

ViSP 2.6.1: Visual Servoing Platform

3D and vision related transformations

Lagadic project
<http://www.irisa.fr/lagadic>

September 1, 2011

Manikandan Bakthavatchalam
François Chaumette
Eric Marchand
Antony Saunier
Fabien Spindler
Romain Tallonneau

Contents

1	3D transformations	4
1.1	Coordinate system	4
1.2	Rotation representation	4
1.2.1	Rotation matrices	4
1.2.2	$\theta\mathbf{u}$ representation	5
1.2.3	Euler angles	6
1.2.4	Quaternions	8
1.3	Translation	8
1.4	Pose	9
1.4.1	Pose vector	9
1.4.2	Homogeneous transformation matrix	10
1.5	Twist transformation matrices	12
1.5.1	Velocity twist matrix	12
1.5.2	Force-torque twist matrix	13
2	Vision related transformations	14
2.1	Camera models	14
2.1.1	Perspective camera	14
2.1.2	Catadioptric sensor	18
2.2	Homography	19
2.2.1	Planar structure	19
2.2.2	Non planar structure	20

1 3D transformations

1.1 Coordinate system

ViSP uses a right-handed coordinate system for 3D data. When considering the camera frame, the \mathbf{z} axis is coming into the screen. All angles are in radians and all distances are in meters. All transformations considered here preserve the distances and the angles between the reference frames.

1.2 Rotation representation

Let us first note that the standard representation for rotation in ViSP is the rotation matrix. Nevertheless other representations exist such as three versions of the Euler angles and the $\theta\mathbf{u}$ representation (angle and axe of the rotation). To date the quaternion representation is not considered in ViSP.

1.2.1 Rotation matrices

Let ${}^a\mathbf{R}_b$ be a rotation matrix from frame \mathcal{F}_a to \mathcal{F}_b .

- \mathbf{R} is a 3×3 orthonormal matrix. Hence $\mathbf{R}^T \mathbf{R} = \mathbf{R} \mathbf{R}^T = \mathbf{I}$. This implies:

1. $\mathbf{R}^{-1} = \mathbf{R}^T$
2. $\det(\mathbf{R}) = 1$ (since the chosen coordinate frame is right handed)
3. the lines $\mathbf{R}_{i\bullet}$ and columns $\mathbf{R}_{\bullet j}$ define an orthogonal basis.

The space of rotation matrices, $SO(3) \subset \mathbb{R}$, is called the special orthogonal group. Under the operation of matrix multiplication, it satisfies the axioms of closure, identity, inverse and associativity. It is defined by:

$$SO(3) = \{\mathbf{R} \in \mathbb{R}^{3 \times 3} : \mathbf{R} \mathbf{R}^T = \mathbf{I}, \det(\mathbf{R}) = 1\} \quad (1)$$

Rotation matrix can be defined using the `vpRotationMatrix` class. It inherits from `vpMatrix` but most of the methods have been overloaded for efficiency issues or to ensure the specific properties of the rotation matrix (for example the $+$ or $-$ operators are no longer defined).

Construction. *The default value for a rotation matrix is identity. The rotation can be built either from another rotation matrix or from any other representation (see the next paragraphs):*

```
1 vpRotationMatrix R(Ri) ; // where Ri is a vpRotationMatrix
2 vpRotationMatrix R(rzyx) ; // where rzyx is a vpRzyxVector
3 vpRotationMatrix R(rzyz) ; // where rzyz is a vpRzyzVector
4 vpRotationMatrix R(rxyz) ; // where rxyz is a vpRxyzVector
5 vpRotationMatrix R(theta) ; // where theta is a vpThetaUVector
```

It can also be built directly from three floats ($\theta\mathbf{u}_x, \theta\mathbf{u}_y, \theta\mathbf{u}_z$) which correspond to the $\theta\mathbf{u}$ representation (see section 1.2.2):

```
1 vpRotationMatrix R(0.2,0.3,0.5) ; // (thetaux,thetay,thetauz)
```

Alternatively, the method `buildFrom(...)` can be used:

```

1 vpRotationMatrix R ; R.buildFrom(Ri) ; // where Ri is a vpRotationMatrix
2 vpRotationMatrix R ; R.buildFrom(rzyx) ; // where rzyx is a vpRzyxVector
3 vpRotationMatrix R ; R.buildFrom(rzyz) ; // where rzyz is a vpRzyzVector
4 vpRotationMatrix R ; R.buildFrom(rxyz) ; // where rxyz is a vpRxyzVvector
5 vpRotationMatrix R ; R.buildFrom(thetau) ; // where thetau is a vpThetaUVector
6
7 vpRotationMatrix R ; R.buildFrom(0.2,0.3,0.5) ; // (thetaux,thetauy,thetauz)

```

Operations. For example if aRb and bRc define the rotation between frame \mathcal{F}_a and \mathcal{F}_b (respectively \mathcal{F}_b and \mathcal{F}_c), computing transformation between \mathcal{F}_a and \mathcal{F}_c is implemented by:

```

1 vpRotationMatrix aRb(...);
2 vpRotationMatrix bRc(...);
3
4 vpRotationMatrix aRc;
5
6 aRc = aRb*bRc;

```

In a similar way the inverse transformation can be obtained using

```

1 vpRotationMatrix cRa;
2
3 cRa = (aRb*bRc).inverse();
4 cRa = (aRb*bRc).t(); // (since R^-1 = R^T)
5 cRa = aRc.t();

```

Since a representation of a rotation as a rotation matrix is not minimal it is interesting to have other formulations of a rotation.

1.2.2 $\theta\mathbf{u}$ representation

$\theta\mathbf{u}$ where $\mathbf{u} = (u_x, u_y, u_z)^\top$ is a unit vector representing the rotation axis and θ is the rotation angle. $\theta\mathbf{u}$ is one of the minimal representation.

It is possible to build the rotation matrix \mathbf{R} from the $\theta\mathbf{u}$ representation using the Rodrigues formula:

$$\mathbf{R} = \mathbf{I}_3 + (1 - \cos \theta) \mathbf{u}\mathbf{u}^\top + \sin \theta [\mathbf{u}]_\times \quad (2)$$

with \mathbf{I}_3 the identity matrix of dimension 3×3 and $[\mathbf{u}]_\times$ the skew matrix :

$$[\mathbf{u}]_\times = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \quad (3)$$

On the other hand $\theta\mathbf{u}$ is obtained from \mathbf{R} using:

$$\cos \theta = \frac{1}{2} (\text{trace}(\mathbf{R}) - 1) \quad (4)$$

and

$$\sin \theta [\mathbf{u}]_\times = \frac{1}{2} (\mathbf{R} - \mathbf{R}^T) \quad (5)$$

From 4 we get θ :

$$\theta = \arccos \left(\frac{\mathbf{R}_{11} + \mathbf{R}_{22} + \mathbf{R}_{33} - 1}{2} \right) \quad (6)$$

Fixing $\theta > 0$, if $\sin \theta \neq 0$, \mathbf{u} can be determined from 5 :

$$\mathbf{u} = \frac{1}{2 \sin(\theta)} \begin{bmatrix} \mathbf{R}_{32} - \mathbf{R}_{23} \\ \mathbf{R}_{13} - \mathbf{R}_{31} \\ \mathbf{R}_{21} - \mathbf{R}_{12} \end{bmatrix} \quad (7)$$

The $\theta\mathbf{u}$ representation is implemented in the `vpThetaUVector` class. It is nothing but an array of 3 float values. As previously stated, the main rotation representation in ViSP is the rotation matrix. The only operators defined for the `vpThetaUVector` are the construction operators:

Construction. Default value for the `vpThetaUVector` is a null vector. The vector can be build either from another `vpThetaUVector` or from any other representation of a rotation:

```
1 vpThetaUVector thetau(thetaui) ; // where thetaui is a vpThetaUVector
2 vpThetaUVector thetau(R) ; // where R is a vpRotationMatrix
3 vpThetaUVector thetau(rzyx) ; // where rzyx is a vpRzyxVector
4 vpThetaUVector thetau(rzyz) ; // where rzyz is a vpRzyzVector
5 vpThetaUVector thetau(rxyz) ; // where rxyz is a vpRxyzVector
```

Alternatively, the method `buildFrom(...)` can be used:

```
1 vpThetaUVector thetau ; thetau.buildFrom(R) ; // where R is a vpRotationMatrix
2 vpThetaUVector thetau ; thetau.buildFrom(rzyx) ; // where rzyx is a vpRzyxVector
3 vpThetaUVector thetau ; thetau.buildFrom(rzyz) ; // where rzyz is a vpRzyzVector
4 vpThetaUVector thetau ; thetau.buildFrom(rxyz) ; // where rxyz is a vpRxyzVector
```

It can also be built directly from three floats ($\theta\mathbf{u}_x, \theta\mathbf{u}_y, \theta\mathbf{u}_z$) which correspond to the $\theta\mathbf{u}$ representation:

```
1 vpThetaUVector thetau(0.2,0.3,0.5) ; // thetaux, thetauy, thetauz
```

Building a rotation matrix R from a $\theta\mathbf{u}$ vector is done through the constructor or the `buildFrom(...)` method of the `vpRotationMatrix` class.

```
1 vpRotationMatrix R(thetaui) ; // where thetau is a vpThetaUVector
2 vpRotationMatrix R ; R.buildFrom(thetaui) ; // where thetau is a vpThetaUVector
```

Access. To access directly to the $\theta\mathbf{u}$ angles, the `[]` operator is defined.

```
1 vpThetaUVector thetau;
2 thetau[0] = 0.2 ; // thetaux;
3 thetau[1] = 0.3 ; // thetauy;
4 thetau[2] = 0.5 ; // thetauz;
```

1.2.3 Euler angles

In the Euler representation, the rotation is contained into 3 rotations around predefined axes (the rotation is then described by 3 angles (φ, θ, ψ)). This is also a minimal representation. ViSP implements three versions of the Euler angles.

Considering each elementary rotation matrix:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix} \mathbf{R}_y(\beta) = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix} \mathbf{R}_z(\gamma) = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (8)$$

The overall rotation is the product of these elementary matrices:

$$\mathbf{R}_{zyx}(\varphi, \theta, \psi) = \mathbf{R}_z(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_x(\psi) \quad (9)$$

This representation is not unique. In ViSP, the Euler representations of \mathbf{R}_{xyz} and \mathbf{R}_{zyz} are also implemented. In these cases:

$$\mathbf{R}_{xyz}(\varphi, \theta, \psi) = \mathbf{R}_x(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_z(\psi) \quad (10)$$

and

$$\mathbf{R}_{zyz}(\varphi, \theta, \psi) = \mathbf{R}_z(\varphi)\mathbf{R}_y(\theta)\mathbf{R}_z(\psi) \quad (11)$$

The Euler angles representations are implemented in the `vpRzyxVector` class for the \mathbf{R}_{zyx} representation, the `vpRxyzVector` class for the \mathbf{R}_{xyz} representation, and the `vpRzyzVector` class for the \mathbf{R}_{zyz} representation.

The only operators defined for the `vpRzyxVector`, `vpRxyzVector` and `vpRzyzVector` are the construction operators. Each class is by default a null vector and can be initialised from any representation of a rotation (either directly in the constructor or by using the `buildFrom(...)` method).

vpRzyxVector

```
1 vpRzyxVector rzyx(r) ; // where r is a vpRzyxVector
2 vpRzyxVector rzyx(R) ; // where R is a vpRotationMatrix
3 vpRzyxVector rzyx(theta) ; // where theta is a vpThetaUVector
```

```
1 vpRzyxVector rzyx ; rzyx.buildFrom(R) ; // where R is a vpRotationMatrix
2 vpRzyxVector rzyx ; rzyx.buildFrom(theta) ; // where theta is a vpThetaUVector
```

It can also be built directly from three floats which correspond to the Euler angles around z, y and x:

```
1 vpRzyxVector rzyx(0.2,0.3,0.5) ; // Euler angles around z, y and x
```

vpRzyzVector

```
1 vpRzyzVector rzyz(r) ; // where r is a vpRzyzVector
2 vpRzyzVector rzyz(R) ; // where R is a vpRotationMatrix
3 vpRzyzVector rzyz(theta) ; // where theta is a vpThetaUVector
```

```
1 vpRzyzVector rzyz ; rzyz.buildFrom(R) ; // where R is a vpRotationMatrix
2 vpRzyzVector rzyz ; rzyz.buildFrom(theta) ; // where theta is a vpThetaUVector
```

It can also be built directly from three floats which correspond to the Euler angles around z, y and z:

```
1 vpRzyzVector rzyz(0.2,0.3,0.5) ; // Euler angles around z, y and z
```

vpRxyzVector

```
1 vpRxyzVector rxyz(r) ; // where r is a vpRxyzVector
2 vpRxyzVector rxyz(R) ; // where R is a vpRotationMatrix
3 vpRxyzVector rxyz(theta) ; // where theta is a vpThetaUVector
```

```
1 vpRxyzVector rxyz ; rxyz.buildFrom(R) ; // where R is a vpRotationMatrix
2 vpRxyzVector rxyz ; rxyz.buildFrom(theta) ; // where theta is a vpThetaUVector
```

It can also be built directly from three floats which correspond to the Euler angles around x, y and z:

```
1 vpRxyzVector rxyz(0.2,0.3,0.5) ; // Euler angles around x, y and z
```

Rotation matrix from an Euler angles vector. The initialisation is done through the constructor or the `buildFrom(...)` method of the `vpRotationMatrix` class.

```

1 vpRotationMatrix R(rzyx) ; // where rzyx is a vpRzyxVector
2 vpRotationMatrix R(rxyz) ; // where rxyz is a vpRxyzVector
3 vpRotationMatrix R(rzyz) ; // where rzyz is a vpRzyzVector
4 // or
5 vpRotationMatrix R ; R.buildFrom(rzyx) ; // where rzyx is a vpRzyxVector
6 vpRotationMatrix R ; R.buildFrom(rxyz) ; // where rxyz is a vpRxyzVector
7 vpRotationMatrix R ; R.buildFrom(rzyz) ; // where rzyz is a vpRzyzVector

```

Access. Read/write access of the angles is done using the `[]` operator:

```

1 vpRzyxVector rzyx;
2 rzyx[0] = 0.2 ; rzyx[1] = 0.3 ; rzyx[2] = 0.5 ; // Euler angles around z, y and x;
3
4 vpRzyzVector rzyz;
5 rzyz[0] = 0.2 ; rzyz[1] = 0.3 ; rzyz[2] = 0.5 ; // Euler angles around z, y and z;
6
7 vpRxyzVector rxyz;
8 rxyz[0] = 0.2 ; rxyz[1] = 0.3 ; rxyz[2] = 0.5 ; // Euler angles around x, y and z;

```

1.2.4 Quaternions

To date, quaternions are not implemented in ViSP.

1.3 Translation

Let ${}^a t_b = (t_x, t_y, t_z)$ be a translation from frame \mathcal{F}_a to \mathcal{F}_b . In ViSP, a translation is represented by a column vector of dimension 3.

Translation can be defined using the `vpTranslationVector` class. It inherits from the `vpColVector` but most of the members have been overloaded for efficiency issue.

Construction. Default value for the translation vector is a null vector. The translation can be built from another translation vector:

```

1 vpTranslationVector t(ti) ; // where ti is a vpTranslationVector

```

*It can also be built directly from three floats which correspond to the 3 components of the translation along the **x,y** and **z** axes :*

```

1 vpTranslationVector t(0.2,0.3,0.5) ; // tx,ty and tz

```

Operations. Let atb and btc define the translation between frame \mathcal{F}_a and \mathcal{F}_b (respectively \mathcal{F}_b and \mathcal{F}_c), the transformation between \mathcal{F}_a and \mathcal{F}_c is implemented by:

```

1 vpTranslationVector atb(...);
2 vpTranslationVector btc(...);
3
4 vpTranslationVector atc;
5 atc = atb + btc;

```

In a similar way the inverse transformation can be obtained using:

```
1 vpTranslationVector cta ;
2
3 cta = -(atb + btc) ;
```

To obtain the skew symmetric matrix or the cross product of two translation vectors, the following code can be use:

```
1 vpTranslationVector t1(...) ;
2 vpTranslationVector t2(...) ;
3
4 vpMatrix S(3,3) ; // initialize a 3x3 matrix
5 S = t1.skew() ; // skew symmetric matrix
6
7 vpTranslationVector cross ;
8 cross = vpTranslationVector::cross(t1,t2) ; // cross product t1xt2
```

1.4 Pose

1.4.1 Pose vector

A pose is a complete representation of a rigid motion in the euclidean space. It is composed of a rotation and a translation and is minimally represented by 6 parameters:

$${}^a\mathbf{r}_b = ({}^a\mathbf{t}_b, \theta\mathbf{u}) \in \mathbb{R}^6, \quad (12)$$

where ${}^a\mathbf{r}_b$ is the pose from \mathcal{F}_a to \mathcal{F}_b , with ${}^a\mathbf{t}_b$ being the translation vector between \mathcal{F}_a and \mathcal{F}_b along the $\mathbf{x}, \mathbf{y}, \mathbf{z}$ axis and $\theta\mathbf{u}$, the $\theta\mathbf{u}$ representation of the rotation ${}^a\mathbf{R}_b$ between these frames.

The pose vector is implemented in the `vpPoseVector` class. It is nothing but the concatenation of a `vpTranslationVector` and a `vpThetaUVector`. The only operators defined for the `vpPoseVector` are the construction operators:

Construction. *Default value for the pose vector is a null vector. The pose can be built from any representation of a pose:*

```
1 vpPoseVector r(ri) ; // where ri is a vpPoseVector
2 vpPoseVector r(t,theta) ; // where t is a vpTranslationVector
3 // and theta is a vpThetaUVector
4 vpPoseVector r(t,R) ; // where R is a vpRotationMatrix
5 vpPoseVector r(M) ; // where M is a vpHomogeneousMatrix
```

Alternatively, the operator `buildFrom(...)` can be used:

```
1 vpPoseVector r ; r.buildFrom(t,theta) ; // where t is a vpTranslationVector
2 // and theta is a vpThetaUVector
3 vpPoseVector r ; r.buildFrom(t,R) ; // where R is a vpRotationMatrix
4 vpPoseVector r ; r.buildFrom(M) ; // where M is a vpHomogeneousMatrix
```

It can also be built directly from six floats which correspond to the three components of the translation along the \mathbf{x}, \mathbf{y} and \mathbf{z} axes and the three components of the $\theta\mathbf{u}$ representation of the rotation:

```
1 vpPoseVector r(1.0,1.3,3.5,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Building a homogeneous transformation matrix \mathbf{M} (see section 1.4.2) from a pose vector is done through the constructor or the `buildFrom(...)` operator of the `vpHomogeneousMatrix` class.

```
1 vpHomogeneousMatrix M(r) ; // where r is a vpPoseVector
2 vpHomogeneousMatrix M ; M.buildFrom(r) ; // where r is a vpPoseVector
```

Access. Read/write access of the components is done using the `[]` operator:

```
1 vpPoseVector r ;
2 r[0] = 1.0 ; r[1] = 1.3 ; r[2] = 3.5 ; // tx, ty, tz
3 r[3] = 0.2 ; r[4] = 0.3 ; r[5] = 0.5 ; // thetatau, thetau, thetau
```

1.4.2 Homogeneous transformation matrix

Coordinates relative to a frame \mathcal{F}_a of a 3D point \mathbf{P} relative to another frame \mathcal{F}_b are obtained applying a rotation followed by a translation between the frames :

$${}^a\mathbf{P} = {}^a\mathbf{R}_b {}^b\mathbf{P} + {}^a\mathbf{t}_b \quad (13)$$

The space representing all these transformations is call the Special Euclidean transformation space :

$$SE(3) = \{m = (\mathbf{R}, \mathbf{t}) : \mathbf{R} \in SO(3), \mathbf{t} \in \mathbb{R}^3\} \quad (14)$$

As in most of the robotics literature this transformation (or pose) is represented using a homogeneous transformation matrix given by:

$${}^a\mathbf{M}_b = \begin{bmatrix} {}^a\mathbf{R}_b & {}^a\mathbf{t}_b \\ \mathbf{0}_3 & 1 \end{bmatrix} \quad (15)$$

where ${}^a\mathbf{R}_b$ and ${}^a\mathbf{t}_b$ are, the rotation matrix (see section 1.2.1) and translation vector (see section 1.3) from frame \mathcal{F}_a to \mathcal{F}_b . It is an equivalent representation of the minimal pose vector \mathbf{r} (see section 1.4).

The transformation of a homogeneous 3D point between a frame \mathcal{F}_a and a frame \mathcal{F}_b is expressed by:

$$\begin{bmatrix} {}^a\mathbf{P} \\ 1 \end{bmatrix} = {}^a\mathbf{M}_b \begin{bmatrix} {}^b\mathbf{P} \\ 1 \end{bmatrix} \quad (16)$$

Homogeneous matrices can be composed:

$${}^a\mathbf{M}_c = {}^a\mathbf{M}_b {}^b\mathbf{M}_c, \quad (17)$$

and inverted :

$${}^a\mathbf{M}_b^{-1} = {}^b\mathbf{M}_a = \begin{bmatrix} {}^a\mathbf{R}_b^T & -{}^a\mathbf{R}_b^T {}^a\mathbf{t}_b \\ \mathbf{0}_3 & 1 \end{bmatrix} \quad (18)$$

The `vpHomogeneousMatrix` class implements a homogeneous matrix. It inherits from the `vpMatrix` but most of the members have been overloaded for efficiency issue or to ensure the rotation matrix properties (For example the `+` or `-` operators are no longer defined).

Construction. *Default value for the homogeneous matrix is the identity matrix. The homogeneous matrix can be build either from another homogeneous matrix or from another representation of a pose:*

```

1 vpHomogeneousMatrix M(Mi) ; // where Mi is a vpHomogeneousMatrix
2 vpHomogeneousMatrix M(t,R) ; // where t is a vpTranslationVector
3                               // and R a vpRotationMatrix
4 vpHomogeneousMatrix M(t,theta) ; // where t is a vpTranslationVector
5                               // and theta a vpThetaUVector
6 vpHomogeneousMatrix M(r) ; // where r is a vpPoseVector

```

It can also be built directly from six float which correspond to the pose representation (see section 1.4):

```

1 vpHomogeneousMatrix M(1.2,2.3,1.0,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)

```

Alternatively, the method `buildFrom(...)` can be used:

```

1 vpHomogeneousMatrix M ; M.buildFrom(t,R) ; // where t is a vpTranslationVector
2                               // and R a vpRotationMatrix
3 vpHomogeneousMatrix M ; M.buildFrom(t,theta) ; // where t is a vpTranslationVector
4                               // and theta a vpThetaUVector
5 vpHomogeneousMatrix M ; M.buildFrom(r) ; // where r is a vpPoseVector
6
7 vpHomogeneousMatrix M ; M.buildFrom(1.2,2.3,1.0,0.2,0.3,0.5) ; // (tx,ty,tz,...
8                               // thetaux,thetauy,thetauz)

```

The method `insert(...)` can be used to insert the rotation and the translation components independently:

```

1 vpHomogeneousMatrix M ; M.insert(R) ; // where R is a vpRotationMatrix
2 vpHomogeneousMatrix M ; M.insert(theta) ; // where theta is a vpThetaUVector
3 vpHomogeneousMatrix M ; M.insert(t) ; // and t a vpTranslationVector

```

Extracting the rotation matrix R and the translation vector t from an homogeneous matrix is done using the `extract(...)` method.

```

1 vpRotationMatrix R ; M.extract(R) ; // where M is a vpHomogeneousMatrix
2 vpTranslationVector t ; M.extract(t) ;

```

Operations. *Let aMb and bMc define the poses between frame \mathcal{F}_a and \mathcal{F}_b (respectively \mathcal{F}_b and \mathcal{F}_c), the pose between \mathcal{F}_a and \mathcal{F}_c is computed by:*

```

1 vpHomogeneousMatrix aMb(...);
2 vpHomogeneousMatrix bMc(...);
3
4 vpHomogeneousMatrix aMc;
5
6 aMc = aMb*bMc;

```

In a similar way the inverse transformation can be obtained using:

```

1 vpHomogeneousMatrix cMa;
2
3 cMa = (aMb*bMc).inverse(); // or
4 cMa = aMc.inverse();

```

1.5 Twist transformation matrices

1.5.1 Velocity twist matrix

In a frame \mathcal{F}_a , the motion of an object is described by a translational velocity $\mathbf{v} = (v_x, v_y, v_z)$ and an angular velocity $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)$. The resulting vector ${}^a\mathbf{v} = (\mathbf{v}, \boldsymbol{\omega})$ is known in the robotics literature as a velocity screw in the frame \mathcal{F}_a .

The velocity twist transformation matrix allows us to transform velocity screws among coordinate frames. Knowing the velocity screw in the frame \mathcal{F}_b , the corresponding velocity screw in frame \mathcal{F}_a is obtained using:

$${}^a\mathbf{v} = {}^a\mathbf{V}_b {}^b\mathbf{v} \quad (19)$$

with

$${}^a\mathbf{V}_b = \begin{bmatrix} {}^a\mathbf{R}_b & [{}^a\mathbf{t}_b]_{\times} {}^a\mathbf{R}_b \\ \mathbf{0}_3 & {}^a\mathbf{R}_b \end{bmatrix} \quad (20)$$

where ${}^a\mathbf{R}_b$ and ${}^a\mathbf{t}_b$ are, respectively, the rotation matrix and translation vector from frame \mathcal{F}_a to \mathcal{F}_b . ${}^a\mathbf{V}_b$ is called the velocity twist matrix.

Velocity twist matrices are implemented in the `vpVelocityTwistMatrix` class. It inherits from the `vpMatrix` but most of the members have been overloaded for efficiency issue or to ensure the specific properties (for example the `+` or `-` operators are no longer defined).

Construction. *Default value for the velocity twist matrix is the identity matrix. The velocity twist matrix can be build either from another velocity twist matrix, from a translation and the representation of a rotation, or from an homogeneous matrix:*

```
1 vpVelocityTwistMatrix V(Vi) ; // where Vi is a vpVelocityTwistMatrix
2 vpVelocityTwistMatrix V(t,R) ; // where t is a vpTranslationVector
3 // and R a vpRotationMatrix
4 vpVelocityTwistMatrix V(t,theta) ; // where theta is a vpThetaUVector
5 vpVelocityTwistMatrix V(M) ; // where M is a vpHomogeneousMatrix
```

It can also be built directly from six floats which correspond to a translation and the $\theta\mathbf{u}$ representation (see section 1.2.2):

```
1 vpVelocityTwistMatrix V(1.2,2.3,1.0,0.2,0.3,0.5) ; // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Alternatively, the method `buildFrom(...)` can be used:

```
1 vpVelocityTwistMatrix V ; V.buildFrom(t,R) ; // where R is a vpRotationMatrix
2 // and t a vpTranslationVector
3 vpVelocityTwistMatrix V ; V.buildFrom(t,theta) ; // where theta is a vpThetaUVector
4 vpVelocityTwistMatrix V ; V.buildFrom(M) ; // where M is a vpHomogeneousMatrix
```

Operations. *Let ${}^a\mathbf{V}_b$ and ${}^b\mathbf{V}_c$ define the velocity twist matrices between frame \mathcal{F}_a and \mathcal{F}_b (respectively \mathcal{F}_b and \mathcal{F}_c), the velocity twist matrix between \mathcal{F}_a and \mathcal{F}_c is computed by:*

```
1 vpVelocityTwistMatrix aVb(...) ;
2 vpVelocityTwistMatrix bVc(...) ;
3
4 vpVelocityTwistMatrix aVc ;
5
6 aVc = aVb*bVc ;
```

There is no specific object in ViSP to represent velocity screws. Conversion of a velocity screw ${}^b\mathbf{v} = (\mathbf{v}, \boldsymbol{\omega})$ expressed in frame \mathcal{F}_b to a frame \mathcal{F}_a can be done like:

```
1 vpVelocityTwistMatrix aVb(...);
2 vpColVector av = aVb*bv; // where bv is a vpColVector of dimension 6
3 // av = (vx, vy, vz, wx, wy, wz) ^T
```

1.5.2 Force-torque twist matrix

Related to a frame \mathcal{F}_a , the force applied to an object is described by a force vector $\mathbf{f} = (\mathbf{f}_x, \mathbf{f}_y, \mathbf{f}_z)$ and a torque vector $\boldsymbol{\tau} = (\tau_x, \tau_y, \tau_z)$. The resulting vector ${}^a\mathbf{h} = (\mathbf{f}, \boldsymbol{\tau})$ is known in the robotics literature as a force-torque screw in the frame \mathcal{F}_a .

The force-torque twist transformation matrix allows us to transform force-torque screws among coordinate frames. Knowing the force-torque screw in the frame \mathcal{F}_b , the corresponding force-torque screw in frame \mathcal{F}_a is obtained using:

$${}^a\mathbf{h} = {}^a\mathbf{F}_b {}^b\mathbf{h} \quad (21)$$

with

$${}^a\mathbf{F}_b = \begin{bmatrix} {}^a\mathbf{R}_b & \mathbf{0}_3 \\ [{}^a\mathbf{t}_b]_{\times} {}^a\mathbf{R}_b & {}^a\mathbf{R}_b \end{bmatrix} \quad (22)$$

where ${}^a\mathbf{R}_b$ and ${}^a\mathbf{t}_b$ are, the rotation matrix and the translation vector from frame \mathcal{F}_a to \mathcal{F}_b . ${}^a\mathbf{F}_b$ is called the force-torque twist matrix.

Force-torque twist matrix is implemented in the `vpForceTwistMatrix` class. It inherits from the `vpMatrix` but most of the members have been overloaded for efficiency issue or to ensure the specific properties (for example the `+` or `-` operators are no longer defined).

Construction. *Default value for the force-torque twist matrix is the identity matrix. The force-torque twist matrix can be built either from another force-torque twist matrix, from a translation and a rotation, or from a homogeneous matrix:*

```
1 vpForceTwistMatrix F(Fi); // where Fi is a vpForceTwistMatrix
2 vpForceTwistMatrix F(t,R); // where t is a vpTranslationVector
3 // and R a vpRotationMatrix
4 vpForceTwistMatrix F(t,theta); // where theta is a vpThetaUVector
5 vpForceTwistMatrix F(M); // where M is a vpHomogeneousMatrix
```

It can also be built directly from six floats which correspond to a translation and a rotation in the $\theta\mathbf{u}$ representation (see section 1.2.2):

```
1 vpForceTwistMatrix F(1.2,2.3,1.0,0.2,0.3,0.5); // (tx,ty,tz,thetaux,thetauy,thetauz)
```

Alternatively, the method `buildFrom(...)` can be used:

```
1 vpForceTwistMatrix F; F.buildFrom(t,R); // where R is a vpRotationMatrix
2 // and t a vpTranslationVector
3 vpForceTwistMatrix F; F.buildFrom(t,theta); // where theta is a vpThetaUVector
4 vpForceTwistMatrix F; F.buildFrom(M); // where M is a vpHomogeneousMatrix
```

Operations. Let aF_b and bF_c be the force-torque twist matrices between frame \mathcal{F}_a and \mathcal{F}_b (respectively \mathcal{F}_b and \mathcal{F}_c), the force-torque twist matrix between \mathcal{F}_a and \mathcal{F}_c is computed by:

```

1 vpForceTwistMatrix aFb(...);
2 vpForceTwistMatrix bFc(...);
3
4 vpForceTwistMatrix aFc;
5
6 aFc = aFb*bFc;

```

There is no specific object in ViSP to represent force-torque screws. Conversion of a force-torque screw ${}^b\mathbf{h} = (\mathbf{f}, \boldsymbol{\tau})$ expressed in frame \mathcal{F}_b to a frame \mathcal{F}_a can be done like:

```

1 vpForceTwistMatrix aFb(...);
2 vpColVector ah = aFb*bh; // where bh is a vpColVector of dimension 6
3 // ah = (fx, fy, fz, tau_x, tau_y, tau_z)^T

```

2 Vision related transformations

2.1 Camera models

2.1.1 Perspective camera

The perspective model, or pin-hole model, is the most used camera model in computer vision. The camera is denoted $\mathcal{F}_C = (C, \mathbf{x}, \mathbf{y}, \mathbf{z})$. The origin C of \mathcal{F}_C is the projection center. The projection plane or (normalized) image plane π is parallel to the (\mathbf{x}, \mathbf{y}) plane. The intersection of the optical axis (\mathbf{z} axis) with the image plane is called the principal point and is denoted \mathbf{x}_c .

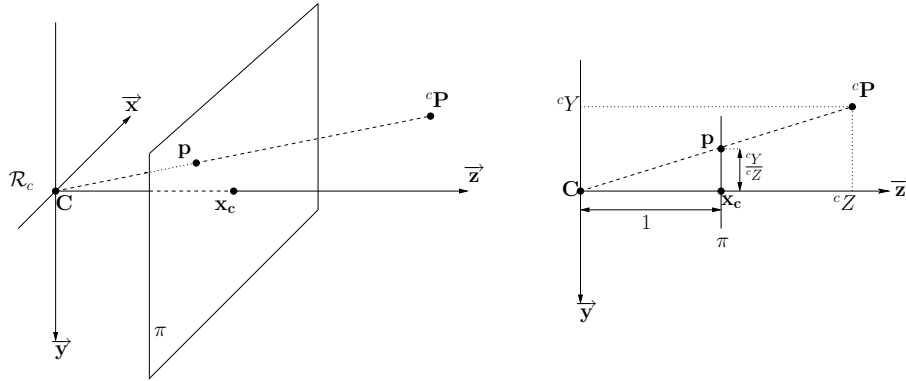


Figure 1: Perspective projection model.

Given a 3D point ${}^c\mathbf{P} = ({}^cX, {}^cY, {}^cZ, 1)$ in \mathcal{F}_C , the perspective projection of ${}^c\mathbf{P}$ in the image plane $\mathbf{p} = (x, y, 1)$ is given by:

$$\mathbf{p} = \mathbf{M}{}^c\mathbf{P}$$

or

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} {}^cX \\ {}^cY \\ {}^cZ \\ 1 \end{bmatrix}$$

It finally gives:

$$x = \frac{{}^cX}{cZ}, y = \frac{{}^cY}{cZ} \quad (23)$$

Remark: *The equation (23) is only valid for points. ViSP allows the projection of various features (point, straight line, circle, sphere, ...).*

Given a 3D point in the object frame \mathcal{F}_o and the camera pose ${}^c\mathbf{M}_o$, it is possible to compute the homogeneous coordinates of the point projected in the image plane:

```

1 vpHomogeneousMatrix cMo ; cMo[2][3] = 2 ; // the camera pose
2
3 vpPoint P ;
4 // set the point coordinates in the object frame
5 P.setWorldCoordinates(1,1,0) ;
6 cout << P.oP << endl ; // output (1,1,0)
7
8 // compute the point coordinates in the camera frame
9 P.changeFrame(cMo) ;
10 cout << P.cP << endl ; // output (1,1,2)
11
12 // project the 3D point in the 2D image plane
13 P.project() ;
14 cout << P.p << endl ; // output (0.5, 0.5)

```

$\mathbf{p} = (x, y, 1)$ coordinates are expressed in meters. To compute the point coordinates in the image frame (ie, in pixel) we have to consider the camera intrinsic parameters. This transformation is given by the following equations :

$$\mathbf{m} = \mathbf{K}\mathbf{p} \quad \text{with} \quad \mathbf{K} = \begin{bmatrix} p_x & 0 & u_0 \\ 0 & p_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\mathbf{x}_c = (u_0, v_0)$ are the coordinates of the principal point and (p_x, p_y) are the ratio between the focal length and the size of a pixel.

We obtain:

$$\begin{cases} u = p_x x + u_0, \\ v = p_y y + v_0 \end{cases} \quad (24)$$

The previous equation assume a perfect perspective camera which is not always the case. In practice we have to consider more complex (and usually non linear) camera model. In ViSP it is possible to introduce parameters to models the radial distortion. The model becomes:

$$\begin{cases} u = u_0 + p_x x (1 + k_{ud}(x^2 + y^2)), \\ v = v_0 + p_y y (1 + k_{ud}(x^2 + y^2)) \end{cases} \quad (25)$$

where k_{ud} is a distortion parameter.

If, using this model, the computation of coordinates in pixels from coordinates expressed in meter is straightforward, the inverse transformation is far more complex. To cope with the issue we introduce a second distortion parameter k_{du} ($k_{ud} \neq k_{du}$) which allows to compute the coordinates in meter from the

coordinates expressed in pixels:

$$\begin{cases} x = \frac{u-u_0}{p_x} \left(1 + k_{du} \left(\left(\frac{u-u_0}{p_x} \right)^2 + \left(\frac{v-v_0}{p_y} \right)^2 \right) \right), \\ y = \frac{v-v_0}{p_y} \left(1 + k_{du} \left(\left(\frac{u-u_0}{p_x} \right)^2 + \left(\frac{v-v_0}{p_y} \right)^2 \right) \right), \end{cases} \quad (26)$$

k_{ud} is used to convert a feature from an undistorted to a distorted representation whereas k_{du} is used to convert a feature from a distorted to an undistorted representation.

Camera parameters implementation In ViSP, the camera intrinsic parameters are stored in the `vpCameraParameters` class.

The selection of the modelisation is done during the `vpCameraParameters` initialisation.

For example, to initialise a camera without distortion:

```

1 #include <visp/vpCameraParameters.h>
2
3 double px = 600;
4 double py = 600;
5 double u0 = 320;
6 double v0 = 240;
7
8 // Create a camera parameter container
9 vpCameraParameters cam;
10
11 // Camera initialization with a perspective projection without distortion model
12 cam.initPersProjWithoutDistortion(px,py,u0,v0);
13
14 // It is also possible to print the current camera parameters
15 cam.printParameters();

```

The same initialisation (for a model without distortion) can be done by:

```

1 #include <visp/vpCameraParameters.h>
2 #include <visp/vpMatrix.h>
3
4 // Create a matrix container for the camera parameters.
5 vpMatrix M(3, 3);
6
7 // Fill the matrix with the camera parameters
8 M[0][0] = 600; // px
9 M[1][1] = 600; // py
10 M[0][2] = 320; // u0
11 M[1][2] = 240; // v0
12 M[2][2] = 1;
13
14 // without distortion model is set.
15 vpCameraParameters cam;
16
17 // Set the parameters
18 cam.initFromCalibrationMatrix(M);

```

Initialisation of a camera for a model with distortion:

```

1 #include <visp/vpCameraParameters.h>
2
3 double px = 600;
4 double py = 600;
5 double u0 = 320;

```

```

6 double v0 = 240;
7 double kud = -0.19;
8 double kdu = 0.20;
9
10 // Create a camera parameter container
11 vpCameraParameters cam;
12
13 // Camera initialization with a perspective projection with distortion model
14 cam.initPersProjWithDistortion(px, py, u0, v0, kud, kdu);

```

The code below shows how to know if the current projection model uses distortion parameters:

```

1 vpCameraParameters cam;
2 ...
3 vpCameraParameters::vpCameraParametersProjType projModel;
4 projModel = cam.get_projModel(); // Get the projection model type

```

Camera parameters parser ViSP allows to save and load camera parameters in a formatted file using a XML parser.

Example of an XML file "myXmlFile.xml" containing intrinsic camera parameters:

```

1 <?xml version="1.0"?>
2 <root>
3   <camera>
4     <name>myCamera</name>
5     <image_width>640</image_width>
6     <image_height>480</image_height>
7     <model>
8       <type>perspectiveProjWithoutDistortion</type>
9       <px>1129.0</px>
10      <py>1130.6</py>
11      <u0>317.9</u0>
12      <v0>229.1</v0>
13    </model>
14    <model>
15      <type>perspectiveProjWithDistortion</type>
16      <px>1089.9</px>
17      <py>1090.1</py>
18      <u0>326.1</u0>
19      <v0>230.5</v0>
20      <kud>-0.196</kud>
21      <kdu>0.204</kdu>
22    </model>
23  </camera>
24 </root>

```

The following code shows how to load the camera parameters from a XML file:

```

1 #include <visp/vpCameraParameters.h>
2 #include <visp/vpXmlParserCamera.h>
3
4 vpCameraParameters cam; // Create a camera parameter container
5 vpXmlParserCamera p; // Create a XML parser
6 vpCameraParameters::vpCameraParametersProjType projModel; // Projection model
7
8 // We want here to use a perspective projection model without distortion
9 projModel = vpCameraParameters::perspectiveProjWithoutDistortion;
10
11 // Parse the xml file "myXmlFile.xml" to find the intrinsic camera
12 // parameters of the camera named "myCamera" for 640x480 images and for
13 // the specified projection model. Note that the size of the image is optional
14 // if in the XML file camera parameters are provided only for one image size.

```

```

15 p.parse(cam, "myXmlFile.xml", "myCamera", projModel, 640, 480);
16
17 // Print the parameters
18 std::cout << "Camera Parameters: " << cam << std::endl;
19
20 // Save the parameters in a new file "myXmlFileWithNoise.xml"
21 p.save(cam, "myXmlFileWithNoise.xml", p.getCameraName(), p.getWidth(), p.getHeight());

```

The following code shows how to write the camera parameters in a XML file:

```

1 // Create a camera parameter container. We want to set these parameters
2 // for a 320x240 image, and we want to use the perspective projection
3 // modelisation without distortion.
4 vpCameraParameters cam;
5
6 // Set the pixel ratio (px, py) and the principal point coordinates (u0,v0)
7 cam.initPersProjWithDistortion(563.2, 564.1, 162.3, 122.4); // px,py,u0,v0
8 // Create a XML parser
9 vpXmlParserCamera p;
10 // Save the camera parameters in an XML file.
11 p.save(cam, "myXmlFile.xml", "myNewCamera", 320, 240);

```

Pixel/Meter conversion. The equations (24), (25) and (26) allows the pixel to meter (respectively meter to pixel) conversion for points. These conversions are implemented in the `vpPixelMeterConversion` class (respectively the `vpMeterPixelConversion` class).

To use meter to pixel and pixel to meter conversion, we have first to include the corresponding header files:

```

1 #include <visp/vpPixelMeterConversion.h>
2 #include <visp/vpMeterPixelConversion.h>

```

Then we have to initialise a `vpCameraParameters` as described above with the desired projection model. Here we choose the model without distortion.

```

1 #include <visp/vpCameraParameters.h>
2 vpCameraParameters cam;
3 cam.initPersProjWithoutDistortion(px,py,u0,v0); // where px,py are the pixel ratio
4 // and u0,v0 the principal point coordinates

```

We can finally call the desired conversion function, here we want to convert a point:

```

1 vpPixelMeterConversion::convertPoint(cam,u,v,x,y); // where (u,v) are pixel coordinates
2 // to convert in meter coordinates (x,y)
3
4 vpMeterPixelConversion::convertPoint(cam,x,y,u,v); // where (x,y) are meter coordinates
5 // to convert in pixel coordinates (u,v)

```

ViSP proposes similar conversions for straight lines and moments. But it can be noticed these conversions have no meaning for the projection model with distortion.

Remark: ViSP implements a calibration tools that allows to compute the camera intrinsic parameters.

2.1.2 Catadioptric sensor

To date this projection model is not considered in ViSP.

2.2 Homography

The homography has been introduced in [2]. It is a \mathbb{R}^2 to \mathbb{R}^2 transformation and is valid for all camera motions.

The main idea is to use the 3D coordinates of a plane \mathcal{P} which is linked to the 3D point \mathbf{P} . At first, the point is considered to belong to the plane. In the section 2.2.1, the point is considered to belong the plane; while in the section 2.2.2 this constraint is removed.

2.2.1 Planar structure

We define ${}^a\mathbf{n}$ and d_a the normal to the plane \mathcal{P} and the distance to the projection center \mathcal{C}_a as illustrated on figure 2

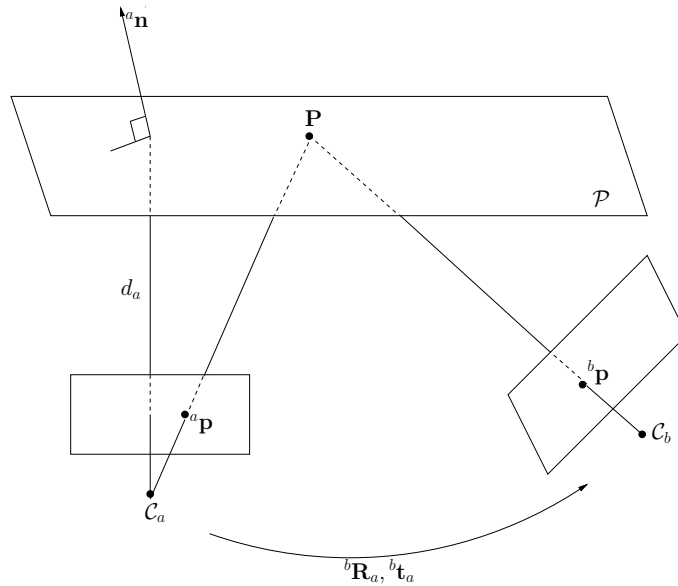


Figure 2: Illustration of the motion from one view to another using an homography in the case of a planar structure.

If the point \mathbf{P} belongs to this plane, then its coordinates ${}^a\mathbf{P} = (X_a, Y_a, Z_a)$ in the frame \mathcal{F}_a verify :

$${}^a\mathbf{n}^T {}^a\mathbf{P} = d_a \quad (27)$$

With this relation, equation (13) can be factorized by ${}^a\mathbf{P}$

$${}^b\mathbf{P} = \left({}^b\mathbf{R}_a + \frac{{}^b\mathbf{t}_a {}^a\mathbf{n}^T}{d_a} \right) {}^a\mathbf{P} \quad (28)$$

The projection equation (23) giving $Z_a {}^a\mathbf{p} = {}^a\mathbf{P}$, it's now possible to link ${}^a\mathbf{p}$ and ${}^b\mathbf{p}$ coordinates by an

homography :

$$\begin{aligned} Z_b {}^b \mathbf{p} &= \left({}^b \mathbf{R}_a + \frac{{}^b \mathbf{t}_a {}^a \mathbf{n}^\top}{d_a} \right) Z_a {}^a \mathbf{p} \\ \frac{Z_b}{Z_a} {}^b \mathbf{p} &= \left({}^b \mathbf{R}_a + \frac{{}^b \mathbf{t}_a {}^a \mathbf{n}^\top}{d_a} \right) {}^a \mathbf{p} \\ \frac{Z_b}{Z_a} {}^b \mathbf{p} &= {}^b \mathbf{H}_a {}^a \mathbf{p} \end{aligned} \quad (29)$$

(30)

The homography is the 3×3 matrix :

$${}^b \mathbf{H}_a = {}^b \mathbf{R}_a + \frac{{}^b \mathbf{t}_a {}^a \mathbf{n}^\top}{d_a} \quad (31)$$

which allows to make a point-to-point correspondence thanks to the introduction of the plane equation. It is an homogeneous transformation defined up to a scale factor. It has eight degrees of freedom.

2.2.2 Non planar structure

If the point doesn't belong to \mathcal{P} (figure 3), the homography ${}^b \mathbf{H}_a$ related to the plane is not sufficient to describe the relation between ${}^a \mathbf{p}$ and ${}^b \mathbf{p}$. As it will be seen, the relation (29) will be improved by a term taking into account the signed distance d between the point and the plane. Noting :

$$d = d_a - {}^a \mathbf{n}^\top (Z_a {}^a \mathbf{p}) \quad (32)$$

In that case, the operation realised in equation (28) can't be simplified and gives :

$${}^b \mathbf{p} = \left({}^b \mathbf{R}_a + \frac{{}^b \mathbf{t}_a {}^a \mathbf{n}^\top}{d_a} \right) {}^a \mathbf{p} + \left(1 - \frac{{}^a \mathbf{n}^\top}{{}^a \mathbf{p}} \right) {}^b \mathbf{t}_a \quad (33)$$

Or using ${}^a \mathbf{p}$ and ${}^b \mathbf{p}$ coordinates :

$$Z_b {}^b \mathbf{p} = Z_a {}^b \mathbf{H}_a {}^a \mathbf{p} + \left(1 - Z_a \frac{{}^a \mathbf{n}^\top}{{}^a \mathbf{p}} \right) {}^b \mathbf{t}_a \quad (34)$$

Dividing equation members by Z_a , the equivalent relation of (29) in a non-planar structure is :

$$\begin{aligned} \frac{Z_b}{Z_a} {}^b \mathbf{p} &= {}^b \mathbf{H}_a {}^a \mathbf{p} + \frac{d}{Z_a d_a} {}^b \mathbf{t}_a \\ \frac{Z_b}{Z_a} {}^b \mathbf{p} &= {}^b \mathbf{H}_a {}^a \mathbf{p} + \beta {}^b \mathbf{t}_a \end{aligned} \quad (35)$$

The term $\beta = \frac{d}{Z_a d_a}$ is called parallax [1, 3]. This term only depends on the parameters expressed in the camera frame \mathcal{F}_a . It measures a relative depth between the point \mathbf{P} and the plane \mathcal{P} . Its sign allows to know on which side of the plane is the point. β allows to find the corresponding point of ${}^a \mathbf{p}$ in the image I_b along the epipolar line l_b of the projected point ${}^b \mathbf{p}' = {}^b \mathbf{H}_a {}^a \mathbf{p}$ (which can be seen as the corresponding point of ${}^a \mathbf{p}$ considering that \mathbf{P} belong to the plane (figure 3)).

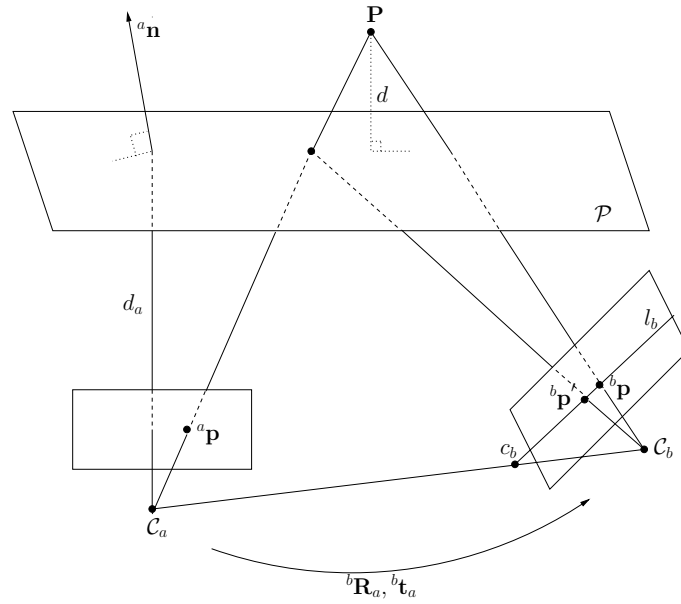


Figure 3: Illustration of the motion from a view to another using an homography in the case of a non planar structure.

References

- [1] B. Boufama and R. Mohr. Epipole and fundamental matrix estimation using virtual parallax. In *IEEE Int. Conf. on Computer Vision*, pages 1030–1036, 1995.
- [2] O. Faugeras and F. Lustman. Motion and structure from motion in a piecewise planar environment. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 2(3):485–508, 1988.
- [3] B. Triggs. Plane + parallax, tensors and factorization. In *European Conference on Computer Vision, ECCV'00*, pages 522–538, Dublin, Eire, May 2000.