coord into another point struct
 *
 * INPUT:
 *   src       source point
 *
 * OUTPUT:
 *   dest      the copied point
 *
 *****************************************************************************/
void
tlCopyPoint( tlPoint *src,
             tlPoint *dest );


/*****************************************************************************
 *
 * tlDrawLine --
 *   draw a line onto a RGB image
 *
 * INPUTS:
 *   p1        line origine
 *   p2        line termination
 *   pic       image to draw on
 *   r,g,b     RGB color [0-255]
 *
 * RETURN:
 *   0 on success, (-1) on failure.
 *
 *****************************************************************************/
int
tlDrawLine( tlPoint *p1,
            tlPoint *p2,
            vlImage *pic,
            unsigned char r,
            unsigned char g,
            unsigned char b );
```

tlModule_hmLib.h

```
/* misc constants */
#define TL_HL_HEAD                    0
#define TL_HL_RIGHT_HAND              1
#define TL_HL_LEFT_HAND               2

/* human model dimensions [cm] */
#define TL_HEAD_WIDTH                15
#define TL_HEAD_HEIGHT               20
#define TL_TORSO_WIDTH               30
#define TL_TORSO_HEIGHT              60
#define TL_ARM_LENGTH               180
#define TL_HAND_WIDTH                 3
#define TL_HAND_HEIGHT                3

/* tracking window extra size
   (around object size) */
#define TL_HEAD_TRACKWIN_X           15
#define TL_HEAD_TRACKWIN_Y           10
#define TL_HAND_TRACKWIN_X           20
#define TL_HAND_TRACKWIN_Y           20

/* morph filters coefficients */
#define TL_HEAD_ER_COEFF              2
#define TL_HEAD_DIL_COEFF             6
#define TL_HAND_ER_COEFF              2
#define TL_HAND_DIL_COEFF             4

/* extraction constants */
#define TL_HL_LOCATE_HEAD_HEIGHT    150    /* max Y pos where we expect a head [pix] */
#define TL_HL_LOCATE_HEAD_DISP        4    /* disparity tolerance in head extraction */
#define TL_HL_LOCATE_HEAD_THRES      90    /* area percent threshold for head
extraction */
#define TL_HL_LOCATE_TORSO_DISP       4    /* disparity range for torso extraction */
#define TL_HL_LOCATE_TORSO_THRES     70    /* confidence for torso extraction [%] */
#define TL_HL_LOCATE_HAND_DISP        4    /* disparity range for hand extraction */
#define TL_HL_LOCATE_HAND_THRES      95    /* confidence for hand extraction [%] */

/* tracking constants */
#define TL_HL_TRACK_HEAD_DISP         2
#define TL_HL_TRACK_HEAD_THRES       50
#define TL_HL_TRACK_HAND_DISP         2
#define TL_HL_TRACK_HAND_THRES       30

/* agitation threshold */
#define TL_HL_AGITATION_THRES        12

/* timer threshold */
#define TL_LOCATE_HEAD_DELAY       1000
#define TL_LOCATE_HAND_DELAY       2000

/* display constant */
#define TL_HL_GRAPH_UPDATE         2500   /* update delay in ms */
#define TL_HL_GRAPH_HEIGHT           25   /* [pixel] */
#define TL_HL_GRAPH_BORDER           10
#define TL_HL_GRAPH_SPACE            20
#define TL_HL_GRAPH_START_LINE      280
```

```
/*****************************************************************************
 *
 * tlLocate_HL_Head --
 *   hmlib specific routine
 *   search for a human and locate the head
 *
 * INPUT:
 *   trackerObject     hint on initial head position
 *   objectParam       pointer to a hlHuman class
 *
 * OUTPUT:
 *   trackerObject     object is updated with head data
 *   location          head location
 *
 * RETURN:
 *   the status of the detection (TL_OK or TL_LOST)
 *
 *****************************************************************************/
int tlLocate_HL_Head( tlPoint *location,
                      tlObject *trackerObject,
                      void *objectParam );


/*****************************************************************************
 *
 * tlLocate_HL_RightHand,
 * tlLocate_HL_LeftHand,
 * tlLocate_HL_BothHands --
 *   hmlib specific routine
 *   knowing a human head location, find one or both hands.
 *
 * INPUT:
 *   trackerObject     hint on initial hand position
 *   objectParam       pointer to a hlHuman class
 *
 * OUTPUT:
 *   trackerObject     object is updated with hand data
 *   location          hand location
 *
 * RETURN:
 *   the status of the detection (TL_OK or TL_LOST)
 *
 * REMARK:
 *   tlLocate_HL_BothHands bypasses function return control and updates
 *   the model by itself (using the objectParam pointer); therefore,
 *   it doesn't take any explicit location and trackerObject input.
 *
 *****************************************************************************/
int
tlLocate_HL_RightHand( tlPoint *location,
                       tlObject *trackerObject,
                       void *objectParam );

int
tlLocate_HL_LeftHand ( tlPoint *location,
                       tlObject *trackerObject,
                       void *objectParam );

int
tlLocate_HL_BothHands( void *objectParam );
```

```
/*****************************************************************************
 *
 * tlTrack_HL_Head --
 *   tracking routine for the head of a human model
 *
 * INPUTS:
 *   trackerObject    parameters for the tracking engine
 *
 * OUTPUTS:
 *   location         coords of the object
 *
 * RETURN:
 *   measure of head shape change [%] if found, TL_LOST otherwise
 *
 *****************************************************************************/
int
tlTrack_HL_Head( tlPoint *location,
                 tlObject *trackerObject,
                 void *objectParam );


/*****************************************************************************
 *
 * tlTrack_HL_RightHand,
 * tlTrack_HL_LeftHand --
 *   tracking routine for one of the hands of a human model
 *
 * INPUTS:
 *   trackerObject    parameters for the tracking engine
 *
 * OUTPUTS:
 *   location         coord of the hand after detection
 *
 * RETURN:
 *   measure of hand shape change [%] if found, TL_LOST otherwise.
 *
 *****************************************************************************/
int
tlTrack_HL_RightHand( tlPoint *location,
                      tlObject *trackerObject,
                      void *objectParam );

int
tlTrack_HL_LeftHand ( tlPoint *location,
                      tlObject *trackerObject,
                      void *objectParam );
```

```
/*****************************************************************************
 *
 * tl_HL_DisplayHuman --
 *   hmlib specific routine
 *   display the human model on a given image
 *
 * INPUT:
 *   human     pointer to a hlHuman class
 *
 * OUTPUT:
 *   pic       image to display on
 *
 * RETURN:
 *   0 on success.
 *
 *****************************************************************************/
int
tl_HL_DisplayHuman( void *humanPtr,
                    vlImage *pic );


/*****************************************************************************
 *
 * tl_HL_InvertHands --
 *   invert the left and right hands of a HL human model
 *
 * INPUT:
 *   hb        the human model object
 *
 * RETURN:
 *   0 on success.
 *
 *****************************************************************************/
int
tl_HL_InvertHands( void *hb );
```

tlDisplay.h

```
/* constants */
#define TL_TEXT_AREA_HEIGHT          43             /* status line */
#define TL_EXTRA_WINDOW_BORDER        0             /* gap around window */


/*****************************************************************************
 *
 * tlDisplayInit --
 *  create a simple display window in Windows
 *
 * INPUTS:
 *   hInst     Windows application instance
 *   hPrev     Initialization instance
 *   sw        Windows window style
 *   width     # of columns
 *   height    # of height
 *
 * RETURNS
 *   On success, the depth (bits) of the visual is returned. On failure,
 *   -1 is returned.
 *
 *****************************************************************************/
int
tlDisplayInit( HINSTANCE hInst,
               HINSTANCE hPrev,
               int sw,
               int width,
               int height );


/*****************************************************************************
 *
 * tlDisplayShutdown --
 *  shutdown the display
 *
 * RETURNS:
 *   0 on success.
 *
 *****************************************************************************/
int
tlDisplayShutdown (void);


/*****************************************************************************
 *
 * tlShow --
 *  display an image of any type. This function converts a tlImage
 *      into a RGB image and then calls _tlDisplayImage()
 *
 * INPUTS:
 *   pic       rgb image
 *   wait      if non-zero, the function will pause until a key is pressed
 *
 * RETURNS:
 *   On success, 0 is returned. Otherwise, -1.
 *
 *****************************************************************************/
int
tlShow( vlImage *image,
        int wait );
```

```
/*****************************************************************************
 *
 * tlShowMsg --
 *      Display a simple dialog box
 *
 * INPUTS:
 *      msg     a LPCSTR string to be displayed in the dialog box.
 *
 *****************************************************************************/
void
tlShowMsg( LPCSTR msg );


/*****************************************************************************
 *
 * tlEvent --
 *   non-blocking function; check for events in the window queue
 *
 * RETURNS:
 *   If there are events, returns TRUE. If no events, returns FALSE.
 *   Returns TL_EXIT if WM_QUIT is received (typically from a WM_DESTROY call)
 *
 *****************************************************************************/
int
tlEvent (void);


/*****************************************************************************
 *
 * tlExit --
 *   perform a clean exit of the library
 *   (includes framegrabber handles closing)
 *
 * REMARK:
 *   this routine simply posts a WM_DESTROY message
 *   the event manager then deals with the clean up
 *   the purpose of this function is to offer an alternative to the
 *   system exit() call - this routine prevents hardware freeze that
 *   then requires reboot
 *
 *****************************************************************************/
void
tlExit( void );


/*****************************************************************************
 *
 * tlDisplayText --
 *   display text on status line below the displayed image
 *
 *****************************************************************************/
void
tlDisplayText( char *text );


/*****************************************************************************
 *
 * tlDrawRect --
 *   draw a rectangle anywhere in the program window
 *   with current color settings
 *
 * INPUT:
 *   x,y               upper left corner coord
 *   width, height     rectangle dimensions
 *
 *****************************************************************************/
void
tlDrawRect( int x,
            int y,
            int width,
            int height );
```

```
/*****************************************************************************
 *
 * tlSetBkColor --
 *   set current drawing color to background color
 *
 *****************************************************************************/
void
tlSetBkColor( void );


/*****************************************************************************
 *
 * tlSetColor --
 *   set drawing color to any given RGB value
 *
 * INPUT:
 *   r,g,b     the desired RGB color
 *
 *****************************************************************************/
void
tlSetColor( unsigned char r,
            unsigned char g,
            unsigned char b );
```

```
/*****************************************************************************
 *
 * UNTESTED FUNCTIONS --
 *   never used (provided for convenience)
 *
 *****************************************************************************/


/*****************************************************************************
 *
 * tlButtonPress --
 *  Check for a mouse button press.
 *
 * RETURNS:
 *   If a button has been pressed, returns TRUE. If no press, returns FALSE.
 *
 *****************************************************************************/
int
tlButtonPress( void );


/*****************************************************************************
 *
 * tlGetPoint --
 *  Returns mouse click coordinates (if there was a click only).
 *  This is NOT a BLOCKING function.
 *
 * OUTPUTS:
 *   point     point to be filled with coordinates
 *
 * RETURNS:
 *   The button which was pressed:
 *      1  left button
 *      2  middle button
 *      3  right button
 *
 *   Or -1 if no click was detected or on error.
 *
 *****************************************************************************/
int
tlGetPoint( vlPoint *point );


/*****************************************************************************
 *
 * tlShowUntilClick --
 *   display an image and wait for a mouse click
 *
 * INPUTS:
 *   pic       rgb image
 *
 * RETURNS:
 *   On success, 0 is returned. Otherwise, -1.
 *
 * WARNING
 *   Because of the way Windows event manager work, it is not possible to
 *   catch click events without disrupting regular event processing.
 *   In particular, WM_QUIT and WM_DESTORY messages will not reach the
 *   main thread if processed in this function.
 *   Therefore, this function will return TL_EXIT in such cases;
 *   whoever called the function will have to deal with this value to ensure
 *   clean exit of the program.
 *
 *****************************************************************************/
int
tlShowUntilClick( vlImage *pic );
```

tlGrab.h

```
#define TL_GRAB_LEFT_CHANNEL  0      /* channel for the left cam framegrabber */
#define TL_GRAB_RIGHT_CHANNEL 1      /* channel for the right cam framegrabber */

/* dynamically linked library (framegrabber driver) */
#define TL_PXC_DLL     "pxc_95.dll"
#define TL_FRAME_DLL   "frame_32.dll"

/* this program can use either 24 bit color or 8 bit gray frames.
   change the value of PIXEL_TYPE to PBITS_RGB24 or PBITS_Y8 */
#define TL_PIXEL_TYPE PBITS_RGB24

/* library name */
#define TL_WIN_LIB_NAME "H.O.T. - Human Oriented Tracking"


/*****************************************************************************
 *
 * tlDualGrabInit --
 *   initialize the 2 framegrabbers
 *
 * INPUTS:
 *   cols      # of columns to grab (-1 for VL_COLS_DEFAULT)
 *   rows      # of rows to grab    (-1 for VL_ROWS_DEFAULT)
 *   pal       input is PAL, FALSE if input is NTSC,
 *             -1 for automatic detection
 *
 * RETURNS:
 *   On success, 0 is returned. Otherwise, -1.
 *
 *****************************************************************************/
int
tlDualGrabInit ( int cols,
                 int rows,
                 int pal );


/*****************************************************************************
 *
 * tlDualShutdown --
 *   perform a clean shutdown of the hardware
 *
 * RETURNS:
 *   0 on success.
 *
 *****************************************************************************/
int
tlDualGrabShutdown ( void );


/*****************************************************************************
 *
 * tlGrab --
 *   grab two images from the framegrabber
 *   and update the leftImage and rightImage globals
 *
 * RETURNS:
 *   0 on success.
 *
 *****************************************************************************/
int
tlGrab( void );
```