6
Prototype application

As a constituent part of this work, a prototype application which demonstrates the chosen approach to the manipulation of virtual 3D objects, has been developed.

6.1
Requirements

The basic requirements for the prototype application were as follows:

- the application must run at interactive rates.
- the application must implement the following basic set of manipulation operations: SELECT, DESELECT, TRANSLATE, ROTATE, SCALE.
- the hardware part must be based on commercial, off-the-shelf components (standard PC, inexpensive low-resolution web cameras).
- the system shall use passive stereo vision in order to reconstruct 3D position of the hand.
- the system shall use the 3 d.o.f. hand model (for each hand, therefore effectively creating a 6 d.o.f. input device), as described in Section 2.3.1 on page 23.

6.2
Constraints, assumptions and restrictions

The constraints on the system were as follows: the workplace (Figure 6.1) consists of a standard office cubicle equipped with a personal computer with two cameras (i.e. a stereo pair) connected. Cameras are fixed at the top of the cubicle and are directed down, at a certain angle, relative to the surface of the desktop. A stereo pair of cameras enables us, due to the phenomenon called *stereo disparity*, to estimate 3D positions of various hand features, thus offering us a way to integrate our hands into the VE.

Accordingly, we define the *workspace* as the intersection of the two visual cones defined by the respective cameras' field of view — the user must move his

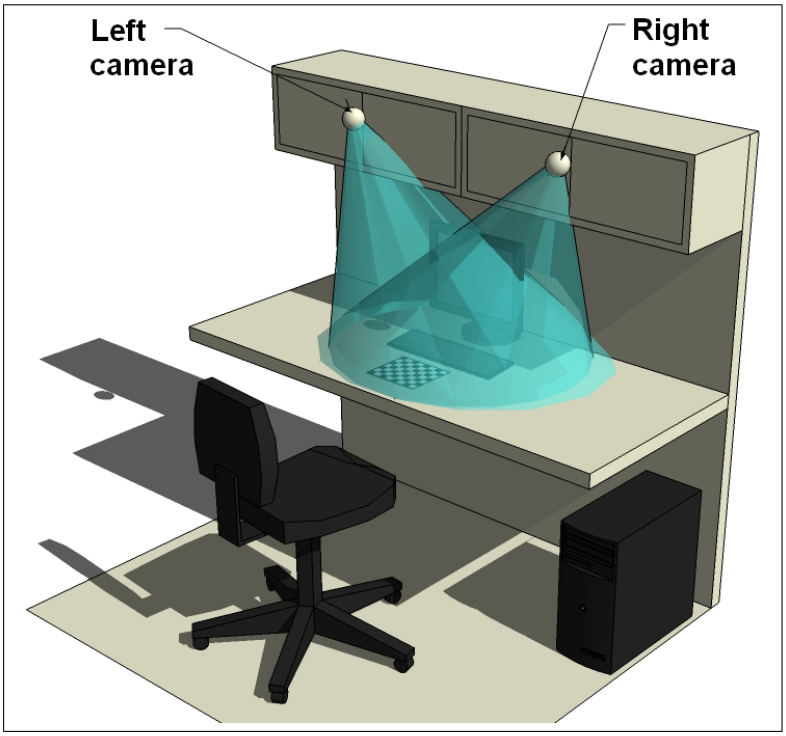

Figure 6.1: The user's workplace

hands in this working space, in order for the system to register hand movements and gestures. If a hand exits the workspace, the system stops tracking the hand.

Also:

- Cameras are fixed, oriented downwards and towards the desktop surface. and the background (desktop surface) does not change in time.
- Palms must always be approximately co-planar with the desk surface, in order for the system to successfully detect hand postures.
- Illumination cannot be too weak, because in this case the pixel segmentation process, based on human skin color, would fail.
- The hands must move approximately in the far-distance field of the cameras, because if they move too near the cameras, the perspective distortion gets large and 3D triangulation fails.

6.3 Hand postures defined

Figure 6.2 depicts the three hand postures we use in our application. Note that the image depicts the right hand only, but both the left and the right hand can assume these postures. (The left hand assumes postures which are simply mirrored relative to the vertical axis.) Also please note that the hand

postures shown in the image are inclined at an angle, which approximates the natural hand inclination when it is being tracked in the workspace. We now define the following hand postures (also called *hand states*), from which all the manipulation operations are being defined, as follows:

1. `HAND_POSTURE_OPEN` — flat palm with all fingers spread apart
2. `HAND_POSTURE_POINTING` — all fingers closed, except the index finger
3. `HAND_POSTURE_FIST` — all fingers closed

Figure 6.2: Three hand postures utilized by the system: `HAND_POSTURE_OPEN` (left), `HAND_POSTURE_POINTING` (middle) and `HAND_POSTURE_FIST` (right)

As a matter of convenience, we defined one more posture, `HAND_POSTURE_UNKNOWN`, which designates any hand posture that is not recognized by the system.

The criteria used when choosing postures and the mapping between postures and manipulation operations were:

1. the "naturalness" of the posture, that is, the similarity between similar hand movements and gestures when manipulating real, physical objects, and
2. sufficient degree of inter-posture "otherness", in other words the appearances of postures had to be sufficiently different in order to achieve better visual separability of postures, thus allowing better detection performance.

6.4 Manipulation operations implemented

This section gives the list of implemented hand gestures for the manipulation of 3D virtual objects.

Using the hand postures defined in Section 6.3, we now define *direct 3D manipulation operations*. Manipulation operations can be either one-handed or two-handed:

1. `OP_SELECT` (one-handed) — selects an object. Based on the posture `HAND_POSTURE_POINTING`.

2. `OP_DESELECT` (one-handed) — deselects an object. Based on the posture `HAND_POSTURE_POINTING`.

3. `OP_TRANSLATE` (one-handed) — translates (moves) selected objects. Based on the posture `HAND_POSTURE_FIST`.

4. `OP_ROTATE` (two-handed) — Rotates selected object. Based on two `HAND_POSTURE_POINTING` hand postures.

5. `OP_SCALE` (two-handed) — scales objects. Based on two `HAND_POSTURE_FIST` hand postures.

Therefore, we have two two-handed spatial operations: `OP_ROTATE` and `OP_SCALE` — these use both hands, the left and the right one, at the same time. The remaining three spatial operations (`OP_SELECT`, `OP_DESELECT` and `OP_TRANSLATE`) are one-handed — must be performed by just one hand, either just by the left or just by the right hand. Each of these operations will now be described in detail.

6.4.1 Selecting and deselecting objects

Operation `OP_SELECT` selects an object in the scene, while operation `OP_DESELECT` deselects an (already selected) 3D object. In order to (de)select a 3D object, the user extends the index finger of *one and exactly one* hand (thus changing that hand's state into `HAND_POSTURE_POINTING`), and moves the hand into the object. (We emphasized "one and exactly one" because *two* tracked hands which are in the `HAND_POSTURE_POINTING` state perform the operation `OP_ROTATE`, see Section 6.4.3.) As soon as the application detects that the tracked hand's centroid entered the interior of the object, while the hand is in state `HAND_POSTURE_POINTING`, the objects gets (de)selected. The user can now move her hand out of the object; the object stays (de)selected.

In WIMP terms, operation OP_SELECT is equivalent to moving the mouse pointer onto a screen object (for example, onto an icon) and then pressing the mouse button, thus selecting the icon. Similarly, operation OP_DESELECT is equivalent to moving the mouse pointer onto an already selected screen object (for example, onto an already selected icon) and then pressing the mouse button, which deselects the icon.

6.4.2 Translating objects

Operation OP_TRANSLATE translates (moves) all the currently selected objects (Figure 6.3). For this to work, *one and exactly one* hand must be in the HAND_POSTURE_FIST state. (We say "exactly one" because if both tracked hands are in the HAND_POSTURE_FIST state, we will be performing the OP_SCALE operation, see Section 6.4.4.)

The operation initiates at the moment the user changes the state of one (and exactly one) of her hands into HAND_POSTURE_FIST. The position of that hand's centroid is then the starting point of the translation vector. User moves the hand about, and the selected objects move along too. When the user decides the translation is just right, she changes the hand's state into HAND_POSTURE_OPEN thus terminating the OP_TRANSLATE operation.

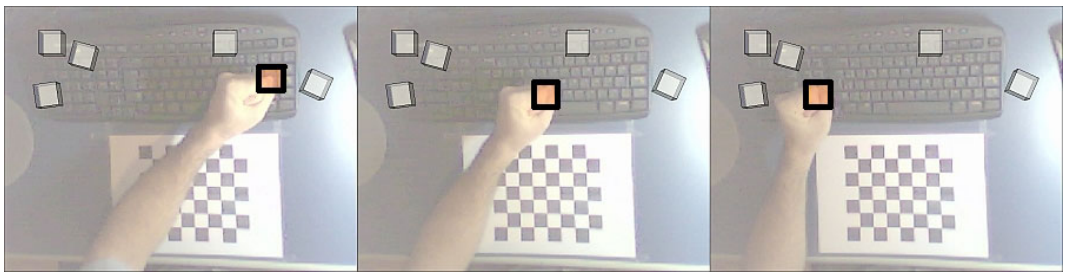

Figure 6.3: OP_TRANSLATE operation, based on one HAND_POSTURE_FIST posture

6.4.3 Rotating objects

Operation OP_ROTATE rotates all the currently selected objects (Figure 6.4). It is a *bimanual-asymmetric* operation (see Section 3.2 on page 28). At the moment when the application detects that *both* hands have their index finger extended (thus entering into the HAND_POSTURE_POINTING state), the hands' respective positions (A for the left, and B for the right hand) get memorized and defined as the vector $\vec{u} = B - A$. If subsequent positions of both the left and the right hand are C and D respectively, and if define $\vec{v} = D - C$, then

the current rotation axis is the vectorial cross-product $\vec{u} \times \vec{v}$, and the rotation angle is equal to the angle between vectors $\vec{u}$ and $\vec{v}$.

The user can now move her hands about, and the selected objects rotate around their (local) origins in real time. When the user decides to stop the rotation, she changes both hands' state into `HAND_POSTURE_OPEN` thus terminating the `OP_ROTATE` operation.

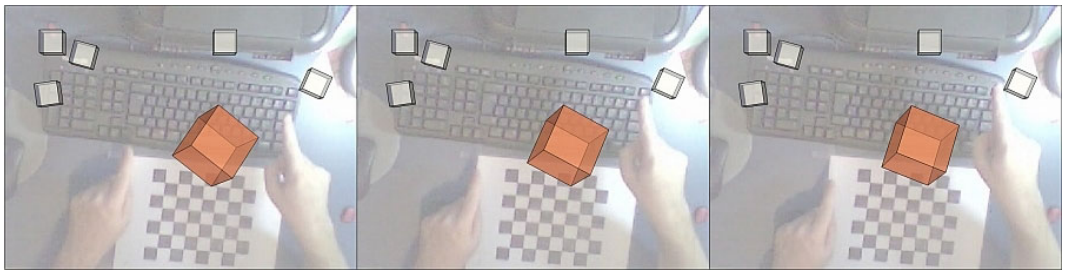

Figure 6.4: The two-handed `OP_ROTATE` operation is based on two `HAND_POSTURE_POINTING` postures. An example of a CCW rotation shown

6.4.4 Scaling objects

Operation `OP_SCALE` scales all the currently selected objects (Figure 6.5). It is a *bimanual-symmetric* operation (see Section 3.2 on page 28). At the moment both hands enter into the `HAND_POSTURE_FIST` state, the locations of both hands get memorized and defined as two points A, B. If subsequent positions of both the left and the right hand are C and D respectively, then the current scaling factor is defined simply as the ratio $\frac{|C-D|}{|A-B|}$, which is a ratio of lengths: first length defined by points A and B, and the second length defined by points C and D.

Therefore, during the scaling operation, the user can move her hands, and the selected objects scale in real-time around their (local) origins, in the directions defined by their local coordinate systems. When the user decides to stop the scaling, she changes one (or both) hand's state into `HAND_POSTURE_OPEN` thus terminating the `OP_SCALE` operation.

f

6.5 Control flow

Before anything, the stereo rig must be calibrated i.e. its parameters (intrinsic and extrinsic) determined. Only by knowing these camera parameters,

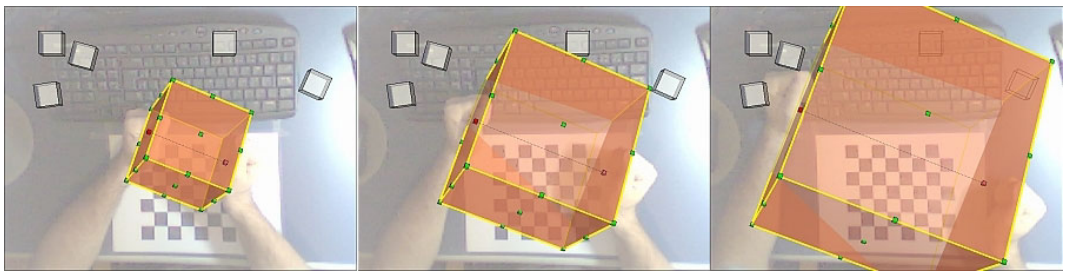

Figure 6.5: The two-handed `OP_SCALE` operation is based on two `HAND_POSTURE_FIST` postures

can CV techniques (specifically, the triangulation technique adopted) reconstruct 3D position of our hands in the workspace.

The set K of intrinsic parameters for a single camera includes two focal lengths (f_x, f_y), the principal point (o_x, o_y) and four distortion parameters (k_1, k_2, k_3, k_4):

$$K = \{(f_x, f_y), (o_x, o_y), (k_1, k_2, k_3, k_4)\}$$

We determined these intrinsic parameters using the Zhang's method [19]. For the calibration pattern we used a 8×7 checkerboard pattern.

Having determined two sets K_L, K_R of intrinsic parameters (for the left and right camera), we can proceed to determining extrinsic parameters (orientation and distance of a camera relative to the pattern). For this, we now fix the 8×7 checkerboard on the desk surface, and orient cameras so that they both have the pattern in their field of view.

Knowing both cameras' intrinsic parameters, we can now compute the extrinsic parameters (rotation matrices R_L and R_R, and translation vectors $\vec{T}_L$ and $\vec{T}_R$) of both cameras, relative to the pattern we've just fixed on the desk's surface. The extrinsic parameters R and $\vec{T}$ of the stereo rig are then simply

$$R = R_R R_L^\tau \qquad \vec{T} = \vec{T}_L - R^\tau \vec{T}_R$$

These parameters R and $\vec{T}$ now completely determine the geometry of our stereo rig and allow us to perform absolute, Euclidean 3D reconstruction of the hand's 3D position in the workspace shown in Figure 6.1.

Another important concept is the fundamental matrix F, a 3×3 matrix of rank 2, which allows us to perform 3D reconstruction using the triangulation method by see Hartley and Sturm [20]. We compute F from a set of N corresponding image points $\{\vec{x}_i \leftrightarrow \vec{x'}_i \mid i = 1, \ldots, N\}$, whereby we determine these image points as defining points of the squares of our calibration pattern (a 8×7 checkerboard, which can be seen for example in Figure 6.3 on page

62), detected in both the left and right image used for stereo rig calibration.

6.5.1 Hand detection

With the stereo rig calibrated, we can now proceed to detecting hands in the stereo input video stream. Here detection serves for the purpose of initiating the process of tracking, described in Section 6.5.3 below.

For this end, we define two "detection areas" (left and right), one for each of both hands in the application client area. By definition, if a hand is not being tracked, its "detection area" gets shown on the application screen as a red rectangle, at the predetermined location and with a predetermined size. If the user now moves her hand into the corresponding detection area, and puts her hand into the predefined posture (`HAND_POSTURE_OPEN` has been chosen as the tracking initialization posture), the system will detect the hand, output the corresponding bounding rectangle and start tracking the hand within this bounding rectangle. At this moment, the red rectangle disappears and is being replaced by a green rectangle, signifying that that particular hand is currently being tracked by the system.

For detection, the Viola-Jones method (see Appendix B on page 109) has been chosen as the detection method, due to the following properties:

- invariance with regard to background
- insensitivity to changes in illumination/lighting
- invariance with regard to person
- invariance with regard to camera
- invariance with regard to scale
- fast execution.

It is a method which requires training & validation using four sets of samples:

- Two *positive* sample sets:
  - **Positive training set** A — for this set we moved our right hand in posture `HAND_POSTURE_OPEN` randomly in the workspace, approximately under the natural inclination (see Figure 6.2 left), under our lab's standard lighting conditions, and took a number (in the range of hundreds) photos containing the hand. Note that "approximately natural inclination" indicates that we included a number of shots of hands rotated to a degree ($\pm 15°$) relative to all three axes, in order to increase the robustness of the classifier.

  - **Positive validation set** B — under the same conditions as above, we took additional photos (few hundreds) to be used as validation images after the training is complete.

- Two *negative* sample sets:

  - **Negative training set** C — for the negatives, we took a number (> 1000) of images that do not contain hands in posture `HAND_POSTURE_OPEN`, from two public domain image collections (burningwell.org, easystockphotos.com).
  - **Negative validation set** D — we took additional images as a negative validation set (a couple of hundreds), from the same public domain image collections.

We then used OpenCV facilities to train boosted cascades of weak classifiers in the following way:

1. we manually marked bounding rectangles for hands in the positive training samples, and saved the list of bounding rectangles in a file F.

2. before training, we set the required false positives threshold to be 10^{-6}.

3. we ran the training tool on file F and on the two training sets, the positive A and negative C. After the training has completed, we obtained a 15-stage classifier for posture `HAND_POSTURE_OPEN` with detection rate of approximately 98%.

4. we successfully tested the classifier using sets B and D.

5. we built the trained classifier into our prototype application. An OpenCV function loads the classifier; other function detects hands in posture `HAND_POSTURE_OPEN` in the current video frame. Note that we trained the classifier with our right hand; for the left hand, before the recognition stage we mirror the left side of the application area in order to be able to use the same classifier to recognize the left hand.

6.5.2 Hand segmentation based on human skin color

The hand detection method described above not only gives an answer whether there is or isn't a hand in an image, but also the bounding rectangle of the image region containing the hand. Considering this region of interest (ROI) only, we now make use of the characteristic hue of human skin to determine the pixels belonging to a hand. The reason we perform this segmentation is to increase the hand tracking robustness — see Section 6.5.3.

To this end, we used color histograms — both in the detection stage (using HSV color space), and for the learning (using normalized RGB histogram) of the color of the hand that has just been detected, i.e. we perform color learning immediately after the hand has been detected (pre-tracking stage).

6.5.3 Hand tracking

After a hand has been detected in an image, and hand pixels color-segmented using the properties of the human skin, we start tracking it. For tracking the method proposed by Kölsch in [35] has been chosen, which in turn is based on Kanade-Lucas-Tomasi (KLT) features (see Appendix C on page 116), also called "good features to track". KLT features are based on the early work done in [23], and then developed further in [24] and [25]. To increase robustness, the "Flocks of Features" approach to tracking by Kölsch adds two additional properties to simple KLT tracking:

- tracked KLT features never exceed a predetermined maximum distance from the median of all tracked KLT features, and
- tracked KLT features can never be closer to each other than a predetermined minimum distance.

Differently from the application showcased by Kölsch in [35], which is able to track only one hand using just one (monocular) camera, our application 1) implements four fully independent object trackers (four due to each camera tracking up to two hands), and 2) uses stereo disparity for 3D reconstruction of the hand's position in 3D workspace.

We now clarify what is meant by "tracking a hand". After a hand has been detected as explained in Section 6.5.1 in both cameras' views, and hand pixels color-segmented, up to N (for example, 100) KLT features are being collocated on the hand (i.e. on the blob defined by the detected hand's pixels). By averaging in each frame the 2D positions of all of these N features, we obtain a mean (average) position P of the hand being tracked. Therefore the

2D position P is the output of the tracking routine. Since we can have up to two active (tracked) hands, and each hand gives rise to one triangulated 3D position, we can have up to $2 \times 3 = 6$ d.o.f. at our disposal to implement spatial manipulation operations.

3D reconstruction (triangulation)

Finally, with the two corresponding 2D points u, u' tracked (point u in the left camera view, point u' it the right camera view, we can compute, in real time, the global 3D position $\vec{x} = (x, y, z)$ of a hand (either the left or the right hand) in the workspace. For this we use the triangulation method by Hartley and Sturm [20], a fast, non-iterative method that always finds the global optimum. Formulated as a least-squares minimization problem, the method computes image points $\hat{u}, \hat{u}'$ such that

$$d(u, \hat{u})^2 + d(u', \hat{u}')^2 \rightarrow \min, \qquad \hat{u}'^{\tau} F \hat{u} = 0$$

where $d(*, *)$ is the Euclidean distance function and F the fundamental matrix of the stereo rig (see Appendix D for more on the Hartley-Sturm triangulation method). Assuming Gaussian error distribution (tracking of u, u' is noisy because of digitization errors), the points $\hat{u}, \hat{u}'$ are the most likely values for true image correspondences. Since the corresponding rays through $\hat{u}, \hat{u}'$ meet *exactly* in 3D space, we can now find easily $\vec{x}$ (the global position of the hand in the workspace) using any other triangulation method, for example Mid-point triangulation method described in Section 5.4.1 on page 46.

6.5.4 Hand posture recognition

The last step in the CV pipeline is the hand posture recognition, which enables us to implement a simple static gesture recognition. For hand posture recognition, we again use the Viola-Jones method. For this we repeated the training process explained in Section 6.5.1, only with positive samples containing other postures besides `HAND_POSTURE_OPEN`.

6.5.5 Activity diagram

A detailed integrated view (an activity/control flow diagram) of all the CV-related processed described in this section can be seen in Figure 6.6.

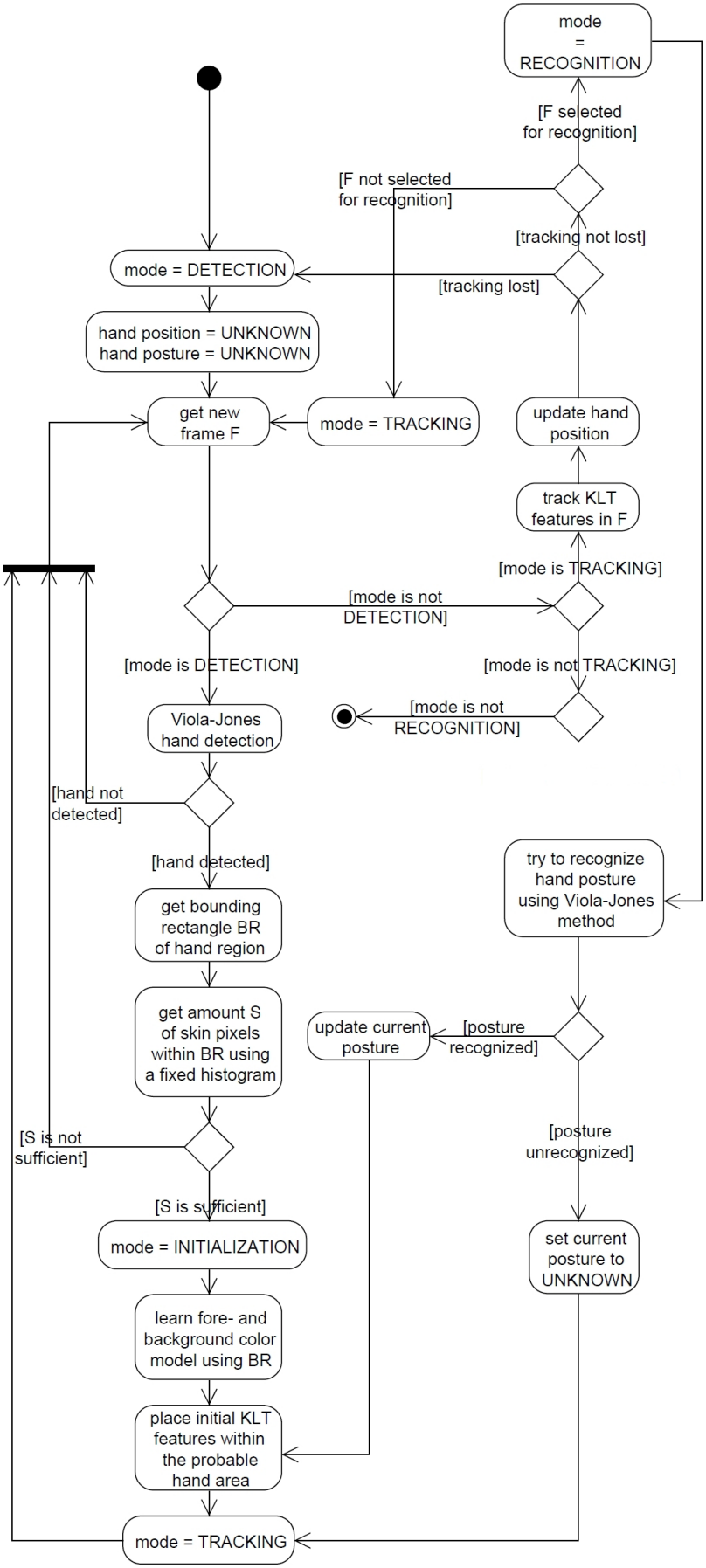

Figure 6.6: Detailed activity diagram for detection, tracking and posture recognition

6.6 Hardware and software configuration

All experiments were done on a personal computer equipped with an 2.66 GHz dual core processor, 2 GB RAM, and two web cameras connected to two dedicated USB 2.0 ports grabbing 30 color frames per second at the resolution of 320 × 240 pixels.

The software application was developed utilizing the C++ language, together with the following libraries:

- OpenGL for low-level 3D graphics rendering
- GLUT for windows handling
- OpenCV computer vision library for low-level image processing and extended Viola-Jones detection method (see Appendix B on page 109)
- A number of libraries were consulted and used to implement hand tracking based on KLT features (see Appendix C on page 116). These include:
  - KLT implementation by Jean-Yves Bouguet, found in the OpenCV library
  - KLT implementation by Stan Birchfield at the Clemson University (www.ces.clemson.edu/~stb/klt/)
  - KLT implementation by Mathias Kölsch found in the HandVu library (www.movesinstitute.org/~kolsch/HandVu/HandVu.html)
  - GPU-based KLT implementation by Sudipta Sinha at the University of North Carolina at Chapel Hill (cs.unc.edu/~ssinha/Research/GPU_KLT/)

6.7 Tests and results

We'll now assess qualitatively estimation accuracy for a hand's position. Since the difference between hand's estimated position and ground truth is difficult to measure for an uninstrumented hand, we give here the figures demonstrating the hand's trajectory in space, from which we can deduce visually the amount of noise present in estimated positions. We trace three simple figures in space with the right hand: a line, a circle and a figure "eight" (Figure 6.7).

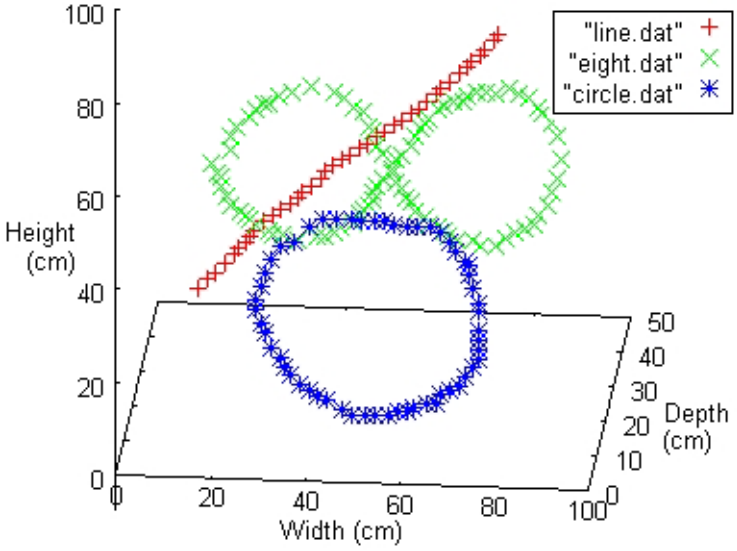

Figure 6.7: 3D plot of estimated hand positions, obtained by tracing a line, a circle and an "eight" in the workspace

6.7.1 Frames per second rates

Using the system described, we achieved tracking-related latencies from 7 to 30ms with just one hand tracked (i.e. with two trackers active), and up to 60ms with both hands tracked (i.e. with all four trackers active). Taking the application as a whole, i.e. taking all the other system processes into consideration, we achieved frame rates from 8 to 15 fps.

6.7.2 Detection performance

In this section the results on detection performance, depending on various training configurations, are presented. Viola-Jones detectors (see Appendix B on page 109) for all three hand postures were trained with various training parameters, and here their performance, relative to these parameters, will be compared.

The parameters which define detector performance are:

- **Number M of positives** (i.e. images that contain at least one instance of the hand in the targeted hand posture)
- **Number N of negatives** (i.e. images that don't contain any instance of the hand in the targeted hand posture)
- **Number K of stages** in the finalized cascaded detector
- **Minimum hit rate** α (for each stage). Note that the overall hit rate for the finalized detector is then α^K.
- **Maximum false alarm rate** β (for each stage). Note that the overall maximum false rate for the finalized detector is then β^K.

The performance parameters being measured (relative to some set containing test images — each of the three postures has its own such set) are:

- **Total hits** Θ — how many positive instances in the images from the test set were correctly detected.
- **Total misses** Γ — how many positive instances in the images from the test set weren't detected.
- **False hits** Ω — how many hits were reported, although there wasn't any positive instance (in the target posture) in the image from the test set, for all images in the set.

Note that by "detector performance" the manner of functioning in terms of hit rates only is presupposed. In other words, the speed of detection is not measured in this work, since in all cases it's good enough (in the range from 10 to 30 ms) to process all frames at the grabbing frame rates.

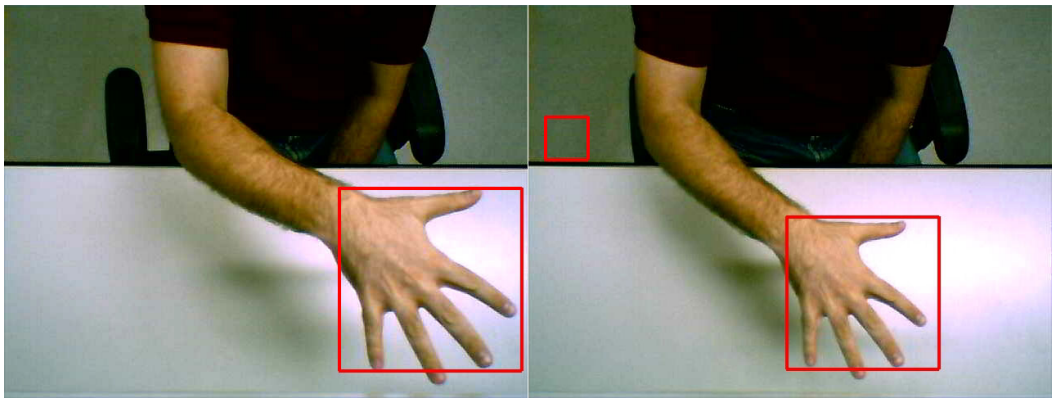

Figure 6.8: A hit (left) and a hit and false hit (right). Posture `HAND_POSTURE_OPEN`

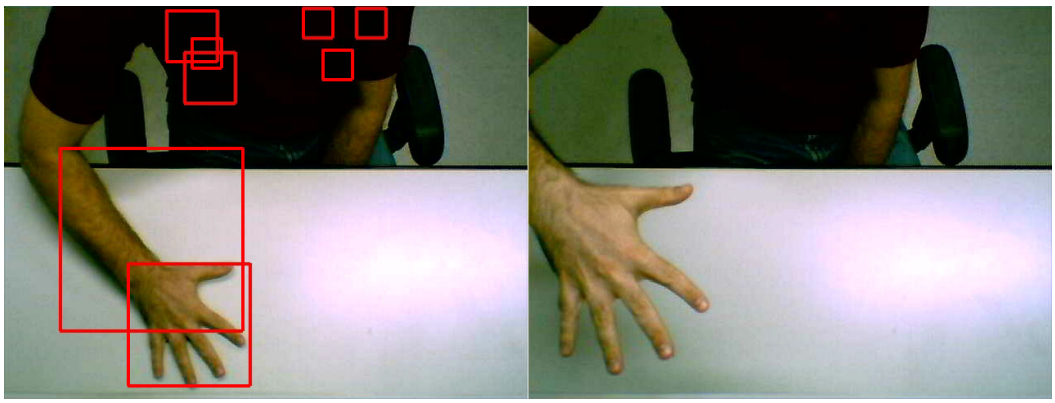

Figure 6.9: A hit and multiple false hits (left), and a miss (right). Posture `HAND_POSTURE_OPEN`

Training times for detectors trained ranged from a couple of hours to several days, using a current personal computer (2.4 GHz dual-core processor) and with 1 GB RAM allocated to the training process.

Detector performance for `HAND_POSTURE_OPEN`

Here we compare the performance of various detectors for the hand posture `HAND_POSTURE_OPEN`. To measure the performance, a set M' of 177 images has been created, some of which contained snapshots of the right hand in the hand posture `HAND_POSTURE_OPEN`, taken at various heights (y-axis) from the desk surface, and at various horizontal (x-axis) and depth (z-axis) locations.

	M	N	K	α	β
index	positives	negatives	number of stages	min. hit rate	max. false alarm rate
1	516	1000	8	0.995	0.5
2	545	1551	3	0.995	0.5
3	545	1551	3	0.995	**0.4**

Table 6.1: Training sets for `HAND_POSTURE_OPEN`

Please note training set #3 which has the same parameters as the set #2 however with maximum false alarm set at 0.4.

	M'	Θ	Γ	Ω
index	test set size	total hits	total misses	false hits
1	177	97 (54.80%)	80 (45.20%)	4 (2.26%)
2	177	154 (87.01%)	23 (12.99%)	813 (459.32%)
3	177	146 (82.46%)	31 (17.54%)	741 (418.64%)

Table 6.2: Detector performance for `HAND_POSTURE_OPEN`

We can see that the training set #1 achieved the highest number of stages and consequently the lowest false hit rate. Sets #2 and #3 achieved required leaf false alarm rate very early in the training phase, which however leads to unacceptably high false hit rates.

Detector performance for `HAND_POSTURE_POINTING`

Here we compare performance of various detectors for the hand posture `HAND_POSTURE_POINTING`. To measure the performance, a set M' of 162 (positive) images has been created some of which contained snapshots of the right

hand in the hand posture HAND_POSTURE_POINTING, taken at various heights (y-axis) from the desk surface, and at various horizontal (x-axis) and depth (z-axis) locations.

	M	N	K	α	β
index	positives	negatives	number of stages	min. hit rate	max. false alarm rate
1	233	1000	7	0.995	0.5
2	233	1000	10	0.995	**0.3**

Table 6.3: Training sets for HAND_POSTURE_POINTING

Training set #2 has the same parameters as the set #2 however with maximum false alarm set at 0.3.

	M'	Θ	Γ	Ω
index	test set size	total hits	total misses	false hits
1	162	115 (70.99%)	47 (29.01%)	1 (0.62%)
2	162	103 (63.58%)	59 (36.42%)	1 (0.62%)

Table 6.4: Detector performance for HAND_POSTURE_POINTING

As we can see, training sets #1 and #2 have the same false hit rates, however, for larger test sets, we can expect the set #2 to outperform #1 due to greater number of stages.

Detector performance for HAND_POSTURE_FIST

Here we compare performance of various detectors for the hand posture HAND_POSTURE_FIST. To measure the performance, a set M' of 191 (positive) images has been created some of which contained snapshots of the right hand in the hand posture HAND_POSTURE_FIST, taken at various heights (y-axis) from the desk surface, and at various horizontal (x-axis) and depth (z-axis) locations.

As we can see, training set #2 has the same parameters as the set #1 however with maximum false alarm set at 0.3. Training set #4 has the same parameters as the set #1 however with 1053 negatives instead of 1000, and only four stages. Also, training sets #3 and #4 have an increased number of positives (453). Sets #3 and #4 achieved required leaf false alarm rate very early in the training phase, which led to low number of stages for these detectors.

index	M positives	N negatives	K number of stages	α min. hit rate	β max. false alarm rate
1	222	1000	10	0.995	0.5
2	222	1000	10	0.995	**0.3**
3	453	1000	5	0.995	0.5
4	453	**1053**	**4**	0.995	0.5

Table 6.5: Training sets for `HAND_POSTURE_FIST`

index	M' test set size	Θ total hits	Γ total misses	Ω false hits
1	191	91 (47.64%)	100 (52.36%)	2 (1.05%)
2	191	78 (40.84%)	113 (59.16%)	3 (1.57%)
3	191	55 (28.80%)	136 (71.20%)	312 (163.35%)
4	191	63 (32.98%)	128 (67.02%)	406 (212.57%)

Table 6.6: Detector performance for `HAND_POSTURE_FIST`

We can see that the training set #2 has higher false hit rates than the set #1, however this is probably due to statistical fluctuation. For larger test sets, we can expect the set #2 to outperform #1 due to lower maximum false alarm rate.

An example session while working with the application

In this section, several snapshots of the video taken when working with the prototype application, are shown.

The scene consists of just two simple objects (wire-frame spheres) to be manipulated; by default, the right sphere is already selected, which is indicated by its red color. The left camera's image (of the stereo camera pair) is being rendered as a textured polygon at the bottom of the working volume, depicted here as a simple box delineated by a couple of gray lines.

Finally, the top left corner contains a picture-in-picture movie of hands, taken in real time with a third camera, in order to give a better overview of gestures and operations being performed.

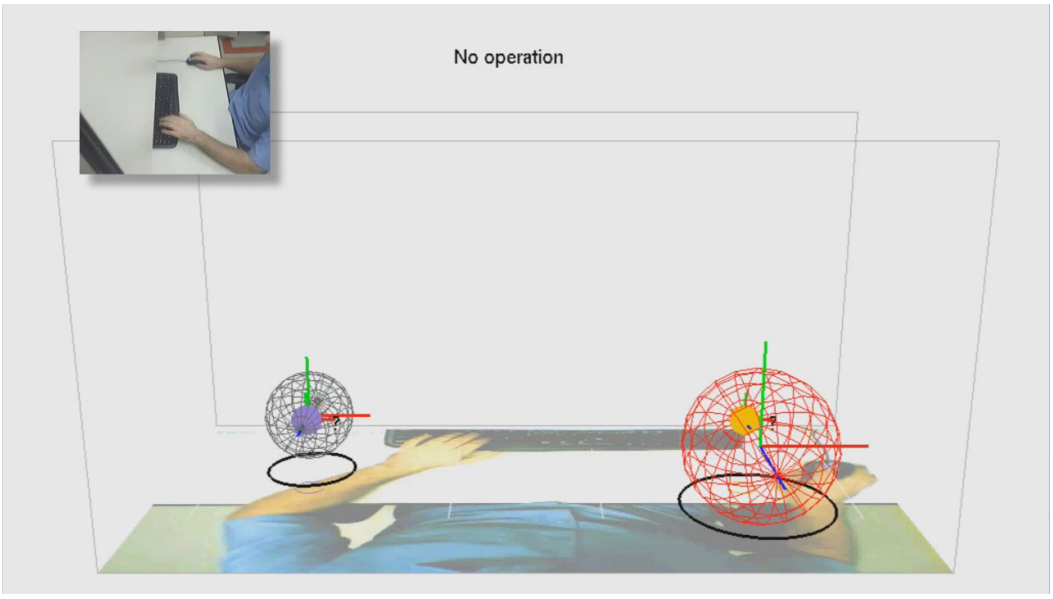

Figure 6.10: The application upon startup. No hand has been detected yet, therefore hands are not being tracked, thus no static gesture is being recognized, thus no manipulation operation is being performed

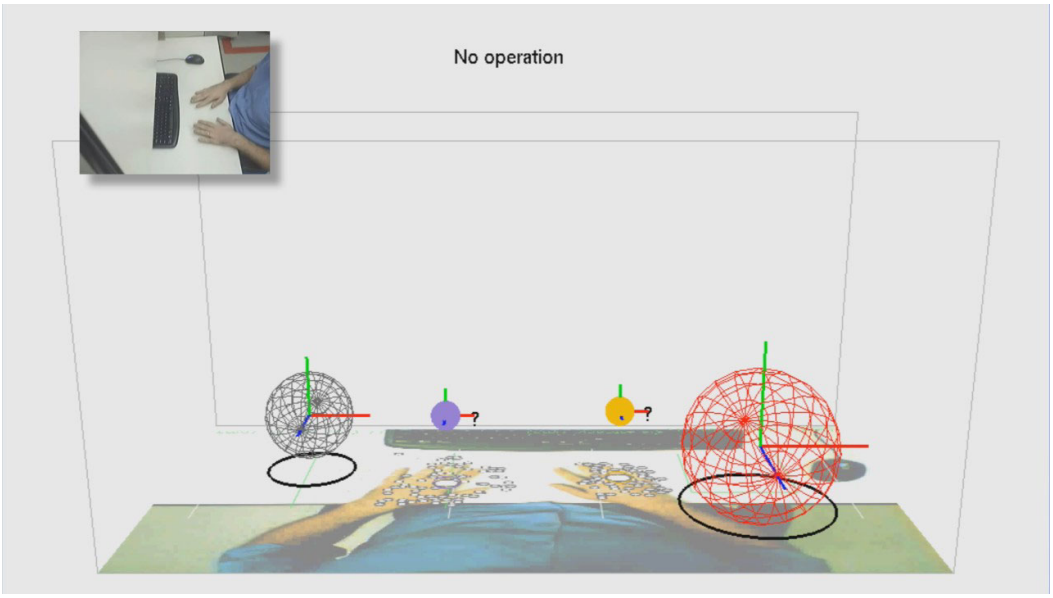

Figure 6.11: Application started to track hands, after both of them assumed posture `HAND_POSTURE_OPEN`. We can see that the application placed two flocks of KLT features on both hands

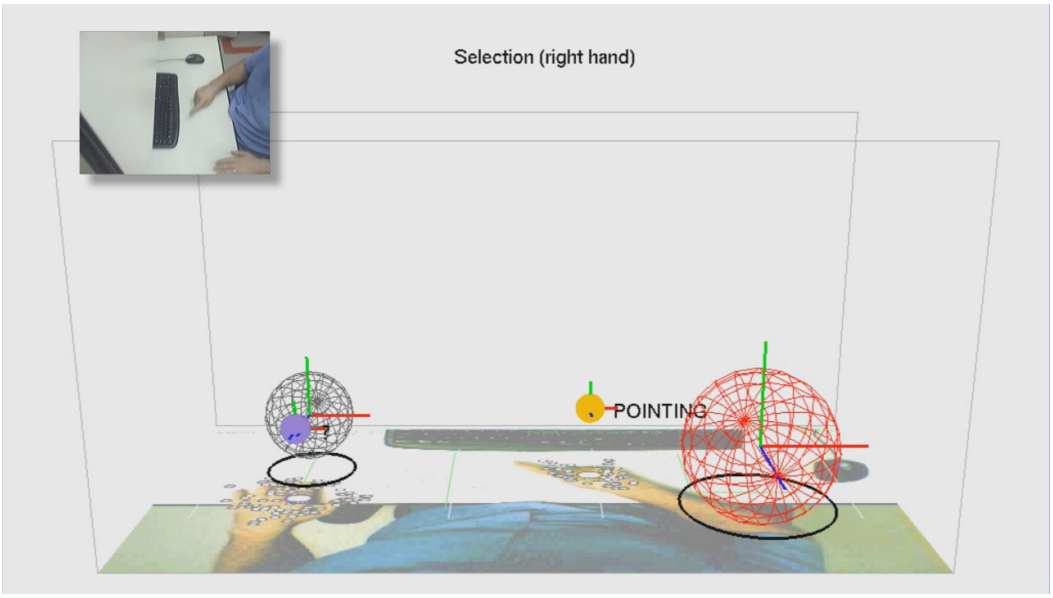

Figure 6.12: The right hand assumed posture `HAND_POSTURE_POINTING`, therefore the application started performing the operation `OP_SELECT` using the right hand

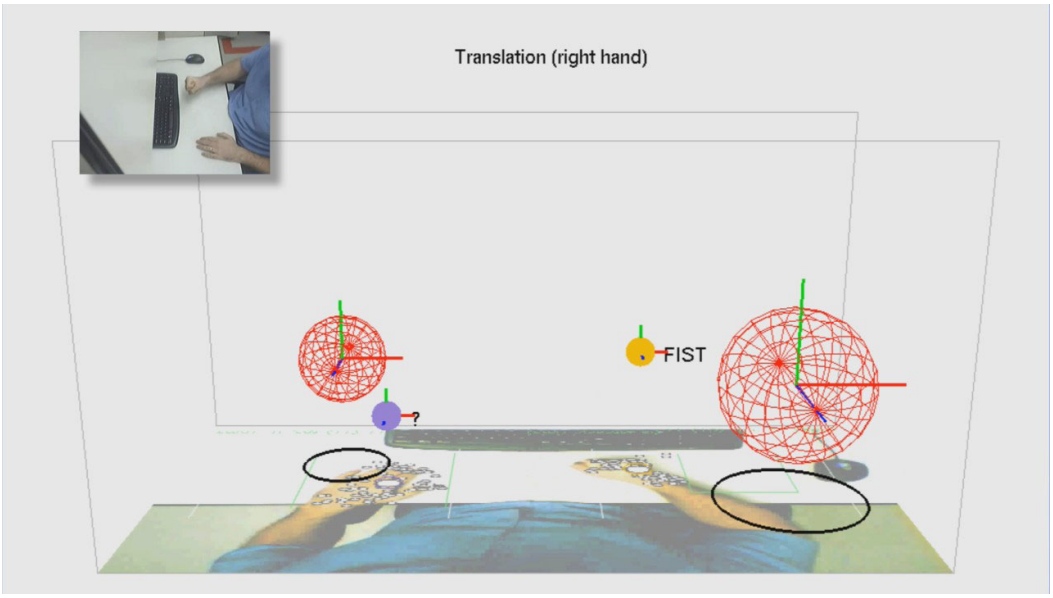

Figure 6.13: The right hand assumed posture `HAND_POSTURE_FIST`, therefore the application started performing the operation `OP_TRANSLATE` using the right hand

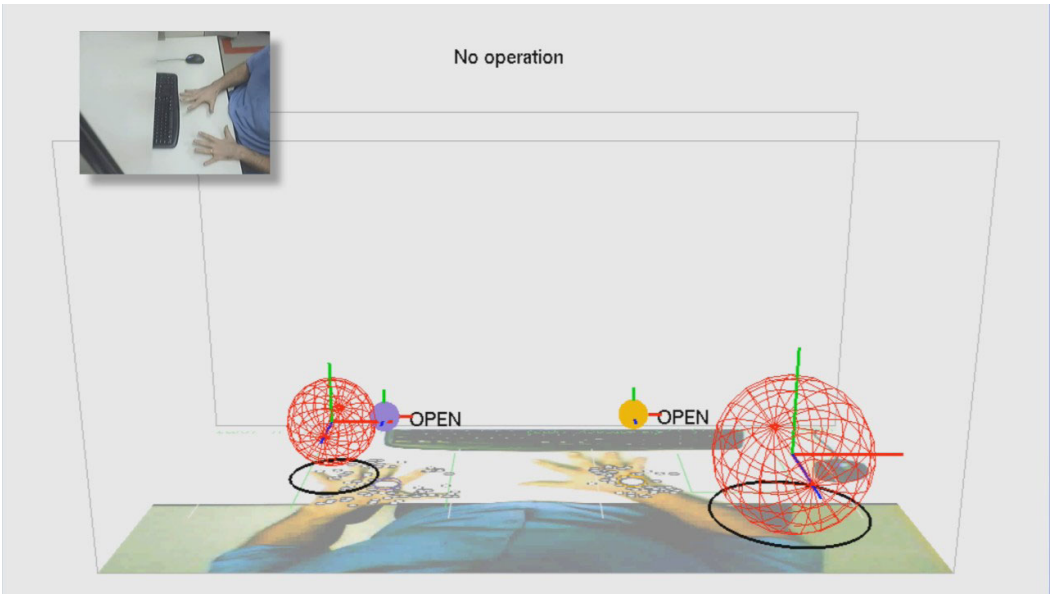

Figure 6.14: Both hands assumed posture `HAND_POSTURE_OPEN`, upon which the previous manipulation operation has been cancelled

Figure 6.15: The left hand assumed posture `HAND_POSTURE_POINTING`, therefore the application started performing the operation `OP_SELECT` using the left hand

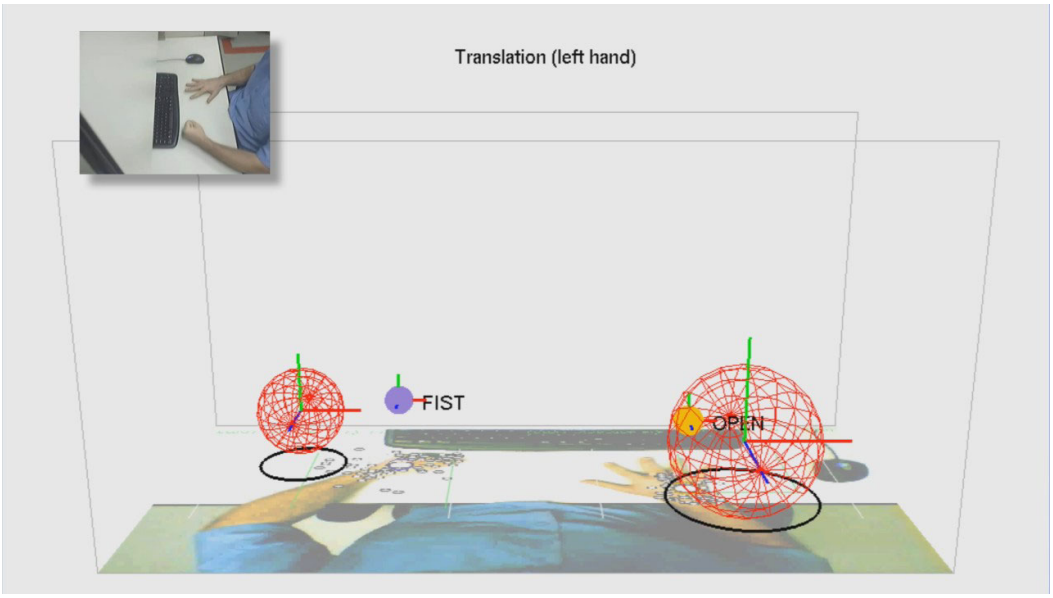

Figure 6.16: The left hand assumed posture `HAND_POSTURE_FIST`, therefore the application started performing the operation `OP_TRANSLATE` using the left hand

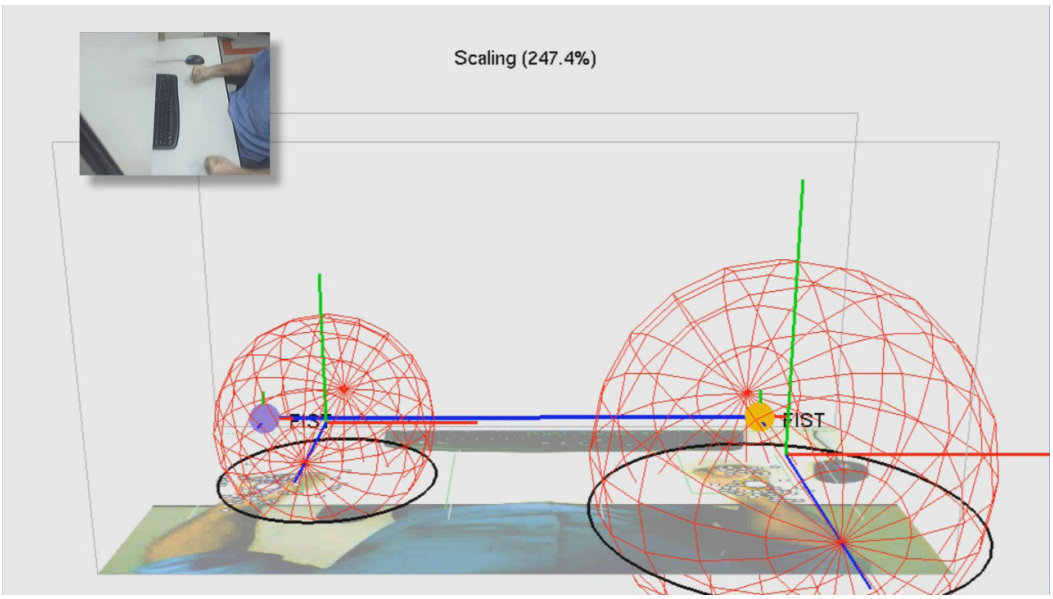

Figure 6.17: Both hands assumed posture `HAND_POSTURE_FIST`, therefore the application started performing the operation `OP_SCALE` using both hands

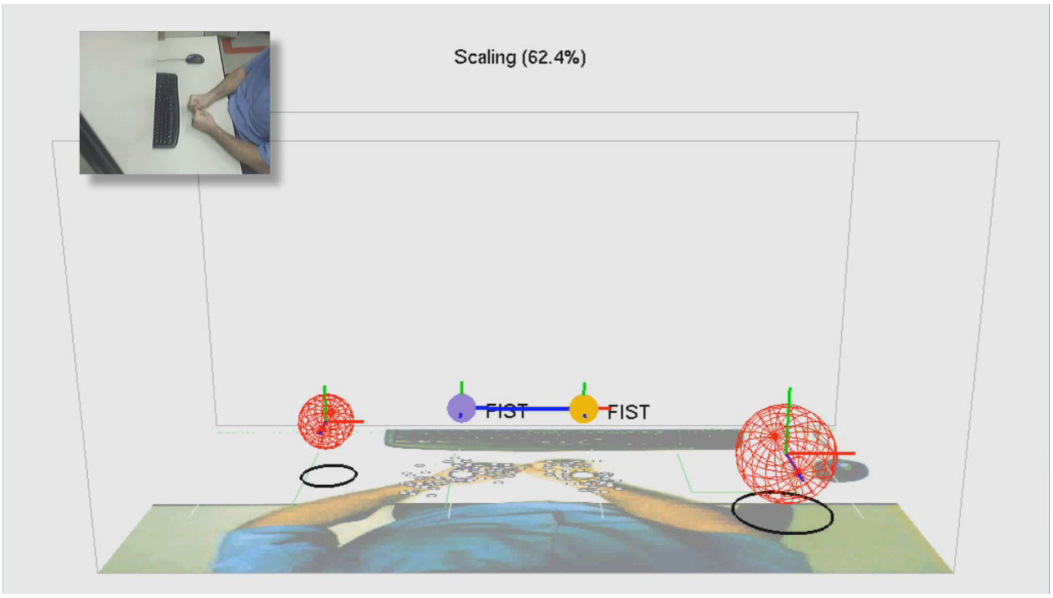

Figure 6.18: Another example of `OP_SCALE`

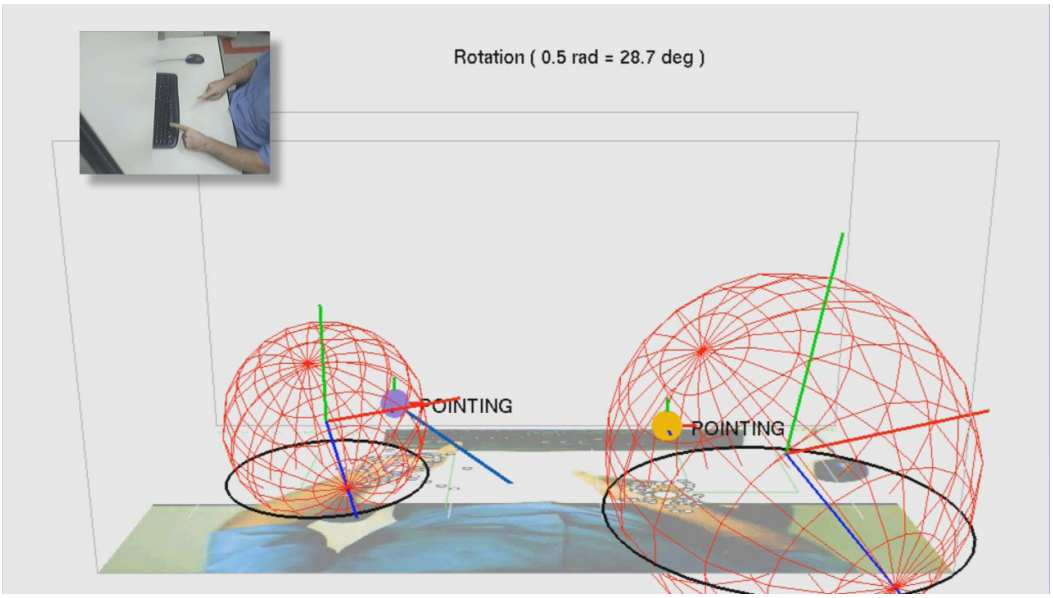

Figure 6.19: Both hands assumed posture `HAND_POSTURE_POINTING`, therefore the application started performing the operation `OP_ROTATE` using both hands