Particle Filtering for Visual Tracking

Pedram Azad

Contents

1

Particle Filtering for Visual Tracking

Author: Pedram Azad

In this chapter, particle filters and their application to visual tracking are introduced. After giving a formal definition of the general particle filter, commonly used likelihood functions for evaluating a model configuration in the context of visual tracking are introduced. Each of these likelihood functions can be regarded as a separate cue, operating on extracted image information that is specific to it. Finally, the general approach for fusing several cues within a particle filter is presented. The Sections 1.2–1.4 explain the model and the cues by the example of markerless human motion capture. The concepts can be used analogously for the tracking of rigid objects, as mentioned in Section 1.5

1.1 Particle Filters

Particle filters, also known as Sequential Monte-Carlo methods, are sampling-based Bayesian filters. A variant of the particle filter is the Condensation algorithm (**Con**ditional **Dens**ity Propag**ation**) [Isard and Blake, 1996, Isard and Blake, 1998], which has become popular in the context of vision-based tracking algorithms. In [Blake and Isard, 1998], the application of the Condensation algorithm to the tracking of *active contours* is presented. In the following, given a pre-specified model, a particle filter is understood as an estimator, which tries to determine the model configuration that matches the current observations in the best possible way. The general particle filtering framework is implemented in the class `CParticleFilterFramework` from the IVT (see Chapters 2 and 3).

Particle filters approximate the probability density function of a probability distribution, modeling the estimator's state by a fixed number of particles. A particle is a pair $(\boldsymbol{s}, \pi)$, where $\boldsymbol{s} \in \mathbb{R}^n$ denotes a model configuration and π is an associated likelihood; n denotes the dimensionality of the model. The state of the set of all N particles at the discrete time step t is denoted as:

$$X_t = \{(\boldsymbol{s}_t^{(i)}, \pi_t^{(i)})\} \tag{1.1}$$

with $i \in \{1, \ldots, N\}$ and $t \geq 0$. Particle filtering is an iterative algorithm operating on this set. In each iteration, the following three steps are performed:

1. Draw n particles from X_{t-1}, proportionally to their likelihood.
2. Sample a new configuration $\boldsymbol{s}_t$ for each drawn particle $(\boldsymbol{s}_{t-1}, \pi_{t-1})$.
3. Compute a new likelihood π_t for each new configuration $\boldsymbol{s}_t$ by evaluating the likelihood function $p(\boldsymbol{z}_t \mid \boldsymbol{s}_t)$.

where $p(\boldsymbol{z} \mid \boldsymbol{s})$ is a likelihood function that computes the a-posteriori probability of the configuration $\boldsymbol{s}$ matching the observations $\boldsymbol{z}$. Here, $\boldsymbol{z}$ is an abstract variable that stands for any type of data, e.g. for image data in the case of vision-based tracking algorithms. The three steps of a particle filter iteration are summarized in Algorithm 1 in pseudo code.

Algorithm 1 ParticleFilter$(X_{t-1}, \boldsymbol{z}_t) \rightarrow X_t$

```
$X_t := \{ \}$
**for** $k := 1$ **to** $N$ **do**
  Draw $i$ with probability $\propto \pi_{t-1}^{(i)}$
  Sample $\boldsymbol{s}_t^{(k)} \propto p(\boldsymbol{s}_t \mid \boldsymbol{s}_{t-1}^{(i)})$
  $\pi_t^{(k)} := p(\boldsymbol{z}_t \mid \boldsymbol{s}_t^{(k)})$
  $X_t := X_t \cup \{(\boldsymbol{s}_t^{(k)}, \pi_t^{(k)})\}$
**end for**
```

Drawing a particle $(\boldsymbol{s}_t^{(i)}, \pi_t^{(i)})$ with probability $\propto \pi_{t-1}^{(i)}$ can be accomplished efficiently by binary subdivision [Isard and Blake, 1998] (see method `CParticleFilterFramework::PickBaseSample` of the IVT.

Sampling a new particle from $p(\boldsymbol{s}_t \mid \boldsymbol{s}_{t-1}^{(i)})$ is usually accomplished by taking into account a dynamic model of the object to be tracked, and by adding Gaussian noise for handling unpredictable movements. The given formulation for the sampling step is specific to the Condensation algorithm. In the general particle filter, new particles are sampled from $p(\boldsymbol{s}_t \mid \boldsymbol{s}_{t-1}^{(i)}, \boldsymbol{z}_t)$, i.e. the current observations are taken into account in the sampling step as well.

If no dynamic model is used, then the dimensionality of the model equals the number of DoF to be estimated, for instance $n = 2$ for a 2D tracking problem. Sampling is performed merely by adding noise:

$$\boldsymbol{s}_t^{(k)} = \boldsymbol{s}_{t-1}^{(i)} + B\,\boldsymbol{\omega} \tag{1.2}$$

where $\boldsymbol{\omega} \in \mathbb{R}^n$ denotes Gaussian noise i.e. the components of the vector are sampled (independently) from a Gaussian distribution. $B \in \mathbb{R}^{n,n}$ is a diagonal matrix that contains weights for the components of $\boldsymbol{\omega}$.

When using a constant velocity model, one common approach is to estimate the velocity as well, by incorporating the velocity into the particles. Then, for a 2D tracking problem, it would apply $n = 4$. For this purpose, the configuration $\boldsymbol{s} \in \mathbb{R}^n$ is split up into a position part $\boldsymbol{x} \in \mathbb{R}^{n/2}$ and a velocity part $\boldsymbol{v} \in \mathbb{R}^{n/2}$, i.e. $\boldsymbol{s} := (\boldsymbol{x}, \boldsymbol{v})$. Sampling is then performed as follows:

$$\begin{aligned} \boldsymbol{x}_t^{(k)} &= \boldsymbol{x}_{t-1}^{(i)} + \Delta t\, \boldsymbol{v}_{t-1}^{(i)} + B_x\,\boldsymbol{\omega} \\ \boldsymbol{v}_t^{(k)} &= \boldsymbol{v}_{t-1}^{(i)} + B_v\,\boldsymbol{\omega} \end{aligned} \tag{1.3}$$

where $B_x, B_v \in \mathbb{R}^{n/2,n/2}$ and Δt denotes the time elapsed between the discrete time steps t and $t-1$. In order to keep the characteristics of a constant velocity model, the magnitudes of the components of the diagonal matrix B_x must be small. If B_x is chosen to be the zero matrix, then the position is strictly determined by the estimated velocity from the previous particle filter iteration. Note that the likelihood function $p(\boldsymbol{z} \,|\, \boldsymbol{s})$ is usually implemented as $p(\boldsymbol{z} \,|\, \boldsymbol{x})$ in this case, i.e. the velocity is not incorporated explicitly in the weighting function.

Finally, the estimation of a particle filter after an iteration is usually calculated by the weighted mean $\overline{\boldsymbol{s}}$ over all particles:

$$\overline{\boldsymbol{s}} := \sum_{k=1}^{N} \pi_k \cdot \boldsymbol{s}_k \tag{1.4}$$

Various extensions to the standard particle filtering scheme have been proposed. Among these are the partitioned sampling theory [MacCormick and Isard, 2000, MacCormick, 2000], annealed particle filtering [Deutscher et al., 2000, Deutscher et al., 2001], Rao-Blackwellization [Casella and Robert, 1996, Doucet et al., 2000], and auxiliary particle filters [Pitt and Shepard, 1999]. Switching between dynamic models in particle filters has been proposed in [Bando et al., 2004].

1.2 Problem Definition of Human Motion Capture

The general problem definition is to determine the correct configuration of an underlying articulated 3D human model for each input image or image tuple, respectively. The main problem is that search space increases exponentially with the number of degrees of freedom. A realistic model of the human body consists of at least 25 DoF: 6 DoF for the base transformation, 3 DoF for the neck, $2 \cdot 4$ DoF for the arms, and, $2 \cdot 4$ DoF for the legs; or 17 DoF if only modeling the upper body. The large number of degrees of freedom in both cases leads to a very high-dimensional search space.

1.3 Human Upper Body Model

1.3.1 Kinematics Model

For the system presented in [Azad et al., 2006b, Azad et al., 2007b], a kinematics model of the human upper body consisting of 14 DoF was used, not modeling the neck joint. The shoulder is modeled as a ball joint, and the elbow as a hinge joint. With this model, rotations around the axis of the forearm cannot be modeled. Capturing the forearm rotation would require tracking of the hand, which can be regarded a separate problem. The degrees of freedom of the used upper body model are summarized as follows:

- Base transformation: 6 DoF
- Shoulders: $2 \cdot 3$ DoF
- Elbows: $2 \cdot 1$ DoF

The shoulder joints are implemented by an axis/angle representation in order to avoid problems with singularities, which can occur when using Euler angles. The elbows are modeled by the single angle θ for the rotation matrix $R_x(\theta)$. The base rotation is modeled by Euler angles to allow a better imagination so that joint space restrictions can be defined easily.

1.3.2 Geometric Model

Body sections are often fleshed out by sections of a cone with an elliptic cross-section. However, using ellipses instead of circles for modeling the arms causes additional computational effort without leading to considerable practical benefits for application with image-based human motion capture systems. In practice, only the torso requires ellipses for being modeled with sufficient detail. But even with such a model, the torso can hardly be tracked on the basis of projections to the image i.e. without explicit 3D information. The reason is that the only valuable information that could be measured in images are the left and right contour and the shoulder positions. However, the contour is often occluded and changes its appearance depending on the clothing and the arm configuration, as does the appearance of the shoulders, making it hard to track the torso on the basis of edge or region information only.

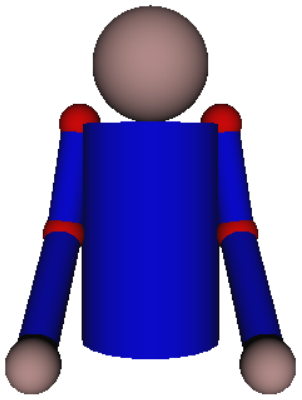


Fig. 1.1. 3D Visualization of the used human upper body model.

In [Azad et al., 2006b, Azad et al., 2007b], all body parts are modeled by sections of a cone with circular cross-sections. The computation of the projected contour of such a 3D primitive is described in the following; the calculations for the more general case with an elliptic cross-section are given in [Azad et al., 2004]. A section of a cone is defined by the center $\boldsymbol{c}$ of the base, the radius R of the base circle, the radius r of the top circle, and the direction vector $\boldsymbol{n}$ of the main axis. Given that all vectors are specified in the camera coordinate system, the position vectors of the endpoints $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{p}_4$ defining the two projected contour lines $\overline{P_1P_2}$ and $\overline{P_3P_4}$ can be calculated as follows:

$$\begin{aligned}
\boldsymbol{u} &= \frac{\boldsymbol{n} \times \boldsymbol{c}}{|\boldsymbol{n} \times \boldsymbol{c}|} \\
\boldsymbol{c}_t &= \boldsymbol{c} + L \cdot \frac{\boldsymbol{n}}{|\boldsymbol{n}|} \\
\boldsymbol{p}_{1,3} &= \boldsymbol{c} \pm R \cdot \boldsymbol{u} \\
\boldsymbol{p}_{2,4} &= \boldsymbol{c}_t \pm r \cdot \boldsymbol{u}
\end{aligned} \tag{1.5}$$

All involved measures and vectors are illustrated in Fig. 1.2. The principle is to calculate the intersection of the plane that runs through the main axis of the cone and at the same time is orthogonal to the plane that goes through the projection center and the main axis. As can be seen, using this model, only very few computations are necessary for calculating the projected contour of a body part. This is crucial for the goal of building a system that can be applied in real-time, since the projection of the body model given a configuration must be evaluated for each particle.

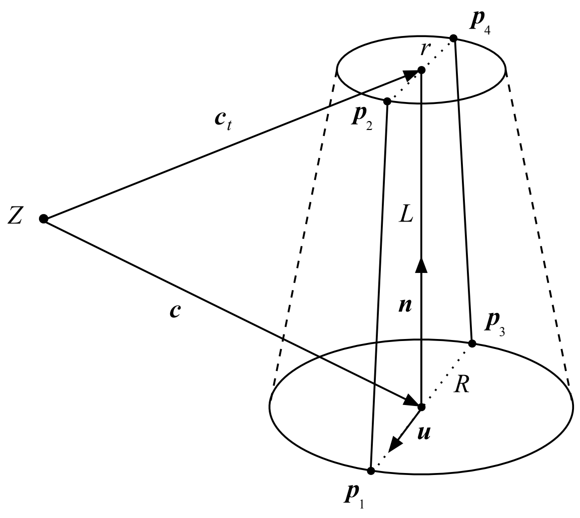


Fig. 1.2. Illustration of the projection of the contour of a section of a cone.

1.4 General Particle Filter Framework for Human Motion Capture

Particle filtering has become popular for various visual tracking applications. The benefits of a particle filter compared to a Kalman filter are the ability to track non-linear movements and the property to store multiple hypotheses simultaneously. These benefits are bought at the expense of a higher computational effort. For methods using a Kalman filter as probabilistic framework, it is typical that an optimization approach is used in the core. The robustness and accuracy of the system is improved by using a Kalman filter for predicting the configuration in the next frame. This prediction yields a considerably better initial condition for the optimization approach. However, tracking approaches that rely on a sufficiently accurate prediction naturally lack the ability to recover when tracking gets lost.

Approaches relying on particle filtering follow a completely different strategy. Instead of using an optimization method, essentially a statistically profound search is performed for finding the optimal solution. In [Deutscher et al., 2000], it was shown that powerful markerless human motion capture systems operating on images can be built using particle filtering. The proposed system uses three cameras distributed around the area of interest, performs figure-ground segmentation by background subtraction, and has a processing time of approx. 15 seconds per frame on a 1.5 GHz CPU. Nevertheless, the used likelihood functions and the general approach build a valuable starting point for building a system that can be applied on the active head of a humanoid robot system. In the following, these likelihood functions, namely the edge cue and the region cue, are introduced. In [Azad et al., 2006b], an additional distance cue is introduced, incorporating tracked 3D positions of the hands and the head of the person of interest. A system that is capable of robust tracking of upper body motion of a person with a humanoid active head is explained in detail in [Azad, 2008b].

1.4.1 Edge Cue

Given the current observations $\boldsymbol{z}$ and the projected contour of the human model for a configuration $\boldsymbol{s}$, the likelihood function $p(\boldsymbol{z} \mid \boldsymbol{s})$ for the edge cue calculates the likelihood that the model matches the observations $\boldsymbol{z}$, given a configuration $\boldsymbol{s}$.

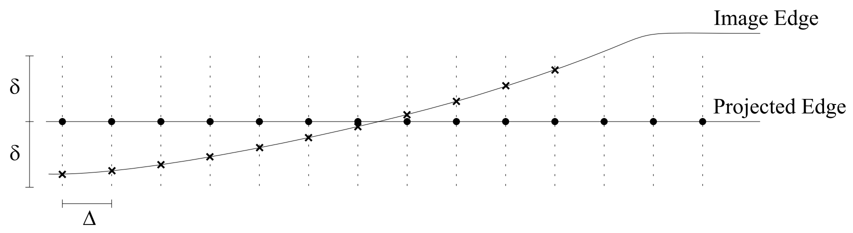


Fig. 1.3. Illustration of the search for edge pixels.

The basic technique is to traverse the projected edges and search at fixed distances Δ for edge pixels in perpendicular direction to the projected edge within a fixed search distance δ, as illustrated in Fig. 1.3 [Isard and Blake, 1996]. For this purpose, the camera image I, which represents the observations $\boldsymbol{z}$, is usually pre-processed to generate a gradient image I_g using an edge filter. The likelihood is calculated on the basis of the SSD. In order to formulate the likelihood function, first for a given point $\boldsymbol{p} \in \mathbb{R}^2$ belonging to an edge $e_{\boldsymbol{p}}$, the set of high-gradient pixels in perpendicular direction to $e_{\boldsymbol{p}}$ is defined by the function $g(I_g, \boldsymbol{p})$:

$$g(I_g, \boldsymbol{p}) = \{ \ (\boldsymbol{x} - \boldsymbol{p})^2 \mid \boldsymbol{x} \in \mathbb{R}^2 \wedge |\boldsymbol{x} - \boldsymbol{p}| \leq \delta \wedge (\boldsymbol{x} - \boldsymbol{p}) \perp e_{\boldsymbol{p}} \wedge I_g(\boldsymbol{x}) \geq t_g \ \} \quad (1.6)$$

where t_g denotes a pre-defined gradient threshold. Given a set of projected contour points $P := \{\boldsymbol{p}_i\}$ with $\boldsymbol{p}_i \in \mathbb{R}^2$, $i \in \{1, \ldots, |P|\}$, the evaluation or error function $w(I_g, P)$ can be formulated as follows:

$$w_g(I_g, P) = \sum_{i=1}^{|P|} \min \left(g(I_g, \boldsymbol{p}_i), \mu^2 \right) \quad (1.7)$$

where μ denotes a penalty distance that is applied in case no high-gradient pixel could be found for a contour point $\boldsymbol{p}_i$ i.e. $g(I_g, \boldsymbol{p}_i) = \{ \ \}$. The notation for the likelihood function now reads:

$$p_g(I_g \mid \boldsymbol{s}) \propto \exp\left\{-\frac{1}{2\sigma_g^2} w_g(I_g, f_g(\boldsymbol{s}))\right\} \tag{1.8}$$

The point set P is acquired by applying the function $f_g(\boldsymbol{s})$, which computes the forward kinematics of the human model and projects the model's contour points of interest to the image coordinate system. A contour line of the model is projected to the image by projecting its two endpoints (see Section 1.3.2). The projection is performed by applying the projection matrix of the camera, which has been computed by the calibration procedure beforehand. Having projected the two endpoints, the line is sampled in the image with the discretization Δ.

Another approach for a gradient-based evaluation function is to spread the gradients in the gradient image I_g by applying a Gaussian filter or any other suitable operator, and to add up the gradient values along a projected edge, as done in [Deutscher et al., 2000]. By doing this, the computational effort is reduced significantly, compared to performing a search perpendicular to each pixel of the projected edge. The computation of the evaluation function is efficient, even when choosing the highest possible discretization of $\Delta = 1$ pixel. Assuming that the spread gradient image has been remapped to the interval [0, 1], the evaluation function can be formulated as:

$$w_g(I_g, P) = \frac{1}{|P|} \sum_{i=1}^{|P|} (1 - I_g(\boldsymbol{p}_i))^2 \tag{1.9}$$

1.4.2 Region Cue

The second cue commonly used is region-based, for which a figure-ground segmentation technique has to be applied. The segmentation algorithm is independent from the likelihood function itself. In the segmentation result I_r, pixels belonging to the person's silhouette are set to 1 and background pixels are set to 0, i.e. $I(u, v) \in \{0, 1\}$. The evaluation function for the region cue can then be formulated as [Deutscher et al., 2000]:

$$w_r(I_r, P) = \frac{1}{|P|} \sum_{i=1}^{|P|} (1 - I_r(\boldsymbol{p}_i))^2 = 1 - \frac{1}{|P|} \sum_{i=1}^{|P|} I_r(\boldsymbol{p}_i) \tag{1.10}$$

The main difference to the edge cue is that not projected contour points of the model are sampled but points within the projected contour. This leads to a considerably higher computational effort, since the points are sampled in a grid rather than along a line. The likelihood function for the region cue finally reads:

$$p_r(I_r \mid \boldsymbol{s}) \propto \exp\left\{-\frac{1}{2\sigma_r^2} w_r(I_r, f_r(\boldsymbol{s}))\right\} \tag{1.11}$$

where the function $f_r(\boldsymbol{s})$ computes the forward kinematics of the human model and projects the model's body part points to the image coordinate system. This is achieved by computing a grid within the area defined by the four projected contour endpoints $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{p}_4$ (see Section 1.3.2).

1.4.3 Fusion of Multiple Cues

The general approach for fusing the results of multiple cues within a particle filtering framework is to multiply the likelihood functions of the cues in order to obtain an overall likelihood function. For the introduced edge cue and region cue, this would yield:

$$\begin{aligned} p(I_g, I_r \mid \boldsymbol{s}) &\propto \exp\left\{-\frac{1}{2\sigma_g^2} w_g(I_g, f_g(\boldsymbol{s}))\right\} \cdot \exp\left\{-\frac{1}{2\sigma_r^2} w_r(I_r, f_r(\boldsymbol{s}))\right\} \\ &= \exp\left\{-\left[\frac{1}{2\sigma_g^2} w_g(I_g, f_g(\boldsymbol{s})) + \frac{1}{2\sigma_r^2} w_r(I_r, f_r(\boldsymbol{s}))\right]\right\} \qquad (1.12) \end{aligned}$$

Any other cue can be fused within the particle filter with the same rule. One way of combining the information provided by multiple calibrated cameras is to incorporate the likelihoods for each image in the exact same manner, as done in [Deutscher et al., 2000]. This technique can also be used for combining the likelihood functions for the left and right camera image. However, when combining cues with different characteristics, fusion with this method often results in noisy estimations and partly unstable behavior. In order to overcome this problem, in [Azad, 2008b], a prioritized fusion method for fusing the edge cue and a so-called distance cue [Azad et al., 2006b] is proposed.

1.5 Rigid Object Tracking

The introduced cues can be used for the tracking of rigid objects as well. With rigid objects, the problem of tracking becomes more tractable, since the dimensionality of the search space is reduced six: 3 DoF for the position and 3 DoF for the orientation. As a model-based rigid object tracking algorithm using edge information one usually understands a method that relies on a 3D rigid object model (usually CAD), which usually consists of a number of primitives. Most often these primitives are (straight) lines, since their projection can be computed very efficiently by the projection of the two endpoints. Various non-curved 3D objects can be modeled using lines, such as cuboids or pyramids, as illustrated in Fig. 1.4. Furthermore, other feasible 3D primitives are cones and cylinders, for each of which the two characteristic contour lines can be calculated with few additional computational effort; the (semi) ellipses can be neglected. Note that any 3D shape for which the projection of curved surfaces is crucial is problematic and cannot be tracked with the same approach. An object recognition, pose estimation, and tracking approach for arbitrary shapes combining stereo vision with model-based acquisition of views for an appearance-based method is described in [Azad et al., 2006a, Azad, 2008b].

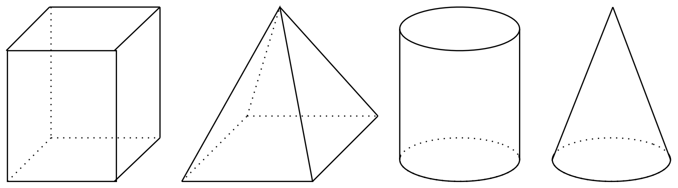


Fig. 1.4. Examples of simple 3D models suitable for model-based tracking.

As explained in Section 1.1, the goal is to find the model configuration that maximizes a likelihood function which is specific to the problem. In the case of edge-based rigid object tracking, the space of model configurations is the space of possible object poses $\in \mathbb{R}^6$, and the likelihood function measures the number of edge pixels along projected model edges (see Section 1.4.1). The computational effort for such particle filter based approaches depends on the number of evaluations of the likelihood function per frame. In conventional particle filters, this number is equal to the number of particles.

In [Klein and Murray, 2006], a real-time system for rigid object tracking using particle filtering is proposed. In order to achieve real-time performance, the evaluation of the likelihood function is implemented on the GPU of a graphics card. Particle filtering is performed in two stages: First, an automatically adjusted number of particles is used on the down-sampled input image of size 320×240, together with a broader i.e. smoothed likelihood function. In the second stage, 100 particles are used with a peaked likelihood function, which is applied on the input image at full resolution. The system achieves a processing rate of 30 fps on a 3.2 GHz CPU with an nVidia GeForce 6800 graphics card.

In the most simple case of 2D position tracking of a segmented blob, the region cue presented in Section 1.4.2 can be used with a square model. The evaluation of the likelihood function can be speeded up significantly by using *integral images*, also referred to as *summed area tables* (for an implementation see the functions `ImageProcessor::CalculateSummedAreaTable` and `ImageProcessor::GetAreaSum` of the IVT). The problem with 3D position tracking of a segmented blob using the region cue is that the evaluation function from Eq. (1.10) prefers small areas. Therefore, a 3D position tracker would tend to $z \to \infty$. One possible solution is to modify Eq. (1.10) by division by the z-coordinate, so that positions at far distances are not blindly preferred, yielding:

$$w_r(I_r, P) = -\frac{1}{z\,|P|}\sum_{i=1}^{|P|} I_r(\boldsymbol{p}_i) \tag{1.13}$$

The leading 1 from Eq. (1.10) is omitted, since the range of possible values is not fixed when dividing by the z-coordinate. An effective solution is to collect all results of Eq. (1.13) for one particle set, then linearly map the values to the interval $[0,\ 1]$, before applying the exponential function. From experience, a weighting factor of $s_s = 10$ leads to good result:

$$p_r(I_r \mid \boldsymbol{s}) \propto \exp\left\{-s_s \cdot w'_r(I_r, f_r(\boldsymbol{s}))\right\} \tag{1.14}$$

where w'_r denotes the values after mapping to the interval $[0,\ 1]$. The necessary computations, including sampling a square in an image for a 3D square model, are summarized in the Algorithms 2 and 3. Here, f denotes the focal length, s the side length of the square in 3D, and $I_{s,l},\ I_{s,r}$ denote the segmentation results for a stereo image pair. Implementations in the form of example applications can be found in `IVT/examples/TrackingApp` and `IVT/win32/TrackingApp`, respectively.

Tracking a segmented blob with the presented method results in a stable tracker, which is robust to image noise, partial occlusions, and also succeeds for imperfect segmentation results, e.g. unconnected sub-blobs of a blob. A problem that remains is that the estimated 3D position, in particular the z-coordinate, is not accurate. However, the estimate can be refined to an accurate 3D position on the basis of a blob analysis and correlation-based stereo triangulation, as shown in [Azad, 2008b].

Algorithm 2 ProjectAndSampleSquare($\boldsymbol{p}$, s) $\rightarrow P$

```
1. $(x,\ y,\ z)^T = \boldsymbol{p}$
2. $(u, v) \leftarrow$ CalculateImageCoordinates($\boldsymbol{p}$)
3. $k := \frac{s \cdot f}{2z}$
4. Sample points from $[u-k,\ u+k] \times [v-k,\ v+k]$ and store the 2D positions in
   the point set $P$.
```

Algorithm 3 ComputeProbabilities($I_{s,l}$, $I_{s,r}$, $\{\boldsymbol{p}_i\}$) $\rightarrow \{\pi_i\}$

```
1. $N := |\{\boldsymbol{p}_i\}|$  { number of particles }
2. For each $i \in \{1, \ldots, N\}$ perform the steps 3–5:
3. $P_l \leftarrow$ ProjectAndSampleSquare($\boldsymbol{p}_i$, $s$)  { Algorithm 2, left camera }
4. $P_r \leftarrow$ ProjectAndSampleSquare($\boldsymbol{p}_i$, $s$)  { Algorithm 2, right camera }
5. $w_i := w_s(I_{s,l}, P_l) + w_s(I_{s,r}, P_r)$  { using Eq. (1.13) }
6. $w_{min} := \min\{w_i\}$
7. $w_{max} := \max\{w_i\}$
8. For each $i \in \{1, \ldots, N\}$ calculate the final likelihood by:
   $\pi_i := \exp\left\{-s_s \frac{w_i - w_{min}}{w_{max} - w_{min}})\right\}$
```

2

Integrating Vision Toolkit (IVT)

Author: Pedram Azad

2.1 Implementation

The *Integrating Vision Toolkit* (IVT) [Azad, 2008a] is an image processing library developed at the chair of Professor Dillmann. It is freely available under the GNU license in source code, and can be downloaded from SourceForge.net.[1] Detailed instructions for the installation of the IVT can be found in Chapter 3. A theoretically well-founded but yet practical and easy-to-understand introduction to computer vision including various example applications using the IVT is given in [Azad et al., 2008] (resp. in German in [Azad et al., 2007a]).

The highest goal with the development of the IVT was to lay a clean, object-oriented architecture as foundation and at the same time to offer efficient implementations of the algorithms. A core component, by which the IVT stands out from most image processing libraries – and also from the popular OpenCV – is the abstraction of image acquisition by an appropriate camera interface. This enables the development of image processing solutions that are perfectly independent of the used image source from a software point of view. Thus changing only one line of code for the choice of the camera module is sufficient, in order to run an application with another camera.

Throughout the implementation of the IVT, the focus was on avoiding dependencies on libraries and between files of the IVT whenever possible. The strict separation of the files which contain only proprietary developments, and those which encapsulate calls to external libraries, allows to configure the IVT in

[1] `http://sourceforge.net`.

a convenient manner. The core of the IVT is written in pure ANSI C/C++ and can be compiled without any library. Alternatively, it is possible to include classes or namespaces which encapsulate the libraries Qt, OpenCV and OpenGL. Due to the strict separation, no mutual dependencies exist, so that these libraries can be added independently.

2.2 Architecture

In this section a compact summary of the architecture of the IVT is given. For this, the interfaces of the image sources, graphical user interfaces, OpenCV and OpenGL are explained. At the end of this chapter, short sample programs illustrate the use of the individual interfaces.

2.2.1 The Class CByteImage

The class `CByteImage` forms the core of the IVT, and is able to represent an 8 bit grayscale image and a 24 bit color image. It is written in pure ANSI C++ and is thus platform independent. In addition to the pure representation of an image, this class can read and write bitmap files[2].

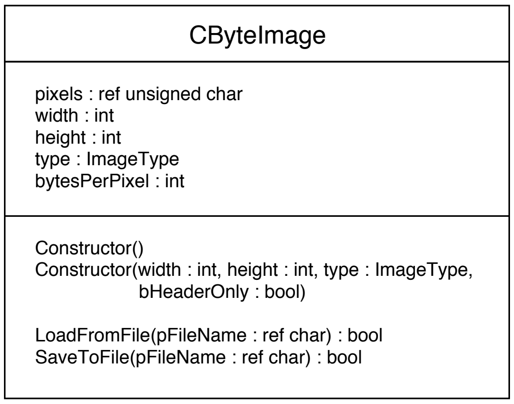

CByteImage
pixels : ref unsigned char
width : int
height : int
type : ImageType
bytesPerPixel : int
Constructor()
Constructor(width : int, height : int, type : ImageType, bHeaderOnly : bool)
LoadFromFile(pFileName : ref char) : bool
SaveToFile(pFileName : ref char) : bool

Fig. 2.1. Representation of the public attributes and methods of the class CByteImage in UML.

The public attributes of this class are the integer variables `width` and `height`, which describe the width and height of the image in pixels, the pointer `pixel`

[2] Image files with the file ending .bmp.

of type `unsigned char*`, which points to the beginning of the storage area of the image, and the variable `type`, which contains the value `eGrayScale` for grayscale images, and the value `eRGB24` for 24 bit color images. Images of type `eRGB24` can likewise contain a 24 bit HSV color image, since these are identical in terms of the representation in memory. As an additional attribute, the variable `bytesPerPixel` is offered, which contains the value 1 for grayscale images and the value 3 for color images.

2.2.2 Connection of Graphical User Interfaces

The elements of graphical user interfaces (GUIs) for image processing applications can be divided into two groups: Input elements such as input fields, slide controls, buttons etc., and the display of images. The latter functionality is encapsulated in the IVT by the interface `CGUIInterface`. The implementation of the input elements is left to the application and can be made directly with the library.

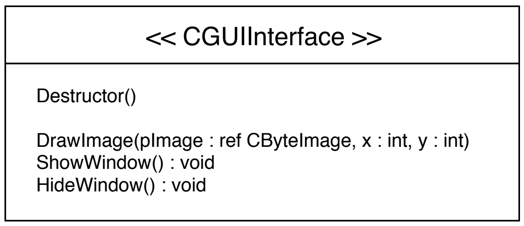


Fig. 2.2. Representation of the methods of the interface CGUIInterface in UML.

A class inheriting the `CGUIInterface` must implement its three virtual methods and the virtual destructor. The method `DrawImage` possesses the parameter `pImage` of type `CByteImage*` as the image to be drawn, and the two optional integer parameters `x` and `y`, with which it is possible to indicate an offset to the top left-hand corner of the window for the beginning of the image. The methods `Show` and `Hide` are responsible for the visibility/invisibility of the window.

Additionally, an initialization must be carried out at the beginning of the program for most libraries, and control given briefly in each run of the main loop, in order to give the library the possibility of handling input and output. The encapsulation of these calls is made by the interface `CApplicationHandlerInterface`, which consists of the two virtual methods `Reset` and `ProcessEventsAndGetExit`. The method `Reset` must be called first, before creating and displaying windows. The method `ProcessEventsAndGetExit` should be called at the end of each cycle run; the return value `true` signals that the user wants to terminate the application.

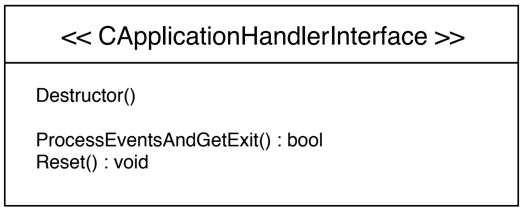

&lt;&lt; CApplicationHandlerInterface >>
Destructor()
ProcessEventsAndGetExit() : bool
Reset() : void

Fig. 2.3. Representation of the methods of the interface CApplicationHandlerInterface in UML.

In the IVT, the classes `COpenCVWindow`/`COpenCVApplicationHandler` and `CQTWindow`/`CQTApplicationHandler` are available as ready-to-use implementations of this interface. Throughout the example applications, the Qt implementation is used, since with it, and by using Qt, it is easy to add graphical input elements. The implementation uses the library Qt3, which is freely available for purely private or scientific purposes. Detailed instructions for the installation of Qt3 can be found in Chapter 3. An example of the use of this interface with Qt is given in Section 2.3.2.

2.2.3 Connection of Image Sources

In the IVT, for each image source, a module is implemented, which supplies images of the type `CByteImage`, using the interface `CVideoCaptureInterface`. This interface is defined in such a manner that it can transfer any number of images with one call, which is necessary with multi-camera systems. Below, the designation *camera module* is used synonymously for the module of an image source.

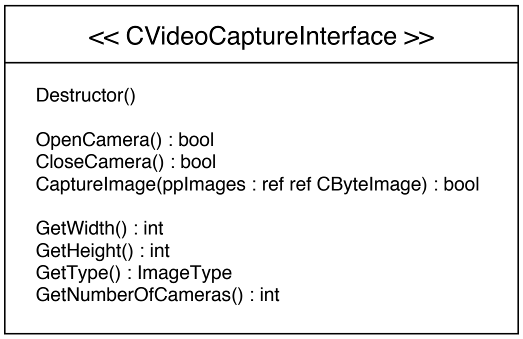

&lt;&lt; CVideoCaptureInterface >>
Destructor()
OpenCamera() : bool
CloseCamera() : bool
CaptureImage(ppImages : ref ref CByteImage) : bool
GetWidth() : int
GetHeight() : int
GetType() : ImageType
GetNumberOfCameras() : int

Fig. 2.4. Representation of the methods of the interface CVideoCaptureInterface in UML.

A camera module must implement the seven virtual methods of the interface and the virtual destructor. The method `OpenCamera` accomplishes the necessary initializations, starts the image recording and returns a boolean value which indicates the result of the initialization. The parameters necessary for the configuration of the module, for example the choice of the resolution or the encoding, differ from image source to image source, and are therefore to be selected through the constructor of the respective module. The method `CloseCamera` terminates the image recording, deletes the objects and frees memory space. The method `CaptureImage` possesses the parameter `ppImages` as input, which is of the type `CByteImage**`. Thus it is possible to transfer as many images as desired with one call. The instances of `CByteImage` must already be allocated and consistent in size, type and number with the requirements of the camera module. For this purpose, this information can be retrieved using the methods `GetWidth`, `GetHeight`, `GetType` and `GetNumberOfCameras`. In order to receive valid values, the method `OpenCamera` must have already been successfully called. An example of the use of a camera module is given in Section 2.3.3.

2.2.4 Integration of OpenCV

The OpenCV[3] is a very popular image processing library, which is also available on SourceForge.net as an open source project. It offers a broad spectrum of image processing algorithms, but, however, does not have an overall object-oriented architecture.

Due to its broad spectrum of offered functionality, the OpenCV is merged optionally into the IVT. For this, functions of the OpenCV were encapsulated and can be used via calls to the IVT, which usually operate on instances of the class `CByteImage`. In this way, functions in the OpenCV are used perfectly transparently, i.e. the called methods are independent of the OpenCV. Since the images do not have to be converted themselves, but only an image header of a few bytes is created, the calls to the OpenCV are made with virtually no additional computational effort. Files which encapsulate functions of the OpenCV, carry the letters `CV` as ending, as for example `ImageProcessorCV.h` and `ImageProcessorCV.cpp`. An example of the use of the OpenCV within the IVT is given in Section 2.3.4.

2.2.5 Integration of OpenGL via Qt

OpenGL[4] is a specification of a platform-independent programming interface for the development of 3D computer graphics. An implementation of this

[3] `http://sourceforge.net/projects/opencvlibrary`.
[4] Open Graphics Library.

interface runs on all common operating systems, thus also under Windows, Mac OS and Linux.

For some 3D image processing applications it can be useful to also visualize the result in 3D. For this purpose, the IVT encapsulates the 3D-primitives sphere and cylinder with the class `COpenGLVisualizer`, offering the methods `DrawSphere` and `DrawCylinder`. Before these methods can be used, the method `Init` must be called, which expects the width and the height of the graphic area as input parameters. Alternatively the method `InitByCalibration` initializes the OpenGL camera model by a given camera calibration stored in an object of the type `CCalibration`. This way, OpenGL can emulate a camera that has been previously calibrated, which is crucial for augmented reality applications. If contents of the graphic area are to be deleted, then the method `Clear` must be called. Below is a description of how the graphic area can be visualized in a window. As an alternative, it is also possible to write the graphic area directly into an RGB 24 color image of the type `CByteImage`. For this, an image of the appropriate size – as indicated before in the method `Init` (resp. `InitByCalibration`) – must be created and passed as argument to the method `GetImage`. Please note that OpenGL writes a vertically flipped image to memory, which can be flipped back by using the function `ImageProcessorCV::FlipY`.

Using the class `CQTGLWindow`, it is possible to visualize the OpenGL graphic area in a window. In the constructor, the size of the window must be indicated. This should correspond to the parameters indicated in the method `Init` (resp. `InitByCalibration`) from `COpenGLVisualizer`, since otherwise either the area to be visualized is cut off or only part of the window is filled out. In order to update the contents of the window, the method `Update` must be called. An example of the use of OpenGL within the IVT is given in Section 2.3.5.

2.3 Example Applications

In this section, examples of the described interfaces and related libraries are given. Each example consists of a short description and an implementation in C++, which can be found at the end of this chapter. The source code can be downloaded from the download section on the web page of the book [Azad et al., 2008].

2.3.1 Use of Basic Functionality

This application is to serve as the simplest example of the use of the IVT. An image is opened, converted using the function `ConvertImage` from

`ImageProcessor` into a grayscale image, and afterwards a gradient filter is applied with `CalculateGradientImageSobel`, also from `ImageProcessor`. The result is written into a bitmap file.

No further libraries are necessary for compiling and running this application. The name of this application is `simpleapp`.

2.3.2 Use of a Graphical User Interface

This application shows the use of the interface `CGUIInterface`. First an image is opened. If successful, a Qt window is created and displayed afterwards, in order to then draw the loaded image in a loop. For the compilation and running of this application, the library Qt3 is necessary. Detailed instructions for the installation of Qt3 are given in Chapter 3. The name of this application is `guiapp`.

2.3.3 Use of a Camera Module

In this application the use of the interface for image sources, as described in Section 2.2.3, is shown. In order to allow readers who do not possess a camera to compile and run this application as well, the class `CBitmapCapture` is used as a camera module. This module is able to read an image from an indicated path, and to simulate a camera with a static image. First this module is loaded and initialized. If successful, an image of appropriate size and type is created. Subsequently, in each run of the main loop, the image is captured by the camera module and then displayed in a window.

If another camera module is to be used, for example in order to connect a real camera, then only the line of code for the creation of the instance of the camera module must be modified at the beginning of the `main` routine.

2.3.4 Use of OpenCV

This application shows the use of the OpenCV within the IVT via the offered encapsulations. In this example, as an alternative to loading an image using the method `LoadFromFile` from the class `CByteImage`, the function with the same name from `ImageAccessCV` is used. This encapsulates a call to the function `cvLoadImage` from the OpenCV, which is also able to load different image formats like JPG, PPM, TIFF and PNG. In order to be able to use this function, the library `highgui` from the OpenCV is necessary.

At the beginning of the application, an image is loaded and in the case of success converted into a grayscale image. Subsequently, the result is converted using the function `Resize` from `ImageProcessorCV` into an image half the width and height. A gradient filter is applied to the result, and the final result is stored as a bitmap file, as already shown in the application from Section 2.3.1. As can be seen, encapsulated calls to the OpenCV and pure IVT calls can be mixed with one another at will.

2.3.5 Use of the OpenGL Interface

In this application, the rotation of a simple 3D object, consisting of two spheres and a cylinder, is animated. For this, firstly Qt is initialized and a window of the type `CQTGLWindow` is created. Subsequently, an instance of the OpenGL encapsulation module is created and initialized. Using a 2D rotation, the end points `point1` and `point2` are rotated in the plane $y = 0$ in each iteration of the main loop by the current angle `angle`.

In order to achieve the same frame rate on different computers, 30 Hz is specified using a timer. For this, the function `get_timer_value` from `IVT/src/Helpers/helpers.h` returns an integer value of the type `unsigned int`, which holds a relative time value in microseconds. If the parameter `true` is passed as argument to the function `get_timer_value`, then the timer is reset to zero.

2.4 Overview of further IVT Functionality

In the previous sections, fundamental classes and interfaces of the IVT were introduced, for the representation and the visualization of images, as well as for the connection of useful libraries. In this section, a short overview of the most important classes and routines provided by the IVT is given.

The directories are to be understood in each case to be a subdirectory of `IVT/src`. If it concerns classes, then these carry the additional letter "C" compared as the first letter of the file name. With namespaces the names are identical.

Camera Calibration (directory `Calibration`)

`CCalibration`:	Camera image functions for an individual camera
`CRectification(CV)`:	Performing a rectification
`CStereoCalibration`:	Calculations for a stereo camera system
`StereoCalibrationCV`:	Computation of the rectification parameters for a given instance of `CStereoCalibration`
`CUndistortionCV`:	Performing an undistortion

Operations on Images and Representations (directory `Image`)

`CByteImage`:	Data structure for the representation of 8 bit grayscale images and RGB 24 color images
`CShortImage`:	Data structure for the representation of 16 bit grayscale images
`CIntImage`:	Data structure for the representation of 32 bit grayscale images
`ImageAccessCV`:	Loading and saving of images with the OpenCV
`ImageProcessor(CV)`:	Point operators, filters, morphological operators, conversion routines etc.
`IplImageAdaptor`:	Conversion between `CByteImage` (IVT) and `IplImage` (OpenCV)
`PrimitivesDrawer(CV)`:	Draws 2D primitives such as circles, ellipses, rectangles, lines etc.
`CStereoVision(SVS)`:	Calculation of depth maps
`CStereoMatcher`:	Stereo triangulation on the basis of stereo correspondences computed with subpixel-accuracy using a ZNCC.

Mathematic Routines (directory `Math`)

`CFloatMatrix`:	Data structure for the representation of a matrix of values of the data type `float` (compatible with `CByteImage` and `CShortImage`)
`CDoubleMatrix`:	Data structure for the representation of a matrix of values of the data type `double` (compatible with `CFloatMatrix`)
`CMatd`:	Data structure and operations for convenient calculating with matrices of arbitrary dimension
`CVecd`:	Data structure and operations for convenient calculating with vectors of arbitrary dimension
`Math2d`:	Data structure and operations for efficiently calculating with vectors and matrices in 2D
`Math3d`:	Data structure and operations for efficiently calculating with vectors and matrices in 3D
`LinearAlgebra(CV)`:	Functions for PCA, SVD, and standard vector/matrix operations, operating on `CFloatMatrix` and/or `CDoubleMatrix`

Others

Directory `Color`:	Classes for color segmentation
Directory `DataStructures`:	Dynamic array and kd-tree for efficient nearest neighbor search
Directory `gui`:	Implementation of `CGUIInterface` for Qt (and additionally highgui from OpenCV) for the visualization of images and 3D rendering
Directory `Helper`:	Useful helper functions such as sleep and time measurements with microsecond resolution, etc.
Directory `ObjectFinder`:	Color blob tracking, encapsulation of the Viola/Jones Haar classifier implemented by the OpenCV
Directory `ParticleFilter`:	General particle filtering framework
Directory `Threading`:	Abstraction of threads, implementing POSIX and Windows threads, and synchronization objects
Directory `Tracking`:	Pose estimation on the basis of point correspondences (2D-3D, 3D-3D), RAPiD tracker [Harris and Stennett, 1990]
Directory `VideoCapture`:	Various camera modules implementing `CVideoCaptureInterface`
Directory `Visualizer`:	Classes for 3D visualization using OpenGL or Open Inventor. `COpenGLVisualizer` can simulate a real camera given an instance of `CCalibration`.

Jan 16, 08 1:50 **Examples** Page 1/5

```
// *****************************************************************************
// Filename:  simpleapp.cpp
// Author:    Pedram Azad
// Date:      2007/02/09
// *****************************************************************************

#include "Image/ImageProcessor.h"
#include "Image/ByteImage.h"

#include <stdio.h>



int main()
{
   CByteImage image;

   // Load image
   if (!image.LoadFromFile("../../files/dish_scene_left.bmp"))
   {
      printf("Error: Could not open image.\n");
      return 1;
   }

   // Convert color image to grayscale image
   CByteImage gray_image(image.width, image.height, CByteImage::eGrayScale);
   ImageProcessor::ConvertImage(&image, &gray_image);

   // Calculate gradient image
   ImageProcessor::CalculateGradientImageSobel(&gray_image, &gray_image);

   // Save image to file
   gray_image.SaveToFile("output.bmp");

   printf("Result image has been saved to 'output.bmp'.\n");

   return 0;
}
```

Jan 16, 08 1:50 **Examples** Page 2/5

```
// *****************************************************************************
// Filename:  guiapp.cpp
// Author:    Pedram Azad
// Date:      2007/02/09
// *****************************************************************************

#include "Image/ByteImage.h"
#include "gui/QTApplicationHandler.h"
#include "gui/QTWindow.h"

#include <stdio.h>



int main(int argc, char **args)
{
   CByteImage image;

   // Load image
   if (!image.LoadFromFile("../../files/dish_scene_left.bmp"))
   {
      printf("Error: Could not open image.\n");
      return 1;
   }

   // Initialize Qt
   CQTApplicationHandler qtApplicationHandler(argc, args);
   qtApplicationHandler.Reset();

   // Create and show window
   CQTWindow window(image.width, image.height);
   window.Show();

   // Main loop
   while (!qtApplicationHandler.ProcessEventsAndGetExit())
   {
      window.DrawImage(&image);
   }

   return 0;
}
```

Jan 16, 08 1:50 **Examples** Page 3/5

```
// ****************************************************************************
// Filename:  cameraapp.cpp
// Author:    Pedram Azad
// Date:      2007/02/09
// ****************************************************************************

#include "Image/ByteImage.h"
#include "gui/QTApplicationHandler.h"
#include "gui/QTWindow.h"
#include "VideoCapture/BitmapCapture.h"

#include <stdio.h>



int main(int argc, char **args)
{
   // Create camera module
   CBitmapCapture capture("../../files/dish_scene_left.bmp");

   // Open camera
   if (!capture.OpenCamera())
   {
      printf("Error: Could not open camera.\n");
      return 1;
   }

   const int width = capture.GetWidth();
   const int height = capture.GetHeight();
   const CByteImage::ImageType type = capture.GetType();

   CByteImage *ppImages[] = { new CByteImage(width, height, type) };

   // Initialize Qt
   CQTApplicationHandler qtApplicationHandler(argc, args);
   qtApplicationHandler.Reset();

   // Create and show window
   CQTWindow window(width, height);
   window.Show();

   // Main loop
   while (!qtApplicationHandler.ProcessEventsAndGetExit())
   {
      if (!capture.CaptureImage(ppImages))
         break;

      window.DrawImage(ppImages[0]);
   }

   delete ppImages[0];

   return 0;
}
```

Jan 16, 08 1:50 **Examples** Page 4/5

```
// ****************************************************************************
// Filename:  opencvapp.cpp
// Author:    Pedram Azad
// Date:      2007/02/09
// ****************************************************************************

#include "Image/ImageProcessor.h"
#include "Image/ImageProcessorCV.h"
#include "Image/ImageAccessCV.h"
#include "Image/ByteImage.h"

#include <stdio.h>



int main()
{
   CByteImage image;

   // Load image
   if (!ImageAccessCV::LoadFromFile(&image, "../../files/dish_scene_left.bmp"))
   {
      printf("Error: Could not open image.\n");
      return 1;
   }

   // Convert color image to grayscale image
   CByteImage gray_image(image.width, image.height, CByteImage::eGrayScale);
   ImageProcessor::ConvertImage(&image, &gray_image);

   // Create image with half width and half height
   CByteImage small_image(image.width/2,image.height/2, CByteImage::eGrayScale);

   // Shrink image
   ImageProcessorCV::Resize(&gray_image, &small_image);

   // Calculate gradient image
   ImageProcessor::CalculateGradientImageSobel(&small_image, &small_image);

   // Save result image to file
   small_image.SaveToFile("output.bmp");

   printf("Result image has been saved to 'output.bmp'.\n");

   return 0;
}
```

Jan 16, 08 1:50 **Examples** Page 5/5

```
// ***************************************************************************
// Filename:  openglapp.cpp
// Author:    Pedram Azad
// Date:      2007/02/09
// ***************************************************************************

#include "gui/QTGLWindow.h"
#include "gui/QTApplicationHandler.h"
#include "Visualizer/OpenGLVisualizer.h"
#include "Math/Math3d.h"
#include "Math/Constants.h"
#include "Helpers/helpers.h"

#include <math.h>



int main(int argc, char **args)
{
   // Initialize Qt
   CQTApplicationHandler qtApplicationHandler(argc, args);
   qtApplicationHandler.Reset();

   // Create OpenGL window and OpenGL visualizer
   CQTGLWindow window(640, 480);
   COpenGLVisualizer visualizer;

   // Initialize OpenGL
   visualizer.Init(640, 480);

   const double r = 250;
   double angle = 0;

   // Main loop
   while (!qtApplicationHandler.ProcessEventsAndGetExit())
   {
      unsigned int t = get_timer_value(true);

      const Vec3d p1 = { r * cos(angle), 0, r * sin(angle) + 1500 };
      const Vec3d p2 = { r * cos(angle + PI), 0, r * sin(angle + PI) + 1500 };

      // Clear graphics area
      visualizer.Clear();

      // Draw spheres and cylinder
      visualizer.DrawSphere(p1, 100, COpenGLVisualizer::red);
      visualizer.DrawSphere(p2, 100, COpenGLVisualizer::red);
      visualizer.DrawCylinder(p1, p2, 50, 50, COpenGLVisualizer::blue);

      // Update window
      window.Update();

      // Increase angle
      angle += 0.05;

      // Timing
      const double T = 1000000.0 / 30; // 30 fps (frames per second)
      while (get_timer_value() < t + T);
   }

   return 0;
}
```

3

Installation of IVT, OpenCV and Qt under Windows and Linux

Author: Lars Pätzold

This tutorial describes the setup of a ready-to-use environment for programming with the IVT library [Azad, 2008a] under Windows and Linux. This includes the setup of the libraries supported by the IVT, namely OpenCV and Qt, as well as a driver for Firewire cameras.

In many parts, the given instructions are very detailed, in order to make the introduction to the use of the IVT easier. A lot of steps are actually necessary to set up the development environment, until one can finally start programming. Despite the additional effort, the installation of all listed libraries is recommended. If the use of a camera is not intended, the steps in which "1394" occurs can be skipped.

The tutorial is divided into two main sections: *Windows* and *Linux*. These two sections are to be regarded as independent from each other. The section for Windows ends with a summary, which can more or less serve as a quick guide for the installation on Windows systems. For the first time, however, the detailed step by step version is recommended, since it contains important information regarding the installation.

It is to be noted that the software introduced here may only be used for private and/or scientific purposes in accordance with its license, and not for commercial use. For more exact information concerning the license regulations, refer to the indicated internet sites.

3.1 Windows

3.1.1 OpenCV

1. Downloading OpenCV

On the internet at `http://sourceforge.net/projects/opencvlibrary/` download the file `OpenCV_1.0.exe` from the download area. This file is found as part of the package *opencv-win* and by following the continuative release link *1.0*. There, the download `OpenCV_1.0.exe` is found. This version is recommended, since the compatibility of the IVT with this version has been ensured by numerous tests.

2. Installation

With the downloaded file, the OpenCV can be installed. The target directory can be set during the setup (e.g. `C:\Program Files\OpenCV`).

3. Settings in the development environment

In the development environment of choice (e.g. Microsoft Visual C++), the directory paths to the include and library files must be set.

For the include files, the following paths are to be set:

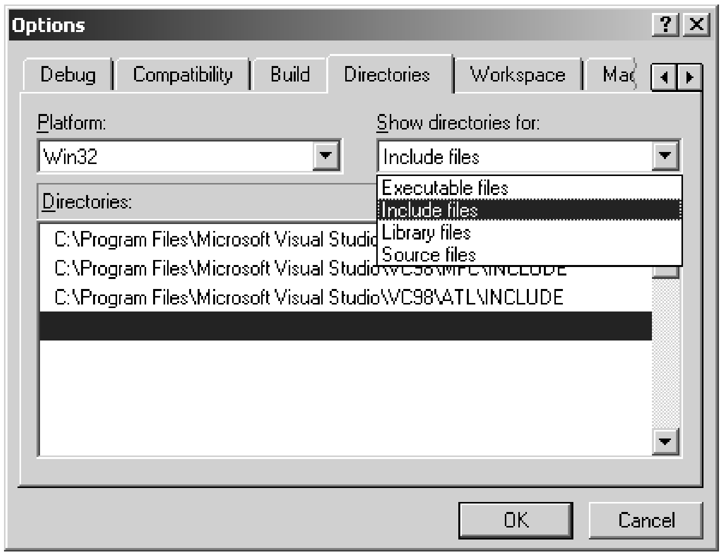
```
C:\Program Files\OpenCV\cv\include
C:\Program Files\OpenCV\cxcore\include
C:\Program Files\OpenCV\otherlibs\highgui
C:\Program Files\OpenCV\cvaux\include
C:\Program Files\OpenCV\otherlibs\cvcam\include
```

The path for the library files reads:

```
C:\Program Files\OpenCV\lib
```

If in the previous step, the default target directory `C:\Program Files\OpenCV` was not selected, then the directory paths have to be adapted accordingly.

In Microsoft Visual C++ 6.0, the directory paths can be modified in the general options as follows: In the menu *Tools*, under *Options...*, the tab *Directories* can be found. In order to set the directories for the include files, the drop-down menu *Include files* must be selected, and for the directory of the library files accordingly the drop-down menu *Library files* (see Fig. 3.1). The directory paths can then be added in the list below.



Platform: Win32

Show directories for: Include files

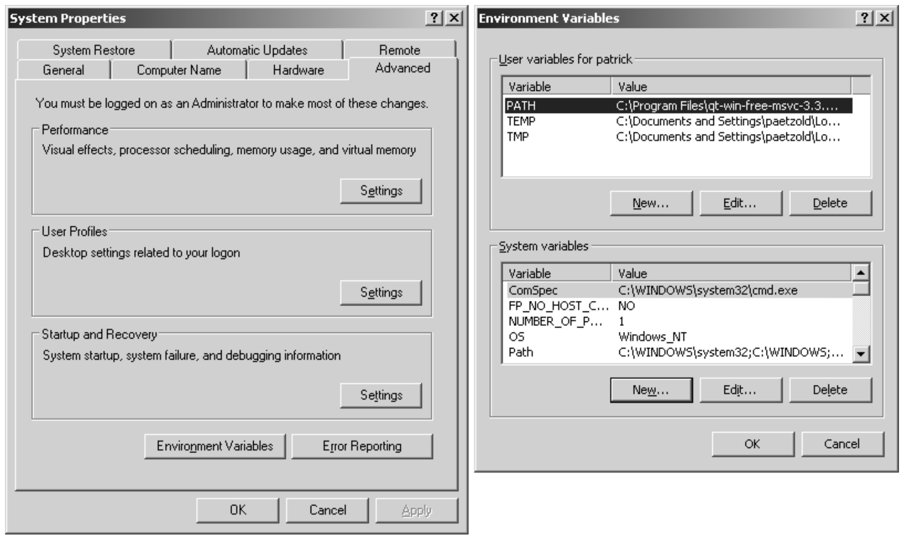
Directories:

C:\Program Files\Microsoft Visual Studio
C:\Program Files\Microsoft Visual Studio\VC98\MFC\INCLUDE
C:\Program Files\Microsoft Visual Studio\VC98\ATL\INCLUDE

Fig. 3.1. Dialog window for the options.

4. Checking the system variable PATH

Via *Start*, *Settings*, *Control Panel*, *System* the dialog window *System Properties* is opened (alternatively use the hot-key Windows key + Break key). There, the appropriate dialog window can be opened via the button *Environment Variables* on the tab *Advanced* (see Fig. 3.2). The PATH variable should contain `C:\Program Files\OpenCV\bin` (the path must be set according to the target directory during setup). If this is not yet the case, the PATH variable must be changed accordingly. It is important to note that multiple paths in the PATH variable must be separated by a semicolon.



You must be logged on as an Administrator to make most of these changes.

Performance
Visual effects, processor scheduling, memory usage, and virtual memory

User Profiles
Desktop settings related to your logon

Startup and Recovery
System startup, system failure, and debugging information

User variables for patrick

Variable	Value
PATH	C:\Program Files\qt-win-free-msvc-3.3....
TEMP	C:\Documents and Settings\paetzold\Lo...
TMP	C:\Documents and Settings\paetzold\Lo...

System variables

Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
FP_NO_HOST_C...	NO
NUMBER_OF_P...	1
OS	Windows_NT
Path	C:\WINDOWS\system32;C:\WINDOWS;...

Fig. 3.2. Dialog window for the system properties (left) and the environment variables (right).

3.1.2 Qt

1. Downloading Qt

At this point, a freely available implementation of Qt3 is recommended, which is easy to download and install. It should be noted that this implementation is independent from the official Qt version by Trolltech. For more exact information, refer directly to the following internet address. The free Qt3 version can be found at `http://sourceforge.net/projects/qtwin/`. In the download area under the package *Unofficial Qtwin*, the continuative link *View older releases of the Unofficial Qtwin package* points to the download of the version qt-win-3.3.4-3. For Microsoft Visual C++ 6.0, the file `setup-qt-win-free-msvc-3.3.4-3.exe` is to be downloaded and for Microsoft Visual Studio .NET 2003, the file `setup-qt-win-free-msvc.net2003-3.3.4-3.exe`.

2. Installation

With the downloaded file, a setup procedure can be started, which allows to select the target directory.

3. Settings in the development environment

In the development environment, the following directory paths must be set:

Path for the include files:
`C:\Program Files\qt-win-free-msvc-3.3.4\include`

Path for the library files:
`C:\Program Files\qt-win-free-msvc-3.3.4\lib`

A guide for changing the paths in Microsoft Visual C++ 6.0 can be found in Step 3 of the previous section dealing with the installation of the OpenCV.

The paths have to be changed accordingly, if the default target directory was not selected.

4. Checking the system variable PATH

The PATH variable should contain:
`C:\Program Files\qt-win-free-msvc-3.3.4\bin` (must be changed according to the provided target directory during the setup, if necessary). How the system variable can be checked and changed is described in Step 4 of the OpenCV installation. Here, the location of the dynamic link library `qt-mt3.dll` is of interest.

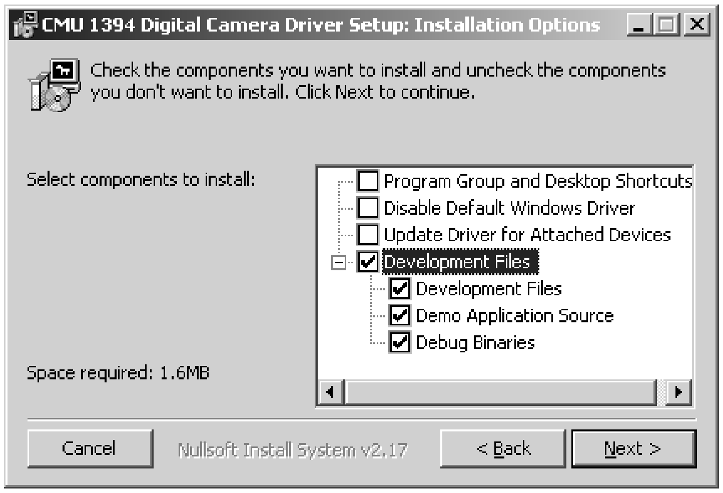
3.1.3 CMU1394

1. Download

The setup version of the CMU1394 driver is found in the download area at `http://www.cs.cmu.edu/~iwan/1394/`. We recommend the version 6.3, since compatibility of this version with the IVT has been ensured by various tests.

2. Installation

Executing the downloaded file starts a setup procedure. During the setup, the components to be installed, as well as the target directory of the installation can be selected. Here, make particularly sure that the component *Development Files* is installed (see Fig. 3.3). The default directory for the installation is `C:\Program Files\CMU\1394Camera`.


CMU 1394 Digital Camera Driver Setup: Installation Options

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- [ ] Program Group and Desktop Shortcuts
- [ ] Disable Default Windows Driver
- [ ] Update Driver for Attached Devices
- [x] Development Files
  - [x] Development Files
  - [x] Demo Application Source
  - [x] Debug Binaries

Space required: 1.6MB

Fig. 3.3. Setup of the CMU1394 driver.

3. Settings in the development environment

The directory paths for the development environment are as follows:

Path for the include files:
`C:\Program Files\CMU\1394Camera\include`

Path for the library files:
`C:\Program Files\CMU\1394Camera\lib`

A guide for setting the paths in Microsoft Visual C++ 6.0 is shown in Step 3 of the section dealing with the installation of the OpenCV.

Again the paths have to be set accordingly, if the default target directory was not selected.

3.1.4 IVT

1. Downloading the IVT

On the internet at `http://sourceforge.net/projects/IVT/` download the zip file (e.g. ivt-1.1.3.zip) from the download area.

2. Unpacking

The downloaded zip file can be unpacked into any directory. It is recommended that the file is unpacked to the same location as OpenCV and Qt. According to the standard configuration of the OpenCV and Qt installations, this location is the directory `C:\Program Files`. After extraction, the subdirectories `doc`, `examples`, `files`, `lib`, `src` and `win32` can be found in `C:\Program Files\IVT`.

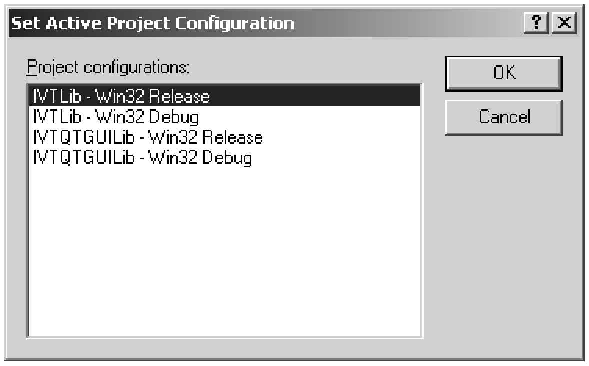
3. Settings in the development environment

The directory paths for the include and library files must be set in the development environment. The include files are located in the subdirectory `src` of the IVT directory (i.e. `C:\Program Files\IVT\src`). The library files are located in the subdirectory `lib\win32` of the IVT directory (e.g. `C:\Program Files\IVT\lib\win32`).

A guide to modifying the paths in Microsoft Visual C++ 6.0 is shown in Step 3 of the section dealing with the installation of the OpenCV. Again the paths have to be set accordingly, if the default target directory was not selected.

4. Building the libraries

The workspace file `IVT.dsw` for Microsoft Visual C++ 6.0 is located in the directory `win32\IVTLib`. If this file is opened in a newer version of Microsoft Visual C++, then an automatic conversion into a new file format takes place and the workspace can be used likewise. Once opened, the version of the library that is to be built can be selected in the menu *Build* using the menu option *Set Active Configuration...* (see Fig. 3.4).


Set Active Project Configuration

Project configurations:

IVTLib - Win32 Release
IVTLib - Win32 Debug
IVTQTGUILib - Win32 Release
IVTQTGUILib - Win32 Debug

Fig. 3.4. Dialog window for the configuration of the active project.

The following versions are available:

IVTLib – Win32 Release
This is the standard version of the IVT library. Debug information is not included. The file name of the library file is `ivt.lib`.

IVTLib – Win32 Debug
The debug version permits step-by-step execution of the functions within the IVT library during debugging. Usually applications that are compiled with the debug version are noticeably slower throughout execution. The library to be generated has the file name `ivtd.lib`.

IVTQTGUILib – Win32 Release
With this selection, a further library is built. This is merely an extension to the standard version of the IVT library, which contains simplified support for graphical user interface with Qt. The file name of the library is `ivtguiqt.lib`.

IVTQTGUILib – Win32 Debug
The debug version of the extension library again permits the debugging within the IVT source code. The file name of the debug version is `ivtguiqtd.lib`.

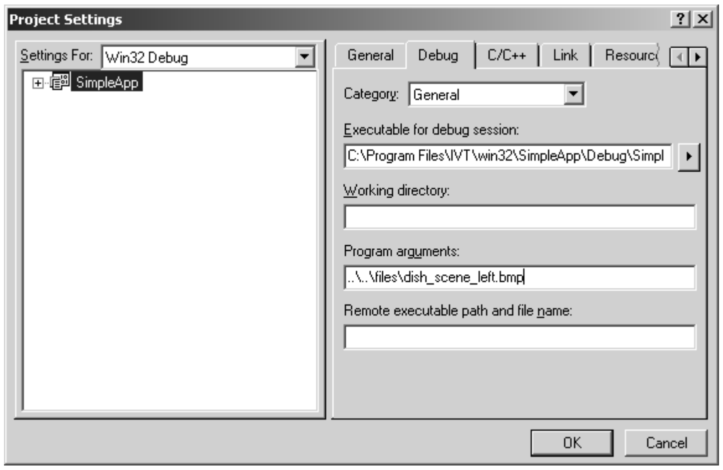
The respective library file can be built afterwards via the menu *Build* and the appropriate menu option (i.e. *Build ivt.lib*). After the build process is finished, the generated library file (e.g. `ivt.lib`) is located in the subdirectory `IVT\lib\win32`. In order to be able to later faultlessly compile all applications with the IVT, all four library files of the IVT should be built.

5. Example application

For checking the installation, and as an introduction to programming with the IVT, a suitable example application is SimpleApp. This is the simplest of the numerous example applications contained in the IVT.

The workspace file `SimpleApp.dsw` is located in the IVT subdirectory `win32\SimpleApp`. After having opened it in Microsoft Visual C++, the application should be compiled without any errors via the menu item *Build SimpleApp.exe* from the menu *Build*. If a problem occurs, first it should verified which version has been selected in the menu *Build* under the menu item *Set Active Configuration...*, and whether the according IVT library file was built in the previous step. The example applications are configured for the debug version of the IVT library, i.e. for ivtd.lib and if necessary for ivtguiqtd.lib.

Before the application is executed, an image file should be set as program argument. The program arguments can be added in *Project settings* (see Fig. 3.5). To get there, go to menu *Project* and choose the menu item *Settings...*. On the tab *Debug*, `..\..\files\dish_scene_left.bmp` can be set as program argument, for example. This path points to an image file, which is contained in the subdirectory `files` of the IVT.



Settings For: Win32 Debug
SimpleApp

Category: General
Executable for debug session: C:\Program Files\IVT\win32\SimpleApp\Debug\Simpl
Working directory:
Program arguments: ..\..\files\dish_scene_left.bmp
Remote executable path and file name:

Fig. 3.5. Dialog window for the project settings.

If the application is now executed from the menu *Build* and the menu item *Execute SimpleApp.exe*, the result *output written to file 'output.bmp'* is printed in the console window. Subsequently, the file `output.bmp` is located in the same directory as the workspace file `SimpleApp.dsw` i.e. in `win32\SimpleApp` in the IVT directory.

In order to verify whether OpenCV and Qt are correctly installed as well, the example applications SimpleAppCV and ShowImageQT can be run. Both applications can be handled in the same way as the application SimpleApp.

If up to here all steps have been accomplished, and the example applications could be compiled and executed, then a ready-to-use environment for programming with the IVT has been set up.

3.1.5 Summary

If after following all installation steps, any problems occur during compilation, the following overview serves for verification of all important directory paths. The overview summarizes all directory paths that should be set in the development environment (i.e. Microsoft Visual C++) and in the system variable PATH. Additionally, the internet addresses mentioned in the installation steps are listed.

Internet addresses:

```
http://sourceforge.net/projects/opencvlibrary/
http://sourceforge.net/projects/qtwin/
http://www.cs.cmu.edu/~iwan/1394/
http://sourceforge.net/projects/IVT/
```

Paths for the include files:

```
C:\Program Files\OpenCV\cv\include
C:\Program Files\OpenCV\cxcore\include
C:\Program Files\OpenCV\otherlibs\highgui
C:\Program Files\OpenCV\cvaux\include
C:\Program Files\OpenCV\otherlibs\cvcam\include
C:\Program Files\qt-win-free-msvc-3.3.4\include
C:\Program Files\CMU\1394Camera\include
C:\Program Files\IVT\src
```

Paths for the library files:

```
C:\Program Files\OpenCV\lib
C:\Program Files\qt-win-free-msvc-3.3.4\lib
C:\Program Files\CMU\1394Camera\lib
C:\Program Files\IVT\lib\win32
```

The system variable PATH should contain the following paths:

```
C:\Program Files\OpenCV\bin;
C:\Program Files\qt-win-free-msvc-3.3.4\bin;
```

3.2 Linux

3.2.1 OpenCV

1. Downloading OpenCV

At `http://sourceforge.net/projects/opencvlibrary` in the download area under the package opencv-linux, follow the link *1.0*. There the file `opencv-1.0.0.tar.gz` can be downloaded.

2. Unpacking

In a console window, change to the directory of the downloaded archive and then run the command `tar xfvz opencv-1.0.0.tar.gz`.

3. Compile the OpenCV libraries

After changing to the unpacked directory `opencv-1.0.0` run the following instructions consecutively:

```
./configure
make
make install
ldconfig
```

Note: For the instructions `make install` and `ldconfig`, root privileges are necessary.

Subsequently, the include files of the OpenCV should be located in the directory `/usr/local/include/opencv`.

3.2.2 Qt

For the installation, a so-called package tool is recommended. It is important to make sure that a developer package of Qt version 3 is installed. This is done under Debian Linux, for example, with: `apt-get install qt3-dev-tools`.

The directory `/usr/include/qt3` should exist afterwards, containing the include files of Qt.

3.2.3 Firewire and libdc1394/libraw1394

1. Installation from libdc1394/libraw1394

The two libraries libdc1394 and libraw1394 can also be installed with a package tool; under Debian Linux, for example with `apt-get install libdc1394` as well as `apt-get install libraw1394`.

2. Installing a firewire camera (optional)

In order to use the camera under Linux, the four kernel modules named raw1394, video1394, ohci1394 and ieee1394 must be loaded. This is accomplished under Debian Linux using the command `modprobe`. `modprobe raw1394` and `modprobe video1394` are sufficient to activate all four modules. With the command `lsmod | grep 1394`, it can be checked whether the four modules have been successfully loaded. This command should list the above mentioned four modules. If this should not be the case, try to add each module individually with the command `modprobe`.

In order to allow a user or an application to access the interface to the camera, the user has to be registered in the appropriate groups (usually `video`) of the devices `/dev/raw1394` and `/dev/video1394`, or the user privileges of the devices must be set appropriately. For this, root privileges are necessary.

3.2.4 IVT

1. Downloading the IVT

Download `ivt-1.1.3.tar.gz` at `http://sourceforge.net/projects/IVT/` in the download area.

2. Unpacking

In a console window, change to the directory of the downloaded archive and then run the command `tar xfvz ivt-1.1.3.tar.gz`. After unpacking, the directory `IVT` should exist in the current directory.

3. Configuration

The IVT offers the possibility of configuring the library before building it. This configuration takes place in the file `IVT/src/Makefile.base`. `Makefile.base` must be opened and edited with a text editor (e.g. vim). In the following, the most important configuration options are listed. These options are configuration variables, which can be set either to 1 or 0. In order to integrate or exclude certain parts of the IVT library, the appropriate configuration variable must be set to 1 for integration and to 0 for exclusion.

USE_QT = 1
The IVT library is extended by classes that allow the easy creation of graphical user interfaces with Qt. The associated source files with the file names `QT*` are located in `IVT/src/gui`.

USE_OPENCV = 1
Part of the IVT-library accesses the OpenCV library. The file names of the source files belonging to this part have the endings `CV.h` and `CV.cpp`, respectively.

USE_OPENGL = 1
OpenGL and GLU are used by the class `COpenGLVisualizer`, which allows the visualization of spheres and cylinders. The source files of this class are located in `IVT/src/Visualizer`.

USE_HIGHGUI = 1
HighGUI is part of the OpenCV library. As an alternative to Qt, windows and images can be graphically displayed with it. The respective source files `OpenCV*`, which use HighGUI, are located in `IVT/src/gui`.

USE_LIBDC1394 = 1
An interface to the library libdc1394 enables the control of IEEE1394 cameras (firewire cameras). The source files of this moduel are called `Linux1394Capture.*` and are located in `IVT/src/VideoCapture`.

Apart from these variables, the directory paths to include and library files as well as the file names of the libraries can be changed in the lower part of the file. The pre-configured paths usually correspond to the default installation paths. However, these paths can differ, depending on the Linux distribution used. Modifying the paths is only recommended if other than the default paths were used throughout installation, or if problems arise.

For the adjustment, a short explanation of the appropriate variables is given in the following:

INCPATHS_BASE
Contains all directory paths to included files. Adding a path takes place by using the operator += and adding the path with the leading parameter -I. For example INCPATHS_BASE += -I/usr/include/qt3

LIBPATHS_BASE
Contains all directory paths to library files. Adding a path takes place by using the operator += and adding the path with the leading parameter -L. For example LIBPATHS_BASE += -L/usr/lib/qt3/lib

LIBS_BASE
Contains all file names of the libraries. Adding a file name takes place by using the operator += and adding the file name with the leading parameter -l. For example LIBS_BASE += -lqt-mt -livtgui

4. Building the IVT libraries

The libraries are built by running the command `make` in the directory `IVT/src`. If after having built the IVT library once, modifications are made to the IVT files located in `IVT/src`, it is recommended to run the command `make clean` before running the command `make`, in order to enforce a complete new build process.

5. Example application

For checking the installation, and as an introduction to programming with the IVT, a suitable example application is SimpleApp. It is found in `IVT/examples/SimpleApp` in the IVT directory. After changing to the directory `SimpleApp`, an executable file with the name `simpleapp` is created by running the command `make`. If the application is started with `./simpleapp ../../files/dish_scene_left.bmp`, then, provided all libraries have been correctly installed, a file called `output.bmp` is produced in the same directory.

Now, the setup, and therefore the tutorial is finished. There are numerous further example applications in the directory `IVT/examples`, with which the features of the IVT can be tested and learned.

A

Mathematics

Author: Pedram Azad

In this section, useful mathematic relationships and formulas, mainly in the context of linear least squares are given. Since the definitions operating on complex numbers are not of interest, only real-valued matrices and vectors will be assumed. A more detailed introductions including derivations of the formulas can be found in [Wikipedia, 2008].

A.1 Singular Value Decomposition

The singular value decomposition (SVD) of a matrix $A \in \mathbb{R}^{m,n}$ is defined as the product

$$A = U\,\Sigma\,V^T \tag{A.1}$$

with $U \in \mathbb{R}^{m,m}$, $\Sigma \in \mathbb{R}^{m,n}$, $V \in \mathbb{R}^{n,n}$. The matrices U and V are orthogonal matrices. The matrix Σ contains non-zero values – the singular values – on the diagonal and zeroes off the diagonal. An algorithm for computing the singular value decomposition of a matrix A is described in [Press et al., 2007]. In the IVT (see Chapters 2 and 3), the implementation of the OpenCV [OpenCV, 2008] is encapsulated by the function `LinearAlgebraCV::SVD`.

A.2 Pseudoinverse

The pseudoinverse $A^+ \in \mathbb{R}^{n,m}$ of a matrix $A \in \mathbb{R}^{m,n}$ is the generalization of the inverse matrix. The mathematic definition of the pseudoinverse, which is also often called the Moore-Penrose pseudoinverse, can be found in [Wikipedia, 2008]. There are several ways for computing the pseudoinverse, from which two commonly used methods are presented in the following.

A.2.1 Using the Regular Inverse

If the matrix A has full rank, then the pseudoinverse can be computed by using the regular inverse. If $m > n$, then it applies:

$$A^{+} = (A^{T} A)^{-1} A^{T} \tag{A.2}$$

otherwise for $m < n$:

$$A^{+} = A^{T} (A\, A^{T})^{-1} \tag{A.3}$$

As can be easily shown, for the special case $m = n$, the regular inverse A^{-1} is computed.

A.2.2 Using the Singular Value Decomposition

A numerically more stable method for computing the pseudoinverse, which also succeeds when the matrix A does not have full rank, is based on the singular value decomposition. Given the singular value decomposition $A = U\, \Sigma\, V^{T}$ (see Section A.1), the pseudoinverse is calculated by:

$$A^{+} = V\, \Sigma^{+}\, U^{T} \tag{A.4}$$

where the matrix $\Sigma^{+} \in \mathbb{R}^{m,n}$ is derived from the matrix Σ by inverting all non-zero values on the diagonal, and leaving all zeroes in place. In practice, the condition that a value is not zero is verified by comparing the absolute value to a predefined epsilon.

A.3 Linear Least Squares

Given the over-determined system of linear equations

$$A\, \boldsymbol{x} = \boldsymbol{b} \tag{A.5}$$

with $A \in \mathbb{R}^{m,n}$, $\boldsymbol{b} \in \mathbb{R}^{m}$, and $m > n$, the task is to find an optimal solution $\boldsymbol{x}^{*} \in \mathbb{R}^{n}$, so that the sum of squared differences $\|A\, \boldsymbol{x}^{*} - \boldsymbol{b}\|_2^2$ becomes minimal. In the following, the three commonly used approaches for solving this problem are presented.

A.3.1 Using the Normal Equation

The normal equation is acquired by left-sided multiplication of A^{T}:

$$A^T A\,\boldsymbol{x} = A^T \boldsymbol{b} \tag{A.6}$$

If the matrix A has full rank, i.e. its rank is n, this system of linear equations can be solved by using the regular inverse of $A^T A$, which equals the computation of the pseudoinverse with the method presented in Section A.2.1. The optimal solution $\boldsymbol{x}^*$ is thus computed by:

$$\boldsymbol{x}^* = (A^T A)^{-1} A^T \boldsymbol{b} \tag{A.7}$$

A.3.2 Using the QR Decomposition

A numerically more stable but also computationally more expensive method is based on the QR decomposition of the matrix A:

$$A = Q\,R \tag{A.8}$$

where $Q \in \mathbb{R}^{m,m}$ is a orthogonal matrix, and $R \in \mathbb{R}^{m,n}$ is an upper triangular matrix. The matrix $R_n \in \mathbb{R}^{n,n}$ is defined as the upper square part of the matrix R, i.e. it is:

$$\begin{pmatrix} R_n \\ O \end{pmatrix} \tag{A.9}$$

where O is a $(m-n) \times n$-matrix containing zeroes only. The optimal solution $\boldsymbol{x}^*$ can then be computed by solving the following system of linear equations:

$$R_n\,\boldsymbol{x}^* = (Q^T \boldsymbol{b})_n \tag{A.10}$$

where $(Q^T \boldsymbol{b})_n$ denotes the upper n values of the vector of $Q^T \boldsymbol{b}$. This system of linear equations can be efficiently solved by utilizing the fact that R_n is an upper triangular matrix. An algorithm for computing the QR decomposition of a matrix A is described in [Press et al., 2007].

A.3.3 Using the Singular Value Decomposition

The numerically most stable but also computationally most expensive method is based on the singular value decomposition. For this purpose, the pseudoinverse A^+ has to be computed by using the method presented in Section A.2.2, yielding the optimal solution $\boldsymbol{x}^* = A^+ \boldsymbol{x}$.

A.3.4 Homogeneous Systems

In the case of a homogeneous system of linear equations, i.e. $\boldsymbol{b} = \boldsymbol{0}$, all previously presented methods fail, since multiplication with $\boldsymbol{b}$ results in the zero vector. In this case, the singular value decomposition $A = U\,\Sigma\,V^T$ can be used to directly compute the optimal solution $\boldsymbol{x}^*$, which is given by the last column of the matrix V.

A.4 Functions for Rotations

In this section, some useful functions for calculating rotation matrices and rotation angles based on a given rotation axis are presented. Given a rotation axis $\boldsymbol{a}$ and a rotation angle α, the rotation matrix R performing this rotation can be computed by Algorithm 4.

Algorithm 4 RotationMatrixAxisAngle($\boldsymbol{a}$, α) $\rightarrow R$

```
1. $(x,\, y,\, z) := \dfrac{\boldsymbol{a}}{|\boldsymbol{a}|}$
2. $s := \sin\alpha$
3. $c := \cos\alpha$
4. $t := 1 - c$
5. $R := \begin{pmatrix} tx^2 + c & txy - sz & txz + sy \\ txy + sz & ty^2 + c & tyz - sx \\ txz - sy & tyz + sx & tz^2 + c \end{pmatrix}$
```

The reverse direction, i.e. extracting the axis $\boldsymbol{a}$ and the rotation angle α for a given rotation matrix R, is computed by Algorithm 5. Note that $-\boldsymbol{a}$ and $-\alpha$ result in the same rotation; apart from this, the solution is unique.

Algorithm 5 ExtractAxisAngle(R) $\rightarrow \boldsymbol{a}, \alpha$

```
1. $\begin{pmatrix} r_1 & r_2 & r_3 \\ r_4 & r_5 & r_6 \\ r_7 & r_8 & r_9 \end{pmatrix} := R$
2. $x := r_8 - r_6$
3. $y := r_3 - r_7$
4. $z := r_4 - r_2$
5. $r := \sqrt{x^2 + y^2 + z^2}$
6. $t := r_1 + r_5 + r_9$
7. $\alpha := \operatorname{atan2}(r,\, t - 1)$
8. $\boldsymbol{a} := (x,\, y,\, z)^T$
```

Finally, a function is presented that calculates the rotation angle α that is necessary for rotating a given vector $\boldsymbol{x}_1$ to another vector $\boldsymbol{x}_2$, with $|\boldsymbol{x}_1| = |\boldsymbol{x}_2|$, around a given rotation axis $\boldsymbol{a}$. To compute the rotation angle α, the vectors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ are parallel projected onto the rotation plane defined by the rotation axis $\boldsymbol{a}$. The sign of the rotation angle is determined by verifying the resulting two alternatives.

Algorithm 6 Angle($\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{a}$) $\rightarrow \alpha$

```
1. $\boldsymbol{n} := \dfrac{\boldsymbol{a}}{|\boldsymbol{a}|}$
2. $\boldsymbol{u}_1 := \boldsymbol{x}_1 - (\boldsymbol{n}\,\boldsymbol{u}_1)\,\boldsymbol{n}$
3. $\boldsymbol{u}_1 := \dfrac{\boldsymbol{u}_1}{|\boldsymbol{u}_1|}$
4. $\boldsymbol{u}_2 := \boldsymbol{x}_2 - (\boldsymbol{n}\,\boldsymbol{u}_2)\,\boldsymbol{n}$
5. $\boldsymbol{u}_2 := \dfrac{\boldsymbol{u}_2}{|\boldsymbol{u}_2|}$
6. $\alpha := \dfrac{\boldsymbol{u}_1\,\boldsymbol{u}_2}{|\boldsymbol{u}_1|\,|\boldsymbol{u}_2|}$
7. $R \leftarrow$ RotationMatrixAxisAngle($\boldsymbol{n}$, $\alpha$)
8. $d_1 := |R\,\boldsymbol{u_1} - \boldsymbol{u}_2|$
9. $d_2 := |R^T\boldsymbol{u_1} - \boldsymbol{u}_2|$
10. If $d_2 < d_1$ then set $\alpha := -\alpha$.
```

References

[Azad, 2008a] Azad, P. (2008a). Integrating Vision Toolkit. `http://ivt.sourceforge.net`.

[Azad, 2008b] Azad, P. (2008b). *submitted thesis: Visual Perception for Manipulation and Imitation in Humanoid Robots*. PhD thesis, University of Karlsruhe, Karlsruhe Germany.

[Azad et al., 2006a] Azad, P., Asfour, T., and Dillmann, R. (2006a). Combining Apperance-based and Model-based Methods for Real-Time Object Recognition and 6D Localization. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 5339–5344, Beijing, China.

[Azad et al., 2007a] Azad, P., Gockel, T., and Dillmann, R. (2007a). *Computer Vision – Das Praxisbuch*. Elektor, Germany.

[Azad et al., 2008] Azad, P., Gockel, T., and Dillmann, R. (2008). *Computer Vision – Principles and Practice*. Elektor International Media BV, Netherlands. `http://wwwiaim.ira.uka.de/computer-vision/index-en.html`.

[Azad et al., 2006b] Azad, P., Ude, A., Asfour, T., Cheng, G., and Dillmann, R. (2006b). Image-based Markerless 3D Human Motion Capture using Multiple Cues. In *International Workshop on Vision Based Human-Robot Interaction*, Palermo, Italy.

[Azad et al., 2007b] Azad, P., Ude, A., Asfour, T., and Dillmann, R. (2007b). Stereo-based Markerless Human Motion Capture for Humanoid Robot Systems. In *International Conference on Robotics and Automation (ICRA)*, pages 3951–3956, Roma, Italy.

[Azad et al., 2004] Azad, P., Ude, A., Dillmann, R., and Cheng, G. (2004). A Full Body Human Motion Capture System using Particle Filtering and On-The-Fly Edge Detection. In *International Conference on Humanoid Robots (Humanoids)*, pages 941–959, Santa Monica, USA.

[Bando et al., 2004] Bando, T., Shibata, T., Doya, K., and Ishii, S. (2004). Switching Particle Filter for Efficient Real-Time Visual Tracking. In *International Conference on Pattern Recognition (ICPR)*, pages 720–723, Cambridge, UK.

[Blake and Isard, 1998] Blake, A. and Isard, M. (1998). *Active Contours*. Springer.

[Casella and Robert, 1996] Casella, G. and Robert, C. P. (1996). Rao-Blackwellisation of Sampling Schemes. *Biometrika*, 83(1):81–94.

[Deutscher et al., 2000] Deutscher, J., Blake, A., and Reid, I. (2000). Articulated Body Motion Capture by Annealed Particle Filtering. In *Int. Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 2126–2133, Hilton Head, USA.

[Deutscher et al., 2001] Deutscher, J., Davison, A., and Reid, I. (2001). Automatic Partitioning of High Dimensional Search Spaces associated with Articulated Body Motion Capture. In *Int. Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 669–676, Kauai, USA.

[Doucet et al., 2000] Doucet, A., de Freitas, N., Murphy, K. P., and Russell, S. J. (2000). Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In *Conf. on Uncertainty in Artificial Intelligence (UAI)*, pages 176–183, Stanford, USA.

[Harris and Stennett, 1990] Harris, C. G. and Stennett, C. (1990). 3D object tracking at video rate – RAPiD. In *British Machine Vision Conference (BMVC)*, pages 73–78, Oxford, UK.

[Isard and Blake, 1996] Isard, M. and Blake, A. (1996). Contour Tracking by Stochastic Propagation of Conditional Density. In *European Conference Computer Vision (ECCV)*, pages 343–356, Cambridge, UK.

[Isard and Blake, 1998] Isard, M. and Blake, A. (1998). Condensation – Conditional Density Propagation for Visual Tracking. *International Journal of Computer Vision*, 29(1):5–28.

[Klein and Murray, 2006] Klein, G. and Murray, D. (2006). Full-3D Edge Tracking with a Particle Filter. In *British Machine Vision Conference (BMVC)*, volume 3, pages 1119–1128, Edinburgh, UK.

[MacCormick, 2000] MacCormick, J. (2000). *Probabilistic models and stochastic algorithms for visual tracking.* PhD thesis, University of Oxford, UK.

[MacCormick and Isard, 2000] MacCormick, J. and Isard, M. (2000). Partitioned sampling, articulated objects, and interface-quality hand tracking. In *European Conference Computer Vision (ECCV)*, pages 3–19, Dublin, Ireland.

[OpenCV, 2008] OpenCV (2008). `http://sourceforge.net/projects/opencvlibrary`.

[Pitt and Shepard, 1999] Pitt, M. K. and Shepard, N. (1999). Filtering via Simulation: Auxiliary Particle Filters. *Journal of the American Statistical Association*, 94(446):590–599.

[Press et al., 2007] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes.* Cambridge University Press, 3rd edition.

[Wikipedia, 2008] Wikipedia (2008). `http://en.wikipedia.org`.