OPENROX

User's Manual of Version 1.0

www.robocortex.com

Contents

Chapter 1

Introduction

OPENROX is a computer vision software written in strict ANSI-C and optimized for real-time applications. The optimization has been performed with SSE 4.2 for x86 processors and Neon for Arm processors. No extra dependencies are required to compile the software, just the standard C library. The software provides advanced algorithms that can be used for applications like vision-guided robot control and augmented reality.

Contrarily to standard computer vision software, **OPENROX** uses **direct methods** and **dense information** to solve computer vision problems. This approach is more robust and accurate than sparse features-based methods, but generally requires high computation resources. The use of a new fast optimization technique called **ESM (Efficient Second-order Approximation Method)** allows to use dense direct methods in real-time. The ESM technique was proved to have a higher convergence rate (third-order) than the Newton optimization technique (second-order) while being more efficient.

The present version of the **OPENROX** is composed of the following modules :

Utils : module with basic structures and functions used by all the other modules ;

Maths : module with mathematical structures and methods ;

Model : module with structures and functions to handle target models ;

Vision : module with structures and functions for computer vision ;

Sensor : module with structures and functions for handling sensors ;

Odometry : module with structures and methods for visual odometry computation (sensor localization). The output of the module is the pose of the sensor (translation and rotation in the Cartesian space) relative to a target model ;

1.1 Quick guide

In the directory openrox, you will find the following directories:

- bin : The directory containing the dynamic libraries and used to build the executable examples. By default, the license file shall be copied to this directory.
- doc : The directory containing the documentation ;
- inc : The directory containing the headers for the various structures and functions ;
- lib : The directory containing the static libraries ;
- lic : The directory containing the license file ;
- res : Directory needed to save example results ;
- seq : The directory containing the sequence of images (.pgm files) used to test the applications. More test sequences can be downloaded from Robocortex website at www.robocortex.com. Instructions are given in eache example file;
- src : Directory containing several examples for using **OPENROX**. Examples are provided to test the library and commented in order to help the user understand how to use and make his own application.;

To compile the examples go to the bin directory:

```
cd bin
```

then use cmake-gui (can be downloaded from http://www.cmake.org/) to generate the platform specific projects to compile the examples:

```
cmake-gui ..
```

Make sure that cmake-gui is added to the system PATH.

1.2 License file

A license file shall be obtained to use **OPENROX** and run the example. By default, the license file shall be copied to the "lic" directory. The user can put the file in a different directory by changing the path in the example files.

Chapter 2

Utils Module

The utils module provides basic structures and functions used by all the other modules. The utils module contains the following sub-modules:

Types : Module with common macros and types definitions ;

Error : Module with structures and methods for handling errors ;

Timer : Module with structures and methods for time measurement ;

License : Module with structures and methods for file manipulation ;

2.1 Types

The standard ANSI C types are renamed with the prefix "Rox_" in order to ensure portability on different devices. If you need further information about the renamed types, please refer to the Programmer Manual.

2.2 Error

Most of the **OPENROX** functions return an error code that can be displayed in human readable format using the following function:

```
Rox_Void rox_error_display (Rox_Error error);
```

Error code is 0 when the function exit successfully.

2.3 Timer

The Timer module contains functions for time measurements. The module is useful to measure the computation time when developing.

2.3.1 The Rox_Timer object

A Rox_Timer object can be defined using the pointer to a Rox_Timer:

```
typedef struct Rox_Timer_Struct* Rox_Timer;
```

The Rox_Timer object can be used both under Linux, Windows or MacOS X without any change.

2.3.2 Creating/Deleting a Rox_Timer

The following function allows to create/delete a Rox_Timer object:

rox_timer_new : Create a Rox_Timer and allocate memory.

rox_timer_del : Delete a Rox_Timer and free memory.

2.3.3 Main functions related to Rox_Timer

The following functions shall be used for time measurement:

rox_timer_start : Start the timer.

rox_timer_stop : Stop the timer and return the time elapsed since the initialization.

Example :

```
//Create Rox_Timer
Rox_Timer timer = 0

//Create Rox_Timer
rox_timer_new(&timer);

//Start the timer
rox_timer_start(timer);

// Here comes the code for which you make time measurements
...

// Stop the timer
Rox_Float time_ellapsed = rox_timer_stop(timer);

// Display time elapsed
printf("Elapsed␣Time␣=␣\%f␣ms", time_elapsed \n);

// Delete the timer
rox_timer_del(&timer);
```

Information about timers is available in the Programmer Manual.

2.4 License

2.4.1 License activation using a file

The user can activate the **OPENROX** license by providing a license file to the following function:

```
Rox_Error rox_license_activate_file (const char *filename);
```

2.4.2 License activation using a serial number

The user can activate the **OPENROX** license by providing a serial number to the following function:

```
Rox_Error rox_license_activate_serial_number (const Rox_Uint *
   serial_number);
```

Important remark: this mode is mandatory if the the dynamic libraries of **OPENROX** are distributed.

Chapter 3

Maths Module

The maths module provides mathematical structures and methods such as matrix and vector manipulations. The maths module contains the following sub-modules:

Linalg : Module with structures and methods for linear algebra ;

Matrix : Module with structures and methods for matrix manipulation ;

Special Matrices : Module with structures and methods for special matrices manipulation ;

Geometry : Module with structures and methods for geometry.

3.1 Linear Algebra

The linear algebra module contains method to manipulate vectors, vector spaces, linear maps and systems of linear equations.

3.1.1 Matrix

The matrix module contains matrix definitions and necessary functions to manipulate matrix.

3.1.1.1 The `Rox_Matrix` object

A matrix object can be declared using the pointer to a `Rox_Matrix_Struct`:

```
typedef struct Rox_Matrix_Struct* Rox_Matrix;
```

The structure is opaque to the user and can only be accessed through constructors, destructors and methods described in the following sections.

3.1.1.2 Creating/Deleting a `Rox_Matrix`

Functions are provided to allocate, initialize and deallocate an `Rox_Matrix` object :

```
Rox_Error rox_matrix_new(Rox_Matrix* matrix, const Rox_Uint cols
   , const Rox_Uint rows);
```

The `rox_matrix_new` function allocates memory for data. In this case, the size of allocated memory depends of parameters 'cols' and 'rows'.

```
Rox_Error rox_matrix_del(Rox_Matrix * M);
```

The `rox_matrix_del` function deallocates memory for an `Rox_Matrix` structure. It is necessary to call this function when the structure is not used any more or before overwriting it.

3.1.1.3 Main functions related to `Rox_Matrix`

Several functions are provided for handling matrices.

Firstly, it is possible to get / set information from `Rox_Matrix` structure:

rox_matrix_get_rows : Returns the number of rows of the matrix.

rox_matrix_get_cols : Returns the number of columns of the matrix.

rox_matrix_get_value : Returns a value of the matrix.

rox_matrix_set_value : changes a value of the matrix.

rox_matrix_set_zero : Sets to 0 all matrix elements.

rox_matrix_set_unit : Sets the identity matrix.

Other functions are provided, further information is available in the programmer manual.

3.1.2 Special Linear Group 3×3

The Special Linear Group SL(3) is defined by the set of 3×3 matrices $\mathbf{H}$ that verify the following constraint:

$$\det(\mathbf{H}) = \mathbf{1}$$

3.1.2.1 The object `Rox_MatSL3`

A `Rox_MatSL3` object can be declared using the pointer to a `Rox_MatSL3_Struct`:

```
typedef struct Rox_MatSL3_Struct* Rox_MatSL3;
```

The structure is opaque to the user and can only be accessed through constructors, destructors and methods described in the following sections.

3.1.2.2 Creating/Deleting a `Rox_MatSL3`

Functions are provided to allocate, initialize and deallocate an `Rox_MatSL3` object :

```
Rox_Error rox_matsl3_new(Rox_MatSL3 * H);
```

The `rox_matsl3_new` function allocates memory for data. By default, the matrix is initialized to the identity.

```
Rox_Error rox_matsl3_del(Rox_MatSL3 * H);
```

The `rox_matsl3_del` function deallocates memory for an `Rox_MatSL3` structure. It is necessary to call this function when the structure is not used any more or before overwriting it.

3.1.2.3 Main functions related to `Rox_MatSL3`

Firstly, it is possible to get / set information from `Rox_MatSL3` structure:

rox_matsl3_get_data : Get a copy of the data of the matsl3 matrix.

rox_matsl3_set_data : Set the data of the matsl3 matrix.

rox_matsl3_set_value : changes a value of the matrix.

Several functions are provided to use matrices, here the most important functions will be described to make basic matrix calculus.

rox_matsl3_mulmatmat : Computes the multiplication of two matrices.

rox_matsl3_inv : Computes the inverse of the matrix.

rox_matsl3_display : Display the matrix.

Other functions are provided, further information is available in the programmer manual.

3.1.3 Matrices of the Special Euclidean Group

The Special Euclidean Group SL(3) is defined by the set of 4×4 matrices $\mathbf{T}$:

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{bmatrix}$$

where $\mathbf{R} \in SO(3)$ is a rotation matrix that verify the following constraint:

$$\mathbf{R}^\top \mathbf{R} = \mathbf{I}$$

Setting:

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

and

$$\mathbf{t} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

The matrix $\mathbf{T}$ can be written as:

$$\mathbf{T} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.1.3.1 The object `Rox_MatSE3`

A matse3 object can be declared using the pointer to a `Rox_MatSE3_Struct`:

```
typedef struct Rox_MatSE3_Struct* Rox_MatSE3;
```

The structure is opaque to the user and can only be accessed through constructors, destructors and methods described in the following sections.

3.1.3.2 Creating/Deleting a `Rox_MatSE3`

Functions are provided to allocate, initialize and deallocate an `Rox_MatSE3` object :

```
Rox_Error rox_matse3_new(Rox_MatSE3 * T);
```

The `rox_matse3_new` function allocates memory for data. By default, the matrix is initialized to the identity.

```
Rox_Error rox_matse3_del(Rox_MatSE3 * T);
```

The `rox_matse3_del` function deallocates memory for an `Rox_MatSE3` structure. It is necessary to call this function when the structure is not used any more or before overwriting it.

3.1.3.3 Main functions related to `Rox_MatSE3`

Firstly, it is possible to get / set information from `Rox_MatSE3` structure:

rox_matse3_get_data_copy : Get a copy of the data of the matse3 matrix.

rox_matse3_get_value : Returns a value of the matse3 matrix.

rox_matse3_set_data : Set the data of the matse3 matrix.

rox_matse3_set_value : changes a value of the matrix.

Several functions are provided to use matrices, here the most important functions will be described to make basic matrix calculus.

rox_matse3_mulmatmat : Computes the multiplication of two matrices.

rox_matse3_inv : Computes the inverse of the matrix.

rox_matse3_display : Display the matrix.

Other functions are provided, further information is available in the programmer manual.

3.2 Geometry

This module contains structures and methods for geometry manipulation.

The object `Rox_Point2D_Double` and `Rox_Point3D_Double` are made to contain the coordinates of 2D and 3D points.

```
typedef struct Rox_Point2D_Double_Struct Rox_Point2D_Double;

typedef struct Rox_Point3D_Double_Struct Rox_Point3D_Double;
```

The structures are defined as follows:

```
struct Rox_Point2D_Double_Struct
{
   /*! The u-axis coordinate in pixel*/
   Rox_Real u;
   /*! The v-axis coordinate in pixel*/
   Rox_Real v;
};

struct Rox_Point3D_Double_Struct
{
   /*! The x-axis coordinate in meters*/
   Rox_Real X;
   /*! The y-axis coordinate in meters*/
   Rox_Real Y;
   /*! The z-axis coordinate in meters*/
   Rox_Real Z;
};
```

More information about the use of geometry objects is available in the Programmer Manual

Chapter 4

Vision Module

The vision module contains structures and methods for computer vision and contains the following sub-modules:

Image : Module with structures and methods for manipulating several image format ;

Image Mask : Module with structures and methods for image mask manipulation.

Image Display : Module with structures and methods for image display.

Identification : Module with structures and methods for image identification.

4.1 Image

The image module contains structures and methods for manipulating several image formats.

4.1.1 The object `Rox_Image`

An image object can be declared using the pointer to a `Rox_Image_Struct` structure.

```
typedef struct Rox_Image_Struct* Rox_Image;
```

4.1.2 Creating/Deleting a `Rox_Image`

Functions are provided to allocate and deallocate a `Rox_Image` object :

```
Rox_Error rox_image_new(Rox_Image * image, Rox_Uint cols,
   Rox_Uint rows);
```

The function allocates memory for data depending parameters 'cols' and 'rows' which correspond to image size.

An image can be created by directly reading a PGM file from disk using the following function:

```
Rox_Error rox_image_new_readpgm (Rox_Image *image, const char *
   filename);
```

The function first allocate memory for the structure with the rows and cols parameters matching the image file to load. Then it stores the image data in the image object.

The following function deallocates memory for a `Rox_Image`:

```
Rox_Error rox_image_del(Rox_Image * image);
```

It is necessary to call this function when the object is not used anymore.

4.1.3 Main functions related to `Rox_Image`

Once the object `Rox_Image` has been allocated, it is possible to get information from `Rox_Image`.

rox_image_get_rows : Get the image rows.

rox_image_get_cols : Get the image columns.

An image can be read from a PGM file if the size is compatible. Typically, we use a `rox_image_new_readpgm` function to read the first image of a sequence and allocate correctly the corresponding structure. Then, assuming that all images of a sequence have the same dimensions, we can re-use the same structure for the next images, using the `rox_image_readpgm` function. Thus, the structure is safely allocated while loading the reference image and the other images are read previously to perform the tracking. Currently, the only supported format in the library is raw PGM (Portable Gray Map) [1]:

```
Rox_Error rox_image_readpgm (Rox_Image image, const char *
   filename);
```

The function returns an error if the dimensions of the image file do not match the dimesions of the allocated image.

External libraries can be easily used to read images from files. Then, **OPENROX** allows to input several image formats. The list of available image formats is given in the enumeration `Rox_Image_Format_Enum` (see the Programmer Manual for a complete list):

```
enum Rox_Image_Format_Enum{
     Rox_Image_Format_Grays,
     Rox_Image_Format_YUV422,
     Rox_Image_Format_Color_RGBA,
```

[1] http://netpbm.sourceforge.net/doc/pgm.html

```
    Rox_Image_Format_Color_BGRA,
    Rox_Image_Format_Color_ARGB,
    Rox_Image_Format_Color_BGR,
    Rox_Image_Format_Color_RGB
};
```

The image data can be set using the following function:

```
Rox_Error rox_image_set_data (Rox_Image image, Rox_Uchar *data,
   Rox_Uint bytesPerRow, enum Rox_Image_Format format);
```

This function transforms an external image buffer to the rox internal format and store this information inside the Rox_Image object. The input image format shall be a 3 or 4 channel interleaved image containing Red, Blue and Green channels. Only one of these channels (specified by the channel parameter) will be used while converting to rox internal format. User needs to specify the exact format of the input buffer (e.g. RGB, BGR, RGBA, etc.) as it is impossible to guess it automatically, as well as how many bytes are used per row in the input image.

Finally, in order to know if an image suitable for identification and odometry computation, the following functions allows to get a quality score.

The first function is fast but do not provide precise information:

```
Rox_Error rox_image_get_quality_score (Rox_Real *score,
   Rox_Image image);
```

The second function is slower but provides precise information on the quality of an image:

```
Rox_Error rox_image_get_quality_score_precise (Rox_Real *score,
   Rox_Image image);
```

4.2 Image mask

4.2.1 The object `Rox_Imask`

The image structures of **OPENROX** can be coupled with an image mask structure `Rox_Imask_Structure` that contains a mask.

A mask can be declared using the pointer `Rox_Imask` to a `Rox_Imask_Structure`:

```
typedef struct Rox_Imask_Struct* Rox_Imask;
```

4.2.2 Creating/Deleting a `Rox_Imask`

Functions are provided to allocate and deallocate a `Rox_Imask` object :

```
Rox_Error rox_imask_new(Rox_Imask * imask, Rox_Uint cols,
   Rox_Uint rows);
```

The `rox_imask_new` function allocates memory for a mask object. In this case, the size of allocated memory depends on parameters 'cols' and 'rows'.

The `rox_imask_del` function deallocates memory for a mask object.

```
Rox_Error rox_imask_del(Rox_Imask * M);
```

It is necessary to call this function when the structure is not used anymore.

4.2.3 Main functions related to `Rox_Imask`

To each pixel of an image corresponds a mask information. If a mask byte is set to 0 (zero), then the corresponding pixel shall be ignored. If a mask byte is set to $\sim$0 (bitwise complement to zero), then the corresponding pixel shall be considered. By default, an image mask is fully set to $\sim$0 (bitwise complement to zero).

The user can easily define his own mask for a given image. The main application in the tracking algorithm is to define a non-rectangular reference region. For example, it is possible to define a circle where all mask bytes inside this circle are set to $\sim$0 (bitwise complement to zero) and all others to 0 (zero). Thus the region to track is not a rectangle but a circle, even if the image structure represents a rectangular region of interest.

The user can create his own functions to set the image mask as he wants. The only thing to do is to set to 0 the mask bytes corresponding to ignored pixels and to $\sim$0 (bitwise complement to zero) all the others.

A function is available to fill a mask object with user defined data :

```
void rox_imask_set_user(Rox_Imask imask, const Rox_Uchar* data);
```

The 'imask' parameter is a `Rox_Imask` structure already allocated with `rox_imask_new`. The second parameter contains the mask values defined by the user.

The following functions are provided to create basic masks:

rox_imask_set_zero : Sets the mask values to zero.

rox_imask_set_one : Sets the mask values to one.

rox_imask_set_ellipse : Sets a mask with elliptic shape.

Example of mask objects settings :

```
{
  // Create and allocate Rox_Imask structures
  Rox_Uint size = 64; // The mask will be square
  Rox_Imask Ellipse = rox_imask_new(size, size);
```

```
Rox_Imask User_defined = rox_imask_new(size, size);
Rox_Imask Polygon = rox_imask_new(size, size);
Rox_Imask Combination = rox_imask_new(size, size);

// Data for the mask defined by the user
Rox_Uchar Data[size * size];

// Initialize Data
for(int i = 0; i<size; i++)
{
  for(int j = 0; j<size; j++)
      Data[i+j*size] = ~0;
}

for(int i = size/2; i<size; i++)
{
  for(int j = 0; j<size/2; j++)
      Data[i+j*size] = 0;
}

for(int i = 0; i<size/2; i++)
{
  for(j = size/2; j<size; j++)
      Data[i+j*size] = 0;
}

//Initialize points for polygon data (In this example, it is a
    triangle)
const Rox_Uint points [6] = { 4, 3, 50, 32, 14, 57};

// Set Masks
rox_imask_set_ellipse(Ellipse);
rox_imask_set_user(User_defined, Data);
rox_imask_set_polygon(Polygon, points, 3);
rox_imask_set_and(Combination, Polygon, User_defined);

// Save masks
rox_imask_savepgm("Ellipse.pgm", Ellipse);
rox_imask_savepgm("User.pgm", User_defined);
rox_imask_savepgm("Combination.pgm", Combination);
rox_imask_savepgm("Polygon.pgm", Polygon);

// Free Memory
rox_imask_del(User_defined);
rox_imask_del(Ellipse);
```

```
  rox_imask_del(Polygon);
  rox_imask_del(Combination);
}
```

In the figure 4.1, you can see the example results. In order of appearance, the masks are: Ellipse, User, User-Ellipse combination and Polygon.

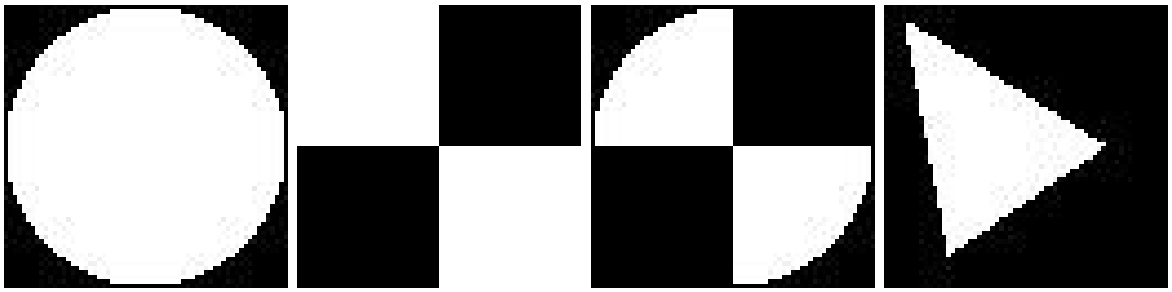

Figure 4.1: Examples of setting masks.

4.3 Image Display

The image display module contains structures and methods for displaing results on images and saving on disk.

4.3.1 The object `Rox_Image_Display`

An image display object can be declared using the pointer to a `Rox_Image_Display_Struct` structure.

```
typedef struct Rox_Image_Display_Struct* Rox_Image_Display;
```

4.3.2 Creating/Deleting a `Rox_Image_Display`

Functions are provided to allocate and deallocate a `Rox_Image_Display` object :

```
Rox_Error rox_image_display_new (Rox_Image_Display *image,
   Rox_Uint cols, Rox_Uint rows);
```

The function allocates memory for data depending parameters 'cols' and 'rows' which correspond to image size.

```
Rox_Error rox_image_display_del (Rox_Image_Display *image);
```

The function deallocates memory for a `Rox_Image_Display` object. It is necessary to call this function when the structure is not used anymore.

4.3.3 Main functions related to `Rox_Image_Display`

An image can be read from a PGM or PPM file if the size is compatible using the following functions:

```
Rox_Error rox_image_display_readpgm (Rox_Image_Display image,
   const char *filename);

Rox_Error rox_image_display_readppm (Rox_Image_Display image,
   const char *filename);
```

After drawing the results, the image can be saved on disk using the following functions:

```
Rox_Error rox_image_display_savepgm (const char *filename,
   Rox_Image_Display image);

Rox_Error rox_image_display_saveppm (const char *filename,
   Rox_Image_Display image);
```

The results of odometry computation can be drwan using the following functions:

```
Rox_Error rox_image_display_draw_projection_model_single_plane (
   Rox_Image_Display image, Rox_Matrix calib, Rox_MatSE3 pose,
   Rox_Model_Single_Plane model, Rox_Uint color);

Rox_Error rox_image_display_draw_projection_model_multi_plane (
   Rox_Image_Display image, Rox_Matrix calib, Rox_MatSE3 pose,
   Rox_Model_Multi_Plane model, Rox_Uint color);
```

4.4 Identification

The identification module contains structures and methods for the identification of model images and contains the following sub-modules:

Photoframe identification : Module with structures and methods for texture identification using a photoframe around the model image.

Texture identification : Module with structures and methods for texture identification. This module allows the user to define textured model images. The identification at run-time is generally slower than the database identification but can be accelerated using a photoframe around the model image.

Database identification : Module with structures and methods for model identification in a database.The database identification module needs the generation of a database from an image model. This offline generation step is time consuming but can be performed once and for all. At runtime, the database identification is faster than the texture identification since many computations have already been done offline.

4.4.1 Photoframe identification

The photoframe identification module performs the identification of textured model images with a black frame around the texture as shown in Figure 4.2.

Figure 4.2: Examples of model that can be identified with the photoframe identification module.

The photoframe image model identification is faster than the identification of phototexture image model since the identification take benefit of the detection of the black frame around the texture. On the other hand, the black frame shall be fully visible in the image.

4.4.1.1 The `Rox_Ident_Photoframe_SE3` object

A `Rox_Ident_Photoframe_SE3` object is a pointer to the opaque structure `Rox_Ident_Photoframe_SE3_Stru`
:

```
typedef struct Rox_Ident_Photoframe_SE3_Struct *
   Rox_Ident_Photoframe_SE3
```

4.4.1.2 Creating/Deleting a `Rox_Ident_Photoframe_SE3`

The rox_ident_photoframe object shall be created before any call to other functions using it :

```
Rox_Error rox_ident_photoframe_new(Rox_Ident_Photoframe *
   ident_photoframe)
```

This function creates a new empty database object.

```
Rox_Error rox_ident_photoframe_del(Rox_Ident_Photoframe *
   ident_photoframe)
```

4.4.1.3 Main functions related to `Rox_Ident_Photoframe_SE3`

A photoframe is a planar object specifically designed to be identified when filmed with a camera. It is a composition of an image template, a thick black border and a white background outside of the border. Figure 1 explains the common scheme of a photoframe. The image inside the black borders is the *template*. This template is the object to identify, and which will be used to track the object after detection if needed. This template size is $[image\ rows, image\ cols]$. Values iheight and iwidth shall be equal to 128 pixels.

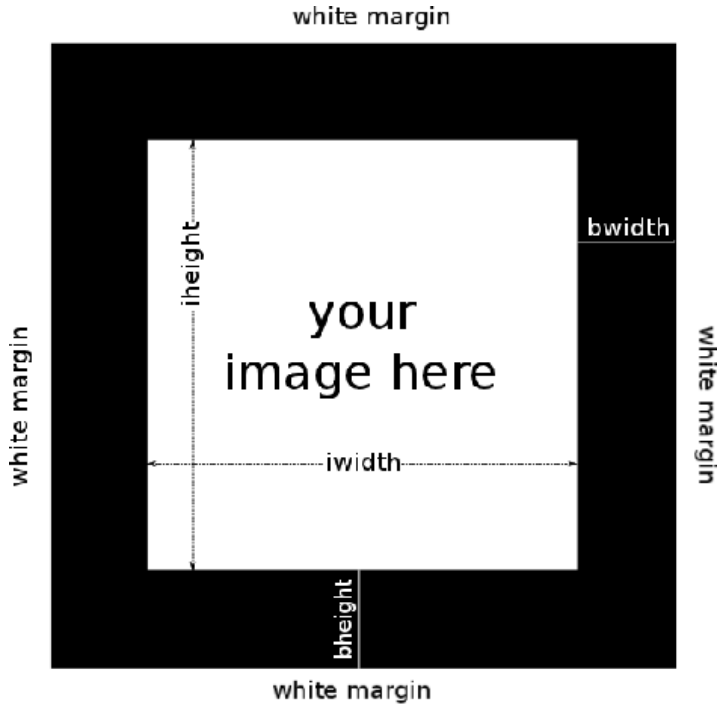

Figure 4.3: A common photoframe scheme

The border size $[border\ rows, border\ cols]$ is choosed optimally to be visible in a large set of viewpoints while not being too large (i.e. not wasting image space). The following calculus is done to compute border size :

- border rows = $\frac{80}{512}$ image rows
- border cols = $\frac{80}{512}$ image cols

Be sure to respect those sizes when creating your photoframe or the identification performance will be severely degraded.

To create a photoframe, choose a template image which is compatible with **OPENROX** (non uniform, with enough texture). Store this template in a PGM image file. With an image editing software (E.g. : photoshop, gimp) draw the borders in black color (See figure 1). Print this image on a white paper sheet. Because of printers approximation, it is not possible to know what will be the exact size (in meters) of this print. Take a ruler to measure the width and height of the image template *inside* the black borders. These real sizes are only needed if used to compute odometry after identification. Repeat this operation for each photoframe you need.

The user can go through the following workflow:

1. An identification structure is created (see section 4.4.1.2).

2. Template images are added to the identification module using the following function:

   ```
   Rox_Error       rox_ident_photoframe_se3_addframe (
      Rox_Ident_PhotoFrame_SE3 obj, Rox_Image model, Rox_Real
      width_meter, Rox_Real height_meter, Rox_Uint border_width
      );
   ```

   Adds a new template to a photoframe identification structure. Shall be called before identification. First parameter is a pointer created with rox_ident_new, second parameter is the template image to add (without black borders) with a size of 128x128 pixels. Please note that one photoframe will be identified only once in one image. If several instances of the same photoframe are in the processed image, only the first found will be considered. Note that the identifier of the added photoframe will be the number of times this function has already been called.

3. The user grabs an image from a stream (e.g. a camera) and gives it to photoframe identification. The identification is done and the number of identified templates are stored in the `Rox_Ident_PhotoFrame_SE3` object:

   ```
   Rox_Error rox_ident_photoframe_se3_make (
      Rox_Ident_PhotoFrame_SE3 obj, Rox_Camera camera);
   ```

   Using the ident structure created with rox_ident_new and filled with templates using rox_ident_add_photoframe, tries to detect photoframes in the given image. This image may be given by a camera for example. Returns the number of identified photoframes in the current image.

4. For each template, the user can check if it has been identified using the following function:

```
Rox_Error rox_ident_photoframe_se3_getresult (Rox_Uint *
   is_identified, Rox_MatSE3 pose, Rox_Ident_PhotoFrame_SE3
   obj, Rox_Uint id);
```

Used after a call to rox_ident_photoframe. Checks if a given photoframe (using its id) has been detected or not. ident is a pointer to a photoframe identification structure created with rox_ident_new. photoframe_id is a number between 0 and the number of added frames. Returns Rox_True if the specific photoframe is detected, Rox_False otherwise.

Used after a call to rox_ident_photoframe and rox_ident_get_identified. If a photoframe with id has been identified, returns the 2D transformation which represents the template image (without the borders) in the current image. 2D transformation is not defined (and may contain random value) if the marker is not identified.

5. Once not used anymore, the identification structure shall be deleted (see section 4.4.1.2).

4.4.2 Texture identification

The texture identification module performs the identification of textured model images as shown in Figure 4.4.

Figure 4.4: Example of image model that can be identified with the texture identification module.

4.4.2.1 The Rox_Ident_Texture_SE3 object

A Rox_Ident_Texture_SE3 object is a pointer to the opaque structure Rox_Ident_Texture_SE3_Struct :

```
typedef struct Rox_Ident_Texture_SE3_Struct *
   Rox_Ident_Texture_SE3
```

4.4.2.2 Creating/Deleting a Rox_Ident_Texture_SE3

Functions are provided to allocate and deallocate a Rox_Ident_Texture_SE3 object :

```
Rox_Error rox_ident_texture_se3_new (Rox_Ident_Texture_SE3 *
   ident_texture);
```

The function creates a new identification structure. Shall be called before any other function of this module. Returns a pointer to the Rox_Ident_Texture object.

```
Rox_Error       rox_ident_texture_se3_del (Rox_Ident_Texture_SE3
    *ident_texture);
```

Deletes an identification structure. First parameter is a pointer created with rox_ident_texture_new. Shall be called to free up memory when user does not need identification anymore.

4.4.2.3 Main functions related to Rox_Ident_Texture_SE3

A model to be identified can be set using the following function:

```
Rox_Error rox_ident_texture_se3_set_model (Rox_Ident_Texture_SE3
    ident_texture, Rox_Model_Single_Plane model);
```

A function is available to identify a given model in a camera object: The second function input the camera containing the current image in which to identify the texture:

```
Rox_Error rox_ident_texture_se3_make (Rox_MatSE3 pose,
   Rox_Ident_Texture_SE3 ident_texture, Rox_Camera camera);
```

4.4.3 Database identification

The computation of visual information (e.g. keypoints features) needed for image identification can be done offline and stored in a database. For each image model we would like to identify in the current image, a specific database item is created. Several items can also mixed together to allow the identification of multiple images.

Figure 4.5 illustrates the structure of the database identification module:

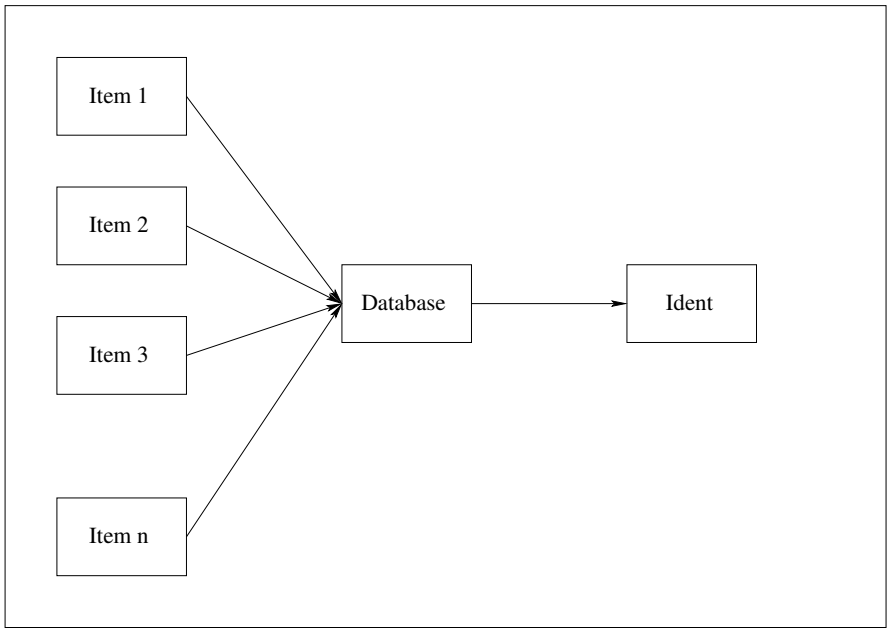

Figure 4.5: Structure of the database identification module.

4.4.3.1 The `Rox_Ident_Database_SE3` object

A `Rox_Ident_Database_SE3` object is a pointer to the opaque structure `Rox_Ident_Database_SE3_Struct` :

```
typedef struct Rox_Ident_Database_SE3_Struct *
   Rox_Ident_Database_SE3
```

4.4.3.2 Creating/Deleting a `Rox_Ident_Database_SE3`

One database item is associated to one template. A Rox_Ident_Database may be considered as a collection of such database items. The database identification module is able to identify several templates simultaneously. Current library is optimized to run on up to 150 simultaneous templates. User can ask the module to detect all possible templates in the same current image or to detect only the most visible template (among the templates in the database). Speed degrades with the number of templates to track simultaneously and quality degrades with the number of templates in the database. It is possible to create several Rox_Ident_Database objects in the same program.

The rox_ident_database object shall be created before any call to other functions using it :

```
Rox_Error rox_ident_database_se3_new (Rox_Ident_Database_SE3 *
   ident, Rox_Uint max_templates_simultaneous);
```

This function creates a new empty database object.

```
Rox_Error rox_ident_database_se3_del (Rox_Ident_Database_SE3 *
   ident);
```

This function deletes the object. Shall be called when you do not need this database object anymore.

4.4.3.3 Main functions related to `Rox_Ident_Database_SE3`

The following function shall be called to enable identification and prepare internal structures for optimized execution:

```
Rox_Error rox_ident_database_se3_set_database (
   Rox_Ident_Database_SE3 ident, Rox_Database db);
```

The following function execute identification process on the image contained in the camera object. Uses all the templates added and compiled before the call to this function. Once a template is found, an internal flag is set and it will be ignored in further calls to rox_ident_database_process.

```
Rox_Error rox_ident_database_se3_make (Rox_Ident_Database_SE3
   ident, Rox_Camera camera);
```

To get the identification result for a given template ID use the following function:

```
Rox_Error rox_ident_database_se3_getresult (Rox_Uint *
   is_identified, Rox_MatSE3 pose, Rox_Ident_Database_SE3 ident,
    Rox_Uint id);
```

4.4.3.4 Creation of database items

First, retrieve an image of your template without any perspective distortion. The biggest the image is, the better the detection will perform after database creation. However, if the image is too big the database file will be large and long to create. Ideally, the template size should be close to the image that will be viewed by the camera. The template image shall not contain large textureless regions and shall contain visual information everywhere. Please keep both files in a directory as they will be necessary for identification process.

The programmer can create its own database items by using the following function:

```
Rox_Error rox_database_item_learn_template (Rox_Database_Item
   item, Rox_Image template);
```

A database item can be saved on disk (use the extension ".rdi" for database items) using the following function:

```
Rox_Error rox_database_item_save (Rox_Char *filename,
   Rox_Database_Item item);
```

A database item can be loaded using the following function:

```
Rox_Error rox_database_item_load (Rox_Database_Item item,
   Rox_Char *filename);
```

4.4.3.5 Creation of a database

A database object shall be created using the following function:

```
Rox_Error rox_database_new (Rox_Database *db));
```

This function creates a new empty database object.

The following function deletes the object.

```
Rox_Error rox_database_del (Rox_Database *db);
```

Shall be called when you do not need this database object anymore.

A database item can be added to the database using the following function:

```
Rox_Error rox_database_add_item (Rox_Database database,
   Rox_Database_Item item, Rox_Real sizx, Rox_Real sizy);
```

After adding several items to the database, they can be fused in the database using the following function:

```
Rox_Error rox_database_compile (Rox_Database database);
```

The resulting database can be saved on disk (use the extension ".rdb" for databases) using the following function:

```
Rox_Error rox_database_save (char *filename, Rox_Database
   database);
```

and loaded from disk using the following function:

```
Rox_Error rox_database_load (Rox_Database db, char *filename);
```

For client/server applications the user can serialise a database using the following function:

```
Rox_Error rox_database_serialize (char *buffer, Rox_Database db)
   ;
```

and after sending it through the network, the user can deserialise it using the following function:

```
Rox_Error rox_database_deserialize (Rox_Database db, char *
   buffer);
```

4.4.3.6 Identification using database features directly

In some applications, like cloud client/server applications, it is useful to store and use directly database features for the identifications. The features are stored in a Rox_Database_Features object.

The programmer can use the following function to create database features:

```
Rox_Error rox_database_features_new (Rox_Database_Features *
   features);
```

The Rox_Database_Features object can be deleted using the following function:

```
Rox_Error rox_database_features_del (Rox_Database_Features *
   features);
```

The reference image can be therefore identified using the database features:

```
Rox_Error rox_ident_database_se3_make_features (
   Rox_Ident_Database_SE3 ident, Rox_Database_Features features,
    Rox_Matrix calib_camera);
```

In the first case, the programmer can also serialize/deserialize the information in the Rox_Database_Features object in order to be sent through a network socket:

```
Rox_Error rox_database_features_serialize (Rox_Char *buffer,
   Rox_Database_Features features);

Rox_Error rox_database_features_deserialize (
   Rox_Database_Features features, Rox_Char *buffer);
```

In the second case, the file containing the database features can be created using the following function:

```
Rox_Error rox_database_features_save (Rox_Char *filename,
   Rox_Database_Features features);
```

The file can be loaded using the following function:

```
Rox_Error rox_database_features_load (Rox_Database_Features
   features, Rox_Char *filename);
```

See the examples "rox_example_identification_database_cloud.c" for an example of use.

Chapter 5

Sensor Module

The sensor module contains structures and methods for sensor manipulation.

The sensor module contains the following sub-modules :

Camera : Module with structures and methods for camera sensors ;

Inertia : Module with structures and methods for inertia sensing ;

5.1 Camera

OPENROXassumes that images are acquired by a camera that respects a standard pin-hole model. With this model, straight lines in the 3D space will project to straight lines in the 2D image space.

In a pin-hole model, the relation between a point M with coordinates (X,Y,Z,1) in reference frame and its equivalent with homogeneous coordinates (u,v,1) in the camera frame can be written as follows :

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \propto \underbrace{\begin{bmatrix} f & s & c_u \\ 0 & fr & c_v \\ 0 & 0 & 1 \end{bmatrix}}_{\text{K}} \underbrace{\begin{bmatrix} & R_{3\times3} & & t_x \\ & & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{T}} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

The K matrix corresponds to camera intrinsic parameters where:

f : Focal lenght in pixels

r : Pixel ratio (equal to 1 for square pixels)

s : Skew (equal to 0 for square or rectangular pixels)

c_u : u-coordinate of the optical center projection in the image

c_v : v-coordinate of the optical center projection in the image

The intrinsic parameters are generally fixed by the hardware and should not change with time.

The procedure to calibrate the intrinsic parameters is described in the section 5.1.4

The T matrix corresponds to camera extrinsic parameters where:

R : Rotation matrix between reference and current position of the camera

t : Translation vector between reference and current position of the camera

The extrinsic parameters change depending on the camera localization and have to be computed for each image.

OPENROX provides functions and structures to handle these parameters. The intrinsic and extrinsic parameters are stored in `Rox_Camera` structure.

5.1.1 The `Rox_Camera` object

The structure `Rox_Camera_Struct` is designed to hold camera data: intrinsic and extrinsic parameters matrices and the image.

A camera object is opaque and can be accessed using the pointer `Rox_Camera` to a `Rox_Camera_Struct` structure.

```
typedef struct Rox_Camera_Struct* Rox_Camera;
```

5.1.2 Creating/Deleting a `Rox_Camera`

The library provides functions to create, initialize and delete `Rox_Camera` structure.

```
Rox_Error rox_camera_new (Rox_Camera *camera, Rox_Uint cols,
   Rox_Uint rows);
```

The `rox_camera_new` function allocates memory for data. In this case, the size of allocated memory depends on parameters 'cols' and 'rows' which correspond to image size.

The `rox_camera_new_readpgm` function allocates memory for data read from a pgm file ('filename'):

```
Rox_Error rox_camera_new_readpgm(Rox_Camera * camera, Rox_Char*
   filename);
```

If the user has a different file format (for example png or jpeg files), he can use its own library to load the image and then fill the Rox_Image object contained in the Rox_Camera object (see the Image module).

The `rox_camera_del` function deallocates memory for an `Rox_Camera` structure. It is necessary to call this function when the structure is not used anymore:

```
Rox_Error rox_camera_del(Rox_Camera * camera);
```

5.1.3 Main functions related to `Rox_Camera`

The main functions to use an `Rox_Camera` structure are :

rox_camera_readpgm : Sets image field by loading a pgm file.

rox_camera_get_pose : Returns the pointer to the pose matrix.

rox_camera_set_image : Set the image in the camera.

5.1.4 Camera Calibration

Camera calibration consists in finding the intrinsic parameters of the following matrix (see section 5.1):

$$\mathbf{K} = \begin{bmatrix} f & s & c_u \\ 0 & fr & c_v \\ 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

This tedious procedure is extremely easy with **OPENROX**.
The user can print or display on his screen any image with sufficient texture. After measuring the size of the rectangle the user can go through the following work-flow for camera calibration:

- create a new calibration object using a 2D model with known size:

  ```
  Rox_Error rox_texture_calibration_mono_perspective_new(
     Rox_Texture_Calibration_Mono_Perspective *calibration,
     Rox_Model_2D model);
  ```

  The size of the printed texture is given in meters.

- add a calibration image:

  ```
  Rox_Error rox_texture_calibration_mono_perspective_add_image
     (Rox_Texture_Calibration_Mono_Perspective calibration,
     Rox_Image image);
  ```

  The user shall add images (up to 20) taken with the camera and viewing the same texture from different point of views. An inclination between 30 and 45 degrees relative to the normal to the texture plane wil give good results. For optimal results, the texture in the current image should have almost the ssame size of the model image. The more images are added, the more camera intrinsic parameters ca be calibrated (see the “method” parameter below).

- make the camera calibration:

```
Rox_Error rox_texture_calibration_mono_perspective_make(
   Rox_Texture_Calibration_Mono_Perspective calibration,
   Rox_Uint method);
```

The user can choose the "method" parameter (between 1 and 5) to calibrate the following camera parameters:

1. method = 1 : calibrate the focal length f only (assuming the principal point c_u, c_v are at the center of the image). This method needs only 1 image in which the texture model is not observed fronto-parallel to the 3D plane.
2. method = 2 : calibrate the focal length f and aspect ration r (assuming the principal point c_u, c_v are at the center of the image). This method needs only 1 image in which the texture model is not observed fronto-parallel to the 3D plane.
3. method = 3 : calibrate the focal length f and the principal point c_u, c_v. This method needs at least 2 images in which the texture model is not observed fronto-parallel to the 3D plane.
4. method = 4 : calibrate the focal length f, aspect ration r and the principal point c_u, c_v. This method needs at least 2 images in which the texture model is not observed fronto-parallel to the 3D plane.
5. method = 5 : calibrate the focal length f, aspect ration r, skew s and the principal point c_u, c_v. This method needs at least 3 images in which the texture model is not observed fronto-parallel to the 3D plane.

Use the method number 4 when calibrating the camera for computing the odometry with **OPENROX**.

- get the camera calibration:

```
Rox_Error
   rox_texture_calibration_mono_perspective_get_intrinsics(
   Rox_Matrix intrinsics,
   Rox_Texture_Calibration_Mono_Perspective calibration);
```

The camera intrinsic parameters are written in a 3x3 matrix.

See the example "rox_example_camera_calibration.c" for an example of camera calibration.

5.2 Inertia

The inertia module, only used for the visual - inertial odometry, is composed by two sub-modules. The `Rox_Inertial` object contains the accelerometer (m/s^2), gyrometer (rad/s) and magnetometer calibrated measures and the acquisition timestamp. This data is required by the visual - inertial odometry and available through the `Rox_Inertial` object. Indeed, after setting the `Rox_Inertial` object with valid data, the user is able to make visual - inertial odometry using the functions described in section 9.3.1.1 and the `Rox_Inertial` object.

5.2.1 The `Rox_Inertial` object

A `Rox_Inertial` object can be defined using the pointer to a `Rox_Inertial_Structure`:

```
typedef struct Rox_Inertial_Struct* Rox_Inertial;
```

5.2.2 Creating/Deleting a `Rox_Inertial`

Functions are provided to allocate and deallocate a `Rox_Inertial` object :

```
Rox_Error rox_inertial_new (Rox_Inertial *inertial, const
   Rox_Float frequency);
```

The function allocates memory for the inertial object and returns a pointer on the newly created object.

```
Rox_Error rox_inertial_del (Rox_Inertial *inertial);
```

The function deallocates memory for a `Rox_Inertial` object. It is necessary to call this function when the object is not used anymore.

5.2.3 Main functions related to `Rox_Inertial`

The inertial measures can be set in the `Rox_Inertial` object using the following function :

```
Rox_Error rox_inertial_set_measure(Rox_Inertial inertial, const
   Rox_Real* A, const Rox_Real* W, const Rox_Real* M, Rox_Real
   timestamp);
```

This function set the accelerometer, gyrometer and magnetometer measures from `Rox_Real` buffers and sets the acquisition timestamp ;

If you need further information about inertial functions, please refer to the Programmer Manual.

Chapter 6

Model Module

The model module contains structures and methods for model manipulation.

The model module contains the following sub-modules :

Model Single Plane : Module with structures and methods for single plane models definition and use ;

Model Multi Plane : Module with structures and methods for multi plane models definition and use ;

6.1 Model Single Plane

6.1.1 The `Rox_Model_Single_Plane` object

A `Rox_Model_Single_Plane` object can be declared using the pointer to a `Rox_Model_Single_Plane_Struct` :

```
typedef struct Rox_Model_Single_Plane_Struct*
   Rox_Model_Single_Plane;
```

The structure is opaque to the user and can only be accessed through constructors, destructors and methods described in the following sections.

6.1.2 Creating/Deleting a `Rox_Model_Single_Plane`

Functions are provided to allocate, initialize and deallocate an `Rox_Model_Single_Plane` object :

```
Rox_Error rox_model_single_plane_new(Rox_Model_Single_Plane *
   model_single_plane, Rox_Image image, Rox_Real sizex, Rox_Real
    sizey);
```

The `rox_model_single_plane_new` function allocates memory for data.

```
Rox_Error rox_model_single_plane_del(Rox_Model_Single_Plane *
   model_single_plane);
```

The `rox_model_single_plane_del` function deallocates memory for an `Rox_Model_Single_Plane` structure. It is necessary to call this function when the structure is not used any more or before overwriting it.

6.1.3 Main functions related to `Rox_Model_Single_Plane`

The texture (i.e. an image) associated to the planar model can be set using the following function:

```
Rox_Error rox_model_single_plane_set_template(
   Rox_Model_Single_Plane model, Rox_Image image, Rox_Real sizex
   , Rox_Real sizey);
```

The real size of the planar target must be given.

6.2 Model Multi Plane

The present version of **OPENROX** allows the user to define simple 3D objects only. Currently, the 3D object is described as a set of 3D quadrilaterals (e.g. : the 6 faces of a cube or parts of the faces). Each quadrilateral is defined by an image texture and four 3D points associated to each corner of the texture (order : Top left -> Top right -> Bottom right -> Bottom left). The 3D vertices of all faces shall be defined in a common reference frame $\mathcal{F}_r$. Be sure to define your 3D coordinates as precisely as possible. It is up to the user to choose the adequate $\mathcal{F}_r$ for your vertices. The pose estimated by the odometry and returned by the SDK is the 3D transformation (rotation and translation) from this reference frame to the camera frame.

6.2.1 The `Rox_Model_Multi_Plane` object

A model 3D object is a pointer to the opaque structure `Rox_Model_Multi\_Plane_Struct` :

```
typedef struct Rox_Model_Multi\_Plane_Struct* Rox_Model_Multi\
   _Plane;
```

The structure is opaque to the user and can only be accessed through constructors, destructors and methods described in the following sections.

6.2.2 Creating/Deleting a Rox_Model_Multi_Plane

Functions are provided to allocate, initialize and deallocate an Rox_Model_Multi_Plane object :

```
Rox_Error rox_model_multi_plane_new(Rox_Model_Multi\_Plane *
   model_multi_plane);
```

The rox_model_multi_plane_new function allocates memory for data.

```
Rox_Error rox_model_multi_plane_del(Rox_Model_Multi\_Plane *
   model_multi_plane);
```

The rox_model_multi_plane_del function deallocates memory for an Rox_Model_Multi_Plane structure. It is necessary to call this function when the structure is not used any more or before overwriting it.

6.2.3 Main functions related to Rox_Model_Multi_Plane

The main function related to the Rox_Model_Multi_Plane is the function that allows to define an additional textured quadrilateral belonging to the 3D object:

```
Rox_Error rox_model_multi_plane_append_plane(
   Rox_Model_Multi_Plane model, Rox_Image image, Rox_Real
   vertices[4])
```

Chapter 7

Motion Detection Module

Visual motion detection is the process of determining the changes occurring in a sequence of images relative to a given reference image. If the camera is static, these changes are mainly due to moving objects. The localization of moving objects in the image is used in several applications such as robotics, active video-surveillance, and many others.

In **OPENROX**, the detection of a moving target is done inside an appropriate Region Of Interest (ROI) selected by the user. In the current version of **OPENROX** the image is supposed to be acquired by a static camera.

The Detection module contains the following sub-modules:

Detection Parameters : module with structures and methods for handling the detection parameters ;

Detection : module with structures and methods for motion detection ;

7.1 Detection Parameters

7.1.1 The Rox_Detection_Params object

The object `Rox_Detection_Params` contains two parameters that can be used to define the visual detection. The parameters are the following:

- Bandwidth : the approximate size of the object to be detected
- Sensitivity : the motion detection sensitivity

The detection parameters are stored in a `Rox_Detection_Params_Struct` structure. A `Rox_Detection_Params` object is opaque and can be accessed using the pointer to a `Rox_Detection_Params_Struct` structure:

```
typedef struct Rox_Detection_Params_Struct* Rox_Detection_Params
    ;
```

7.1.2 Creating/Deleting a Rox_Detection_Params object

OPENROX provides functions to create / delete the Rox_Detection_Params structure :

```
Rox_Detection_Params rox_detection_params_new(Rox_Void);
```

The `rox_detection_params_new` function allocates memory for the structure, sets parameters with default values (`bandwidth = (20x30)pixels`, `sensitivity = 1.0`) and returns a pointer on the newly created structure.

```
Rox_Void rox_detection_params_del(Rox_Detection_Params Params);
```

The `rox_detection_params_del` function deallocates memory for a parameter structure. It is necessary to call this function when the structure is not used anymore or before overwriting it.

7.1.3 Main functions related to the object Rox_Detection_Params

The motion detection algorithm has two important parameters set at the parameters structure initialization : 'bandwidth', 'sensitivity'. Functions are provided to set these parameters :

- `rox_detection_params_set_bandwidth` : Sets the bandwidth parameter which is the approximate size of the moving object to detect. If the bandwidth is small, parts of bigger objects will be detected separately. On the contrary, if the bandwidth is big, smaller objects may be grouped together.
- `rox_detection_params_set_sensitivity` : Sets sensitivity parameter. The higher the sensitivity the more objects will be detected but false detections may appear (due to lighting changes, noise, or other minor changes in the image).

Setting a parameter with a non valid value (for example -1) will leave it unchanged.

Finally, the function `rox_detection_params_set_license_path` allows the user to set the path of the license file.

7.2 Detection

7.2.1 The Rox_Detection object

A `Rox_Detection` object can be defined using the pointer to a `Rox_Detection_Structure`:

```
typedef struct Rox_Detection_Struct* Rox_Detection;
```

7.2.2 Creating/Deleting a Rox_Detection object

Functions are provided to allocate and deallocate a `Rox_Detection` structure :

```
Rox_Detection rox_detection_new(Rox_Detection_Params
   detection_params, Rox_Image image);
```

The `rox_detection_new` function allocates memory for the structure of the motion detection, according to the `detection_params` object and returns a pointer on the newly created structure. The input image corresponds to the model image (background) used for the motion detection.

```
Rox_Void rox_detection_del(Rox_Detection detection);
```

The `rox_detection_del` function deallocates memory for a `Rox_Detection` structure. It is necessary to call this function when the structure is not used anymore or before overwriting it. Remember to set `detection = NULL;` after deleting the object.

The `Rox_Detection` structure is opaque, i.e. its internal fields are hidden and cannot be accessed directly by the user. A consequence is that the user can only declare and manipulate pointer on this structure, and never the structure itself, as the size of the structure is unknown to the user.

In order to detect moving objects in several ROI, the user can declare and create several `Rox_Detection` structures and perform the algorithm on each of them.

7.2.3 The main functions related to the object Rox_Detection

Once the `Rox_Detection` is allocated and initialized, it is possible to:

- Set a different model image
- Set an image mask to avoid detection in specific parts of the ROI
- Set different parameters for each detection
- Detect moving objects

7.2.3.1 Setting a model image

The function `rox_detection_motion_set_model` allows the user to set a model image which may be different from the model image chosen at initialization. This can be useful if the appearance of the model has changed. For example the detection may start in the morning and in the afternoon the lighting conditions have changed so much that it is preferable to renew the model image.

7.2.3.2 Setting an image mask

The following functions allow the user to set an image mask of any shape. This mask can be used to avoid detection in specific parts of the ROI.

7.2.3.2.1 Setting an image mask from windows

The function `rox_detection_set_imask_window` shall be used to select a rectangular region defined by a window.

The function `rox_detection_motion_set_imask_window_list` shall be used to select several rectangular regions defined by a window list.

7.2.3.2.2 Setting an user defined image mask

The function `rox_detection_set_imask` shall be used for defining such regions (for example a disc).
Further information about image masks can be found in section 4.2. Detection can then be performed with `rox_detection_make` function.

7.2.3.3 Setting detection parameters

The motion detection parameters (see section 7.1) can be modified for each image using the following functions:

`rox_detection_motion_set_bandwidth`
`rox_detection_motion_set_sensitivity`

7.2.3.4 Performing motion detection

To perform the detection we call the `rox_detection_make` function with the `Rox_Detection` structure and the image used to perform the detection as parameters. This function shall be called for each image of the sequence in which the user would like to detect a moving object.

The output of the detection is a list of windows defining a bounding box around the moving object.

See the example "rox_example_detection.tex" for an example of use.

Chapter 8

Tracking Module

Visual tracking is the process of determining the position of a target in a sequence of images. The localization of the target in the image can be useful in several applications such as robotics, video-surveillance, and many others.

In **OPENROX**, the selection of the target can be done by the user manually with a selection of an appropriate Region Of Interest (ROI) in the reference image. If the user has already defined such an ROI, **OPENROX** allows to identify a given image model in the current image. The identification module is also used to re-identify the target if the tracking is lost (for example if the ROI is occluded or out of the image).

The Tracking module contains the following sub-modules:

Tracking Parameters : module with structures and methods for handling the tracking parameters ;

Tracking : module with structures and methods for tracking 2D objects ;

8.1 Tracking Parameters

8.1.1 The Rox_Tracking_Params object

The object `Rox_Tracking_Params` contains several parameters that can be used to define the visual tracking. The basic shape for an area of interest is a rectangle. However, the user can also define a region of interest inside a polygon or create a customized area with image masks. The most important parameters are the following:

- Iteration number
- Tracking precision
- Prediction area
- Use of Kalman filter

- Score threshold

The tracking parameters are stored in a `Rox_Tracking_Params_Struct` structure. A `Rox_Tracking_Params` object is opaque and can be accessed using the pointer to a `Rox_Tracking_Params_Struct` structure:

```
typedef struct Rox_Tracking_Params_Struct* Rox_Tracking_Params;
```

8.1.2 Creating/Deleting a Rox_Tracking_Params object

OPENROX provides functions to create / delete the Rox_Tracking_Params structure :

```
Rox_Tracking_Params rox_tracking_params_new(Rox_Void);
```

The `rox_tracking_params_new` function allocates memory for the structure, sets parameters with default values (`miter = 10`, `mprec = 0`, `mpred = 16`, `score_thresh = 0.89`, `kalman = ROX_TRUE`, `auto_update = ROX_FALSE`, `light = ROX_FALSE`) and returns a pointer on the newly created structure.

```
Rox_Tracking_Params rox_tracking_params_new_readfile (Rox_File
   filename);
```

The `rox_tracking_params_new_readfile` function allocates memory for the structure, reads the 'filename' `Rox_File` and sets parameters with read values. A pointer on the newly created patch is returned by this function.

```
Rox_Void rox_tracking_params_del(Rox_Tracking_Params Params);
```

The `rox_tracking_params_del` function deallocates memory for a parameter structure. It is necessary to call this function when the structure is not used anymore or before overwriting it.

8.1.3 The main functions related to the object Rox_Tracking_Params

The tracking algorithm has four important parameters set at the parameters structure initialization : 'miter', 'mprec', 'mpred', 'mtime'. Functions are provided to set these parameters :

- `rox_tracking_params_set_miter` : Sets miter parameter ;
- `rox_tracking_params_set_mprec` : Sets mprec parameter ;
- `rox_tracking_params_set_mpred` : Sets mpred parameter ;

Setting a parameter with a strictly negative value (typically -1) will leave it unchanged.

It is not mandatory to use the same parameters during the whole sequence. For example, a part of the sequence may need a prediction phase and few iterations because of a known high translation, but we would rather use more iterations without prediction for the rest of the sequence. Thus, some parameters can also be changed in runtime using function described in 8.2.3.3.

8.1.3.1 Setting the parameters for tracking objects with changing shape or appearence

In some applications, as for example the visual tracking of targets with changing shape or appearence such as pedestrians, it is more stable to freeze some degrees of freedoms of the homography. The function `rox_tracking_params_set_tracking_model` allows the user to track such targets using the following transformation:

$$\mathbf{H} = \begin{bmatrix} homog[0] & 0 & homog[2] \\ 0 & homog[4] & homog[5] \\ 0 & 0 & homog[8] \end{bmatrix}$$

which provides not only the translation of the template in the image but also its anisotropic scaling (see Figure 8.1).

```
enum Rox_Tracking_Model_Enum {Rox_Tracking_Model_tu,
                              Rox_Tracking_Model_tv,
                              Rox_Tracking_Model_tu_tv_s,
                              Rox_Tracking_Model_tu_tv_su_sv,
                              Rox_Tracking_Model_tu_tv_s_r,
                              Rox_Tracking_Model_SL3 };
```

- `Rox_Tracking_Model_tu` with this model the target motion is supposed to be a translation along the u axis;
- `Rox_Tracking_Model_tv` with this model the target motion is supposed to be a translation along the v axis;
- `Rox_Tracking_Model_tu_tv_s` with this model the target motion is supposed to be a translation and an isotropic change of scale;
- `Rox_Tracking_Model_tu_tv_su_sv` with this model the target motion is supposed to be a translation and an anisotropic change of scale;
- `Rox_Tracking_Model_tu_tv_s_r` with this model the target motion is supposed to be a translation, an isotropic change of scale and a rotation around the optical axis;

- `Rox_Tracking_Model_SL3` with this model the target motion is supposed to be a homographic transformation;

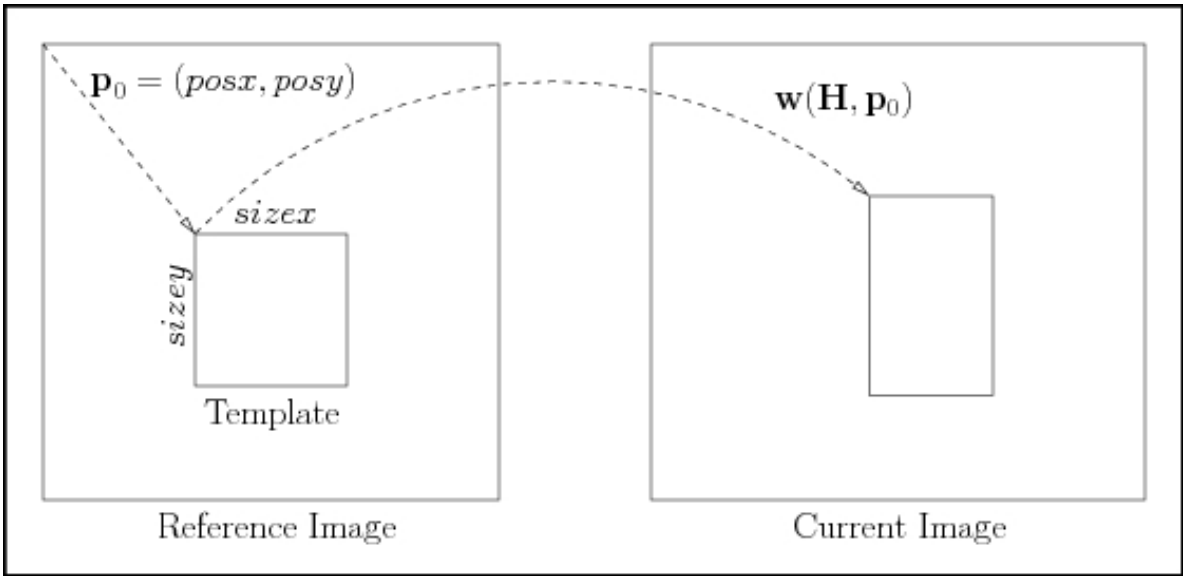

Figure 8.1: Translation and scaling of the reference template.

In several applications, when tracking deformable objects it is better to fix the model to `Rox_Tracking_Model_tu_tv_s` or `Rox_Tracking_Model_tu_tv_su_sv`.

8.2 Tracking

8.2.1 The Rox_Tracking object

A `Rox_Tracking` object can be defined using the pointer to a `Rox_Tracking_Structure`:

```
typedef struct Rox_Tracking_Struct* Rox_Tracking;
```

8.2.2 Creating/Deleting a Rox_Tracking object

Functions are provided to allocate and deallocate a `Rox_Tracking` structure :

```
Rox_Tracking rox_tracking_new(Rox_Tracking_Params P, Rox_Camera
   C);
```

The `rox_tracking_new` function allocates memory for the structure of the track object, according to the 'P' parameters and returns a pointer on the newly created structure.

```
Rox_Void rox_tracking_del(Rox_Tracking T);
```

The `rox_tracking_del` function deallocates memory for a `Rox_Tracking` structure. It is necessary to call this function when the structure is not used any more or before overwriting it.

The `Rox_Tracking` structure is opaque, i.e. its internal fields are hidden and cannot be accessed directly by the user. A consequence is that the user can only declare and manipulate pointer on this structure, and never the structure itself, as the size of the structure is unknown to the user.

In order to track individually several patches, the user can declare and create several `Rox_Tracking` structures and perform the algorithm on each of them. The patches may also belong to different planes.

8.2.3 The main functions related to the object Rox_Tracking

Once the `Rox_Tracking` is allocated and initialized, it is possible to:

- Predict large motion
- Perform tracking pf planar objects
- Change tracking parameters
- Set the mask to track a generic template
- Measure the quality of the tracking

8.2.3.1 Prediction

When a huge displacement occurs between two frames of a sequence, due either to a fast motion or to a low frame rate, the tracked object can get "lost".

In order to handle large displacement, a phase of prediction can be introduced prior to the tracking algorithm. It has to find the tracked object in a search window to determine a translation applied to the current homography matrix and then some iterations of the tracking algorithm can locally refine the displacement.

The search window is set by the 'mpred' parameter of the tracking structure, which defines the number of pixels around the current position of the tracked object.

As the prediction may be an expensive process, it is possible to skip it by setting the 'mpred' tracking parameter to zero.

8.2.3.2 Perform tracking of planar objects

To perform the tracking we call the `rox_tracking_make` function with the `Rox_Tracking` structure and the camera containing the image where to perform the tracking as parameters. This function shall be called for each image of the sequence.

The resulting homography can be retrieved using the function:

`rox_tracking_get_matsl3_copy`

which returns a pointer on a 3×3 matrix. This homography can be used for example to draw a black box around the tracked object in the image sequence, using the function:

`rox_image_draw_target_grays`

OPENROX provides two functions that allow to access the reference and current template warped in the coordinate system of the reference template using the obtained homography:

`rox_tracking_get_image_roi_ref` : Returns the reference ROI.

`rox_tracking_get_image_roi_cur` : Returns the current ROI warped in the coordinate system of the reference template using the obtained homography matrix.

8.2.3.3 Changing the Tracking Parameters during runtime

The tracking algorithm has four parameters set at the `Rox_Tracking` structure initialization : 'miter', 'mprec', 'msamp' and 'mpred'. However, it is not mandatory to use the same parameters during the whole sequence. For example, a part of the sequence may need a prediction phase and few iterations because of a known high translation, but we would rather use more iterations without prediction for the rest of the sequence.

The list of functions that allow the user to set tracking parameters are given in the Programmer Manual. They can change at any moment. Setting a parameter with a strictly negative value (typically -1) will leave it unchanged.

8.2.3.4 Setting the mask to track a generic template

OPENROX allows the user to perform the visual tracking of non-rectangular templates in the reference image. The function `rox_tracking_set_imask` shall be used for such templates.
Further information about mask can be found in section 4.2. Tracking can then be performed with `rox_tracking_make` function.

8.2.3.5 Measuring the quality of the visual tracking

In order to obtain the quality of the tracking result, the function `rox_tracking_get_score` uses the Zero-mean Normalized Cross-Correlation (ZNCC) between the reference template and the current image warped in the coordinate system of the reference template.

If we denote by I_r and I_c the vectors containing the intensity values of the reference template and of the current warped image and by $\overline{I_r}$ and $\overline{I_c}$ the means of the vectors I_r and I_c, the ZNCC can be written as follows:

$$ZNCC(I_r, I_c) = \frac{(I_r - \overline{I_r}) \cdot (I_c - \overline{I_c})}{||I_r - \overline{I_r}|| \times ||I_c - \overline{I_c}||}$$

The value of the ZNCC is between `-1.0` and `1.0`. When the reference template and the current template are identical, the ZNCC is equal to `1.0`. When the reference template and the current template are completely different, the ZNCC is close to `-1.0`. Therefore, the ZNCC is a very good measure that provides the quality of the visual tracking. When the reference template is tracked correctly, the returned value of the ZNCC is close to `1.0` (or bigger than `0.75` for example).

The score s used by **OPENROX** is normalized between 0 and 1 as follows:

$$s = (ZNCC + 1.0)/2.0$$

See the example "rox_example_tracking.tex" for an example of use.

Chapter 9

Odometry Module

Visual odometry is the process of determining the position and orientation (the pose) of a camera relative to a given target by analyzing a sequence of images. Contrarily to object tracking in image coordinates, visual odometry needs calibrated cameras. It implies the knowledge of camera intrinsic parameters. Consequently, the `Rox_Camera` object shall be fed with the correct camera intrinsic parameters. Incorrect camera intrinsic parameters will produce an incorrect localization.

The localization of the camera in the environment can be useful in several applications such as robotics, augmented reality, and many others.

The Odometry module contains the following sub-modules:

Odometry Single Plane : module with structures and methods for camera localization relative to a single plane

Odometry Multi Planes : module with structures and methods for camera localization relative to a multi planes

Odometry Inertial Observer : module with structures and methods for camera localization with inertial observer

9.1 Visual odometry observing a single plane

9.1.1 Odometry parameters

9.1.1.1 The `Rox_Odometry_Params` object

A `Rox_Odometry_Single_Plane_Params` object can be defined using the pointer to a `Rox_Tracking_Params_Structure`:

```
typedef struct Rox_Tracking_Params_Struct*
   Rox_Odometry_Single_Plane_Params;
```

The structure is opaque to the users.

9.1.1.2 Creating/Deleting a Rox_Odometry_Params

Functions are provided to allocate and deallocate a Rox_Odometry_Single_Plane_Params object :

```
Rox_Error rox_odometry_single_plane_params_new (
   Rox_Odometry_Single_Plane_Params *params);
```

The rox_odometry_single_plane_params_new function allocates memory for the odometry object and returns a pointer on the newly created object.

```
Rox_Error rox_odometry_single_plane_params_del (
   Rox_Odometry_Single_Plane_Params *params);
```

The function deallocates memory for a Rox_Odometry_Single_Plane_Params object. It is necessary to call this function when the object is not used anymore.

9.1.1.3 Main functions related to Rox_Odometry_Params

The camera localization is done relative to a known object. It is assumed that a rectangular planar object with size (sizx, sizy) is observed. In this case it is possible to know the pose of the camera relative to the planar object (the object frame is centered on the rectangular object, see figure 9.1) and also the displacement of the camera between two images.

9.1.2 Visual odometry Single Plane

9.1.2.1 The Rox_Odometry_Single_Plane object

A Rox_Odometry_Single_Plane is a pointer to the opaque structure Rox_Odometry_Single_Plane_Structur
:

```
typedef struct Rox_Odometry_Single_Plane_Struct *
   Rox_Odometry_Single_Plane;
```

9.1.2.2 Creating/Deleting a Rox_Odometry_Single_Plane

Functions are provided to allocate and deallocate a Rox_Odometry_Single_Plane object :

```
Rox_Error rox_odometry_single_plane_new (
   Rox_Odometry_Single_Plane *odometry,
   Rox_Odometry_Single_Plane_Params params,
   Rox_Model_Single_Plane model);
```

The function allocates memory for the odometry object, according to the 'model' and 'params' parameters and returns a pointer on the newly created object.

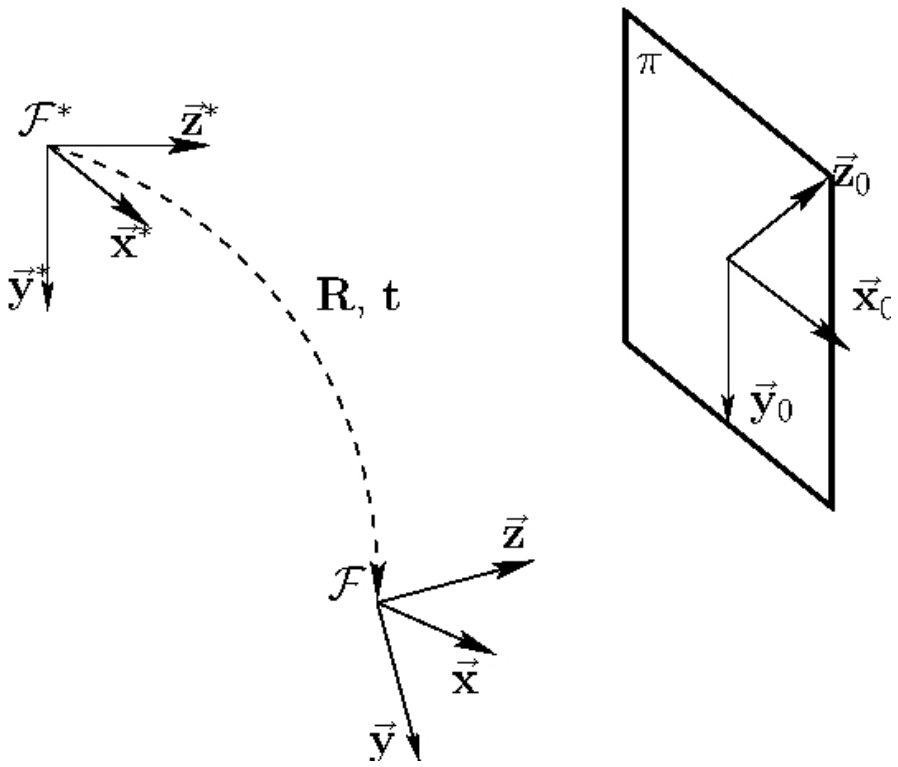

Figure 9.1: Localization relative to a planar object with known size.

```
Rox_Error rox_odometry_single_plane_del (
   Rox_Odometry_Single_Plane *odometry);
```

The function deallocates memory for a `Rox_Odometry_Single_Plane` object. It is necessary to call this function when the object is not used anymore.

9.1.2.3 Main functions related to `Rox_Odometry_Single_Plane`

The main functions to manipulate a `Rox_Odometry` object are :

rox_odometry_single_plane_make : Performs the visual odometry when the displacement of the target is not too large ;

rox_odometry_single_plane_get_pose : Returns the pose matrix ;

rox_odometry_single_plane_set_pose : Set the pose matrix ;

rox_odometry_single_plane_get_score : Returns the quality score of visual odometry ;

Please refer to the Programmer Manual for further information about visual odometry functions,

9.1.2.4 Target identification

When the displacement of the target is very large in the image, it is sometimes necessary to search the target in the whole image. The same problem can arise when the target is lost, occluded or out of the image and comes back in the camera field of view. **OPENROX** allows the user to identify the target by choosing several identification methods. The methods are detailed in section 4.4. See the example "rox_example_odometry_single_plane.c" for odometry with identification using a textured image and "rox_example_odometry_single_plane_databas for odometry with identification using a database of textured images.

If the user needs a very robust identification we recommend to use a photoframe around the target. Section 4.4.1 describes how to build photoframes and use them for target identification. See the example "rox_example_odometry_single_plane_photoframe.c" for an example of use with the odometry.

9.2 Visual odometry observing multi planes

9.2.1 Visual Odometry Multi Plane

The odometry module enables tracking of a textured convex polyhedron of n faces whose real dimensions are known. The estimation of the camera pose is obtained whatever

the part of the object the camera is looking at. The object is considered as a whole and the module uses all visible information to get a precise and stable localization of the object. The simplest example of a convex polyhedron is a cube (6 faces). The object to track using this module shall be textured on all the faces. Constraints related to texture are similar to those of 2D planar odometry.

9.2.1.1 The `Rox_Odometry_Multi_Plane` object

A `Rox_Odometry_Multi_Plane` object is a pointer to the `Rox_Odometry_Multi_Plane_Struct` structure:

```
typedef struct Rox_Odometry_Multi_Plane_Struct*
   Rox_Odometry_Multi_Plane;
```

The structure is opaque to the users.

9.2.1.2 Creating/Deleting a `Rox_Odometry_Multi_Plane`

Functions are provided to allocate and deallocate a `Rox_Odometry_Multi_Plane` object :

```
Rox_Odometry_Multi_Plane rox_odometry_multi_plane_new(
   Rox_Odometry_Multi_Plane_Params params, Rox_Model_Multi_plane
    model);
```

The function allocates memory for the odometry object, according to the `model` and `params` parameters and returns a pointer on the newly created object.

A 3D object model shall be passed to the odometry module for an appropriate odometry process (see section 6.2). The precision of the 3D vertices passed as input is the key to the accuracy of the odometry results. 3D vertices are used to simulate the object image projection. If the vertices are not well defined, the alignment of the real object and the virtual object is not possible. Scale of the vertices is not important for the visual odometry. However the output pose varies with the input scale.

Carefully choosing textures is very important for a robust odometry. The picture captured for each quadrilateral shall be as much parallel as possible relative to the face. Blur, noise and other lighting artefacts shall be avoided. Textures shall be as big as possible (e.g. a 512*512 texture per quadrilateral). If possible, use the same device for capturing the texture and for the odometry process, in order to minimize differences between images.

```
Rox_Void rox_odometry_multi_plane_del(Rox_Odometry_Multi_Plane
   odometry);
```

The `rox_odometry_multi_plane_del` function deallocates memory for a `Rox_Odometry_Multi_Plane` object. It is necessary to call this function when the object is not used anymore.

9.2.1.3 Main functions related to `Rox_Odometry_Multi_Plane`

The main functions to manipulate a `Rox_Odometry_Multi_Plane` object are :

rox_odometry_multi_plane_make : Performs the visual odometry and update the camera pose ;

rox_odometry_multi_plane_get_pose : Returns the pose matrix ;

rox_odometry_multi_plane_set_pose : Set the pose matrix ;

Please refer to the Programmer Manual for further information about visual odometry functions. See the example "rox_example_odometry_multi_plane.c" for an example of use.

9.3 Visual odometry with an inertial observer

9.3.1 Visual and Inertial Odometry

When the displacement between two successive images is too large, the visual odometry may fail. Consequently, it is necessary to identify the target to get the camera pose relative to the known model. Using an inertial sensor allows us to get a high rate prediction of the target position in the image even if the displacement is very important. The visual odometry may be successful as long as the inertial prediction is precise enough. As the inertial prediction does not depend on the actual scene viewed by the camera, it is possible to predict the target position even if it is out of the camera field of view. Consequently, when the target reappears in the camera field of view, no identification method will be necessary to make visual odometry if the inertial prediction is close enough from the real target position.
Note that to make an efficient visual - inertial odometry, the calibration pose between the two sensors shall be known by the user and the target frame shall coincide with the local tangent plane used as reference for the inertial measures.

9.3.1.1 Main functions related to `Rox_Odometry_Visual_Inertial`

The main functions to manipulate visual - inertial odometry are :

rox_odometry_visual_inertial_init_async_observer : Performs the detection of the given target and initializes the inertial observer using the odometry results. This function initializes an asynchronous observer implying that visual and inertial data have to be precisely dated. The dating shall be expressed in seconds and have the same time reference for both inertial and visual data (i.e using the UTC time) ;

rox_odometry_visual_inertial_init_sync_observer : Performs the detection of the given target and initializes the inertial observer using the odometry results. This function initializes a synchronous observer ;

rox_odometry_visual_inertial_make : Makes the visual - inertial odometry. Note that the inertial observer can be reinitialized by a visual detection method if the target has been lost during 10 successive images ;

rox_odometry_visual_inertial_get_matsl3_prediction_copy : Gets a copy of the initialization homography ;

If you need further information about visual and inertial odometry functions, please refer to the Programmer Manual. See the example "rox_example_odometry_visual_inertial.tex" for an example of use.