DsVision Library Documentation

Version: 0.1 Dic. 2010.
Author: Diego F. Asanza

OptiIdent is a c++ library useful for stereo vision. It includes utilities for image capture, camera calibration, image undistortion and rectification, point cloud visualization, descriptor calculation, etc. It was constructed as a wrapper around OpenCV and IVT libraries, in an attempt to simplify the OpenCV and IVT complex interface and to provide an development environment that simplifies the programming of computer vision applications for non-programming specialists.

It was developed as part of the Mobile Service Robotics Project (www.zafh-servicerobotik.de) at the Mannheim University of Applied Sciences (www.ds.hs-mannheim.de).

Installation.

This guide describes a step by step procedure in order to install the library in an Ubuntu-Linux system. Although there is nothing in the library that avoid it to be compiled in another Linux distribution, we have not yet tested this possibility and for that reason this installation guide assumes that you have a Ubuntu-Linux system, version 10.04 or greater. Furthermore, Ubuntu is one of the most widely used Linux distributions and is known for its ease of use. You can download Ubuntu free of charge at http://www.ubuntu.com.

We assume that you have basic programming skills as well as basic knowledge about Linux. We need a computer with at least usb ports (in order to connect the cameras) and a Ubuntu-Linux operating system on it (for help into installing Ubuntu goto http://www.ubuntu.com). On this machine we will proceed to install the library.

Installing development environment.

The first thing to do is install the dependencies. This library depends on some other libraries, as OpenCV and IVT for computer vision algorithms, and Video4Linux and libDC1394 for image capture among others. In order to install them, go to Applications/Accessories/Terminal as shown in Figure 1.

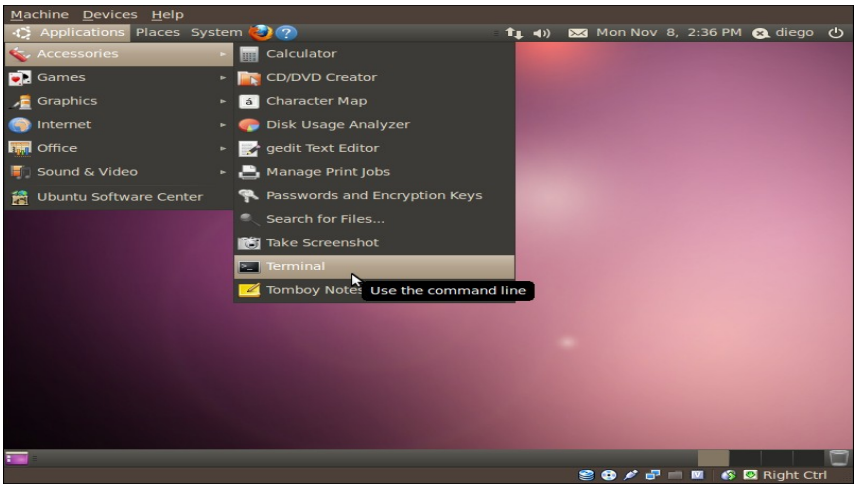

Illustration 1: How to Open a terminal window

That will open a terminal window as shown in figure 2.

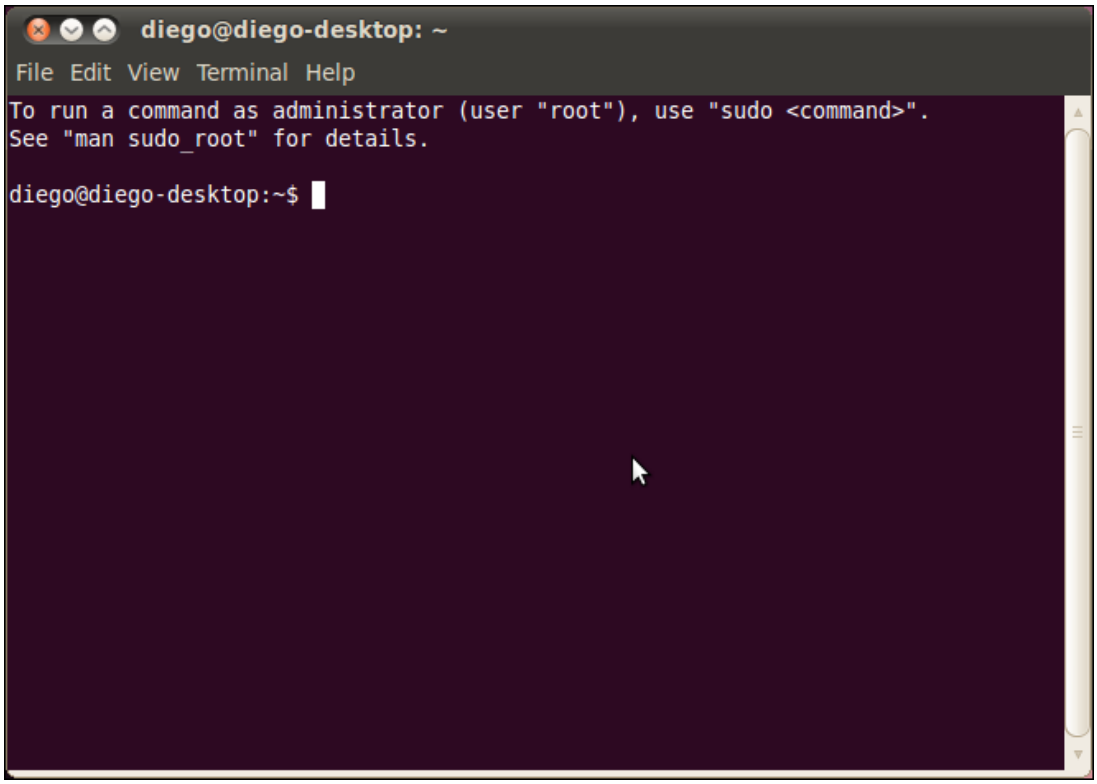

Illustration 2: Terminal Window

In this terminal window, write the following command:

apt-get install subversion cmake libv4l-dev libboost-all-dev libgl1-mesa-dev libglu1-mesa-dev libdc1394-22-dev libdc1394-utils libwxgtk2.8-dev

that will be install all the development environment and libraries needed to compile dsvision.

Download and install opencv.
OpenCv is a library for image processing and computer vision which can be freely downloaded from:

http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.1/OpenCV-2.1.0.tar.bz2/download

in order to download, configure, compile and install opencv, write the following commands on a terminal window:

wget http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.1/OpenCV-2.1.0.tar.bz2
tar -jxvf OpenCV-2.1.0.tar.bz2
cd OpenCV-2.1.0
cmake .
make
sudo make install

Installing the library.

The last version of this library is available at the subversion repositories in sourceforge:

svn co https://dsvision.svn.sourceforge.net/svnroot/dsvision dsvision

in order to use them, write the following on the terminal window:

```
svn co https://dsvision.svn.sourceforge.net/svnroot/dsvision dsvision
cd dsvision
cmake .
make
make install
```

This will create a directory called dsvision on the current path, download the library using subversion there, configure compile and install. The created directory will have following structure:

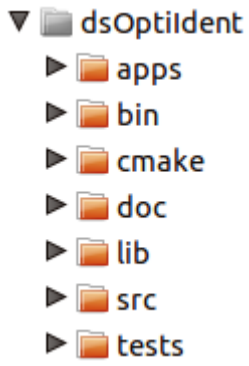

Illustration 3:
Directory Structure

Testing the installation.

Some demos were written using this library. They are available in the apps directory. The directory structure is depicted on figure 3. It contains following folders (in alphabetical order):

Apps

The library is delivered with some test applications that show how to use the library for image capture among other things. These can be found in the app directory on the dsOptiIdent folder.

Some of the application provided with the library are for camera control, object identification sift features extraction, data visualization.

Bin

The bin folder contains some binary applications to generate datastructures needed for some algorithms.

Cmake

The cmake folder contains scripts needed to run the cmake build system in order to compile the library.

Documentation

The doc folder contains some documentation about the library.

Library

The lib folder contains the binary version of the library.

Source

The source code is in the src directory. It is subdivided in following categories:

- 3rdparty: Libraries from and utilities from a 3rd party vendor which for convenience are better to integrate.
- Calibration: Classes and functions for mono and stereo camera calibration.
- Color: Classes and utilities for color processing.
- Stereo: Classes and utilities for stereo processing.
- Voctree: Vocabulary-tree based classificator.
- Communications: Classes and functions for basic data communication (just send and receive images over a network).
- Tools: Source code for the applications in the bin folder.
- Descriptors: Image descriptor definitions.
- Undistort : Image undistortion and 3D geometry.
- Segmentation: Image Segmentation.
- Utils: Common utilities.

Library Modules

Image Capture

Dsvision provides methods to read images from cameras and other video devices. To date, only two kinds of cameras are supported. IDC1394 or firewire cameras and V4L2 devices.

To list the cameras attached to the computer, the library provides the CameraBus class. It abstracts the bus connections and provides methods to search for installed cameras in the system and to initialize them.

```
/** This class defines a standard procedure for camera initialization.
 *  First the enumerate() method list all the cameras present on the
 *  Bus. The getcamera method initializes the camera given and returns
 *  a pointer to the initialized camera.
 */
class CameraBus {
public:
```

```
      virtual ~CameraBus(){};
      /** Lists the cameras present on Bus */
    virtual vector<string> Enumerate()=0;
    /** Initialized the camera given by index */
    virtual Camera* GetCamera(int index)=0;
    /** Initialize the camera given by name */
    virtual Camera* GetCamera(string name) = 0;
};
```

The camera class defines a common interface in order to control the camera. Common parameters that need be controlled and that are present almost on every camera on the market are gain, saturation, shutter speed, etc. This class provides a standard way to access them.

```
/** This clase defines a camera interface.
 * This type of cameras are very simple. They grab
 * just a RGB image and return it with IplImage format
 * to use with opencv. We can set a few parameters too, like
 * the image resolution to use, autogain, autoshutter, shutter value
 * gain value contrast saturation and hue. */
class Camera {
protected:
      friend class ds::CameraBus;
      // the user cannot construct an empty camera. Cameras are initialized
      // using the camerabus.
      string name;
      Camera(){};
      Camera(int index){};
      Camera(string name){this->name = name;};
public:
      /** Camera destructor */
      virtual ~Camera(){};
      /** Get a image from the camera */
      virtual IplImage* GrabFrame() = 0;
      /** Get the resolutions supported for this camera. */
      virtual vector<string> GetAvailableResolutions()=0;
      /** Set the desired format */
      virtual void SetResolution(string resolution)=0;
      /** Get the current resolution */
      virtual string GetResolution()=0;
      /** Get the image width */
      virtual int GetImageWidth()=0;
      virtual int GetImageHeight()=0;
      /** Just 1 or 3 channels supported (mono and RGB) */
      virtual int GetImageChannels()=0;
      virtual string GetCameraName()=0;
      virtual int GetCameraIndex()=0;
      virtual void SetAutogain(bool autogain=true) = 0;
      virtual void SetAutoshutter(bool autoshutter = true) = 0;
```

```
      virtual void SetBrightness(int value) = 0;
      virtual void SetContrast(int value) = 0;
      virtual void SetSaturation(int value) = 0;
      virtual void SetHue(int value) = 0;
      virtual void SetWhitebalance(int value) = 0;
      virtual void SetGain(int value) = 0;
      virtual void SetShutter(int value) = 0;
      virtual bool GetAutogain(bool autogain=true) = 0;
      virtual bool GetAutoshutter(bool autoshutter = true) = 0;
      virtual int GetBrightness() = 0;
      virtual int GetContrast() = 0;
      virtual int GetSaturation() = 0;
      virtual int GetHue() = 0;
      virtual int GetWhitebalance() = 0;
      virtual int GetGain() = 0;
      virtual int GetShutter() = 0;
};
```

In order to maintain things simple, this class does not attempt to access each available parameter of the camera, but just only those that are most used. A more complete control of the devices can be achieved using the derived classes DC1394Camera and V4LCamera.

The following program illustrates the use of these classes in order to get an image.

```
#include <cxcore.h>
#include <cv.h>
#include <dscamera.h>
#include <DC1394CameraBus.h>
#include <V4LCameraBus.h>
#include <iostream>
#include <vector>
#include <string>
#include <cxcore.h>
#include <cv.h>
#include <highgui.h>
using namespace ds;

int main()
{
      /// Placeholder for the initialized camera
      vector<Camera*> cams;
      // Bus instances
      DC1394CameraBus dc1394_bus; //firewire
      V4LCameraBus v4l_bus; // v4l devices
      /// Placeholders for the cameras ids.
      vector<string> fwcams, v4lcams;
      /// Enumerate the cameras present on bus
      fwcams = dc1394_bus.Enumerate();
      v4lcams = v4l_bus.Enumerate();
```

```
	/// by now, fwcams and v4lcams contains a list of devices
	/// present on the bus.
	/// Now try to initialize all the cameras found.
	for(int i = 0; i < v4l_cameras.size(); i++)
	{
		cout << v4l_cameras[i] << endl;
		///get the camera i and add to the placeholder
		cams.push_back(v4l_bus.GetCamera(i));
	}
	for(int i = 0; i < dc1394_cameras.size(); i++)
	{
		cout << dc1394_cameras[i] << endl;
		cams.push_back(dc1394_bus.GetCamera(i));
	}
	if(!cams.size())
	{
		cout << "No cameras found" << endl;
		exit(-1);
	}
	/// we can set the desired resolution (if supported on the camera)
	//cams[2]->SetResolution("1280x960");
	/// This loop grab images and show in windows.
	while(cvWaitKey(30)==-1)
	{
		for(int i = 0; i < cams.size(); i ++)
		{
			stringstream wnames;
			wnames << "camera" << i;
			/// GrabFrame returns a pointer to the
			/// captured image.
			IplImage* im = cams[i]->GrabFrame();
			cvShowImage(wnames.str().c_str(), im);
		}
	}
}
```

The program start lines:

```
	/// Placeholder for the initialized cameras
	vector<Camera*> cams;
	/// Placeholder for the firewire camera identifiers
	vector<string> dc1394_cameras;
	/// firewire bus instantiation
	DC1394CameraBus dc1394_bus;
	/// Enumerate the cameras present on bus
	dc1394_cameras = dc1394_bus.Enumerate();
	cout << "Listing Firewire Cameras:" << endl;
	/// Placeholder for the video for linux cameras
	vector<string> v4l_cameras;
	/// Video for linux camera bus
```

```
V4LCameraBus v4l_bus;
/// enumerate video for linux device
v4l_cameras = v4l_bus.Enumerate();
cout << "Listing V4L Cameras(usb): " << endl;
```

creates a Camera* vector which will be used as placeholder for the cameras found on the system. Additionaly a vector of string called dc1394_cameras and v4l_cameras are used to hold the identificators of the cameras found on the system. Cameras present are listed with the method enumerate of the CameraBus class.

The following lines:

```
for(int i = 0; i < v4l_cameras.size(); i++)
{
      cout << v4l_cameras[i] << endl;
      /// get the camera i and add to the placeholder
      cams.push_back(v4l_bus.GetCamera(i));
}
cout << "Listing firewire cameras" << endl;
for(int i = 0; i < dc1394_cameras.size(); i++)
{
      cout << dc1394_cameras[i] << endl;
      cams.push_back(dc1394_bus.GetCamera(i));
}
if(!cams.size())
{
      cout << "No cameras found" << endl;
      exit(-1);
}
```

go through the founded cameras and initializes them from the Bus with the method GetCamera(int). If no cameras are found, then a error will be printed an the program terminates.

The actual capture is performed in the method GrabFrame() of the class Camera. This method returns an IplImage pointer of the captured image. The underlaying data will be changed with the next call to GrabFrame().

Camera Calibration

In order to understand camera calibration we need first to know a little about the imaging process, or how a scene of the world is transformed in a set of bytes that forms the digital image.

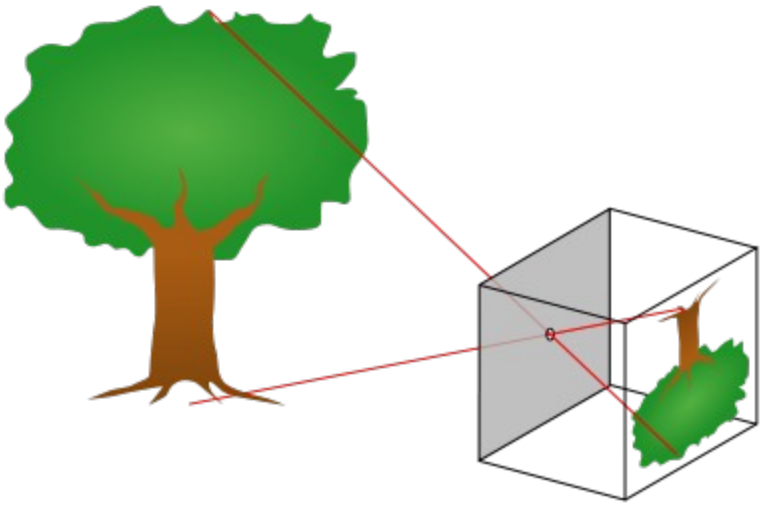

Illustration 4: Pinhole Camera(source: Wikipedia)

The most simple camera is the so called „Pinhole Camera“. It consist on a closed box with a little hole on one of the walls, as shown in the Figure 4. On this camera each individual light rays that come from an object in the 3-dimensional world goes throw the hole and are projected in the opposite wall forming an image in 2 dimensions. To say it another way, a “Pinhole Camera” is a device which projects points from the 3D Space into 2D Space. This can be described mathematically as:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Formula 1: Intrinsics Matrix

Where [X, Y, Z] are the coordinates of the point in 3D space, and [x, y] are the coordinates of the projected point in the image. The notation (x,y,w) is called “Homogeneous Coordinates” and is useful to do projective transformations.

The 3x3 matrix in the middle is called the intrinsics camera matrix, because it codifies information about the inner work of the camera, that is, the focal distance (fx,fy) which is the distance between the aperture and the image plane, and the coordinates (cx and cy) which are the coordinates of the center of the image plane in relation to the perpendicular from the aperture hole into the image plane.

A pinhole camera is easy to model and work mathematically, but in real applications it has a drawback: because the aperture of the hole is so small not so much light goes through, which means that we need a lot of light in order to get a good image. The obvious solution is to make a greater aperture, but because of the nature of light, this produces a burred image (Figure 5).

This blurred image can be corrected using lenses. Lenses help to focus the incoming light rays into individual points eliminating the blur, but lenses by themselves add some other kinds of distortion like chromatic aberration, pincushion aberration, etc.

Applications like 3D reconstruction relies on the assumption that an accurate geometrical model of the object can be extracted, and that is only possible if all the geometrical distortion induced on the camera system are corrected. Fortunately, geometrical distortions can be modeled and corrected, at least the most important.

According to the bibliography[insert reference] two are mainly the sources for image (geometric) distortion:

1. Spheric distortion which appears on spherical lenses because the light does not converge into a point as shown in figure 6, and
2. Tangential distortion, which is due irregularities in the assembly process of the camera which leads to a sensor which is not trully parallel to the lens system, as shown in figure 7.

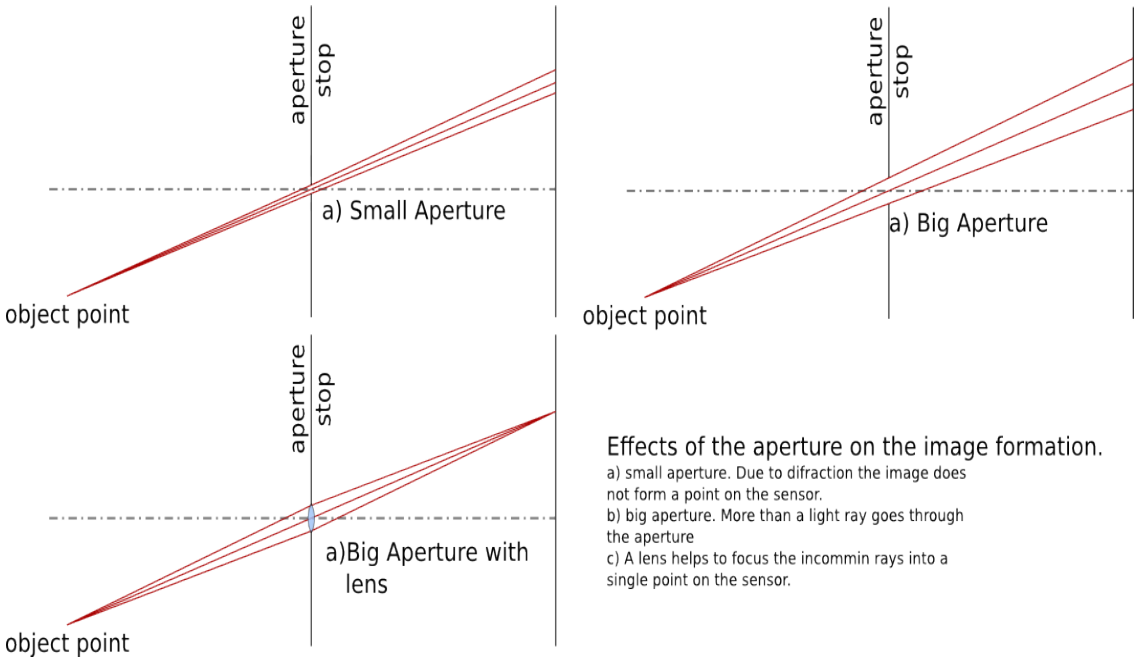

Illustration 5: Effects of the aperture on the image.

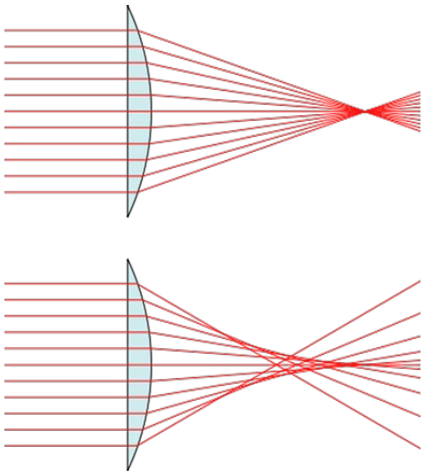

Illustration 6: Spheric Distortion

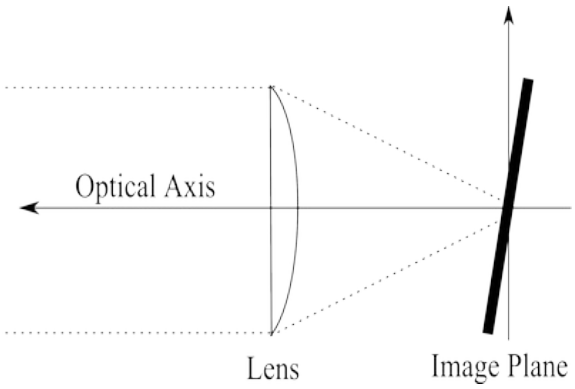

Illustration 7: Tangential Distortion

These distortions are modeled as follows:

$$\begin{bmatrix} x \\ y \end{bmatrix}_{new} = \begin{bmatrix} x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{bmatrix}$$

Formula 2: Radial Distortion

$$\begin{bmatrix} x \\ y \end{bmatrix}_{new} = \begin{bmatrix} x + 2p_1 y + p_2(r^2 + 2x^2) \\ y + p_1(r^2 + 2y^2) + 2p_2 x \end{bmatrix}$$

Formula 3: Tangential Distortion

The parameters k1, k2, k3, p1 and p2 are called distortion parameters and they and also the camera intrinsics matrix can be calculated using a procedure called camera resectioning or also camera calibration.

The camera calibration procedure try to solve the equations 1, 2 and 3 for known values of x,y and X,Y,Z obtained from images of a known pattern called calibration rig. (The dsvision library uses a checkerboard pattern (Fig. 8))

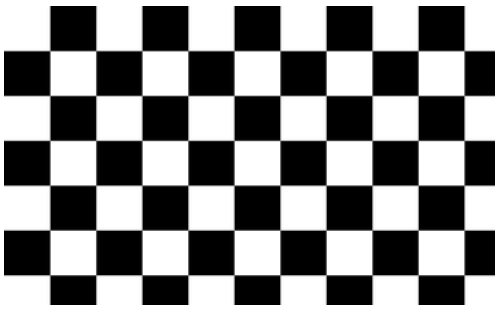

Illustration 8: Calibration rig

As result, we obtain the 3x3 camera intrinsics matrix and a 1x5 element vector (the distortion vector). They are called camera parameters.

Calibration on Stereo Cameras

With stereo cameras, things are pretty much the same with just two differences. First, a set of intrinsics and distortion parameters for each camera must be calculated, and in second place, a stereo pair has additional parameters which describes the relative position between cameras. In particular are important the translation vector, a 3d vector which relates the position of the left camera in relation to the right camera, and the rotation matrix, a 3x3 3D rotation matrix which relates the rotation of the left camera in relation to the right camera (Fig. 9).

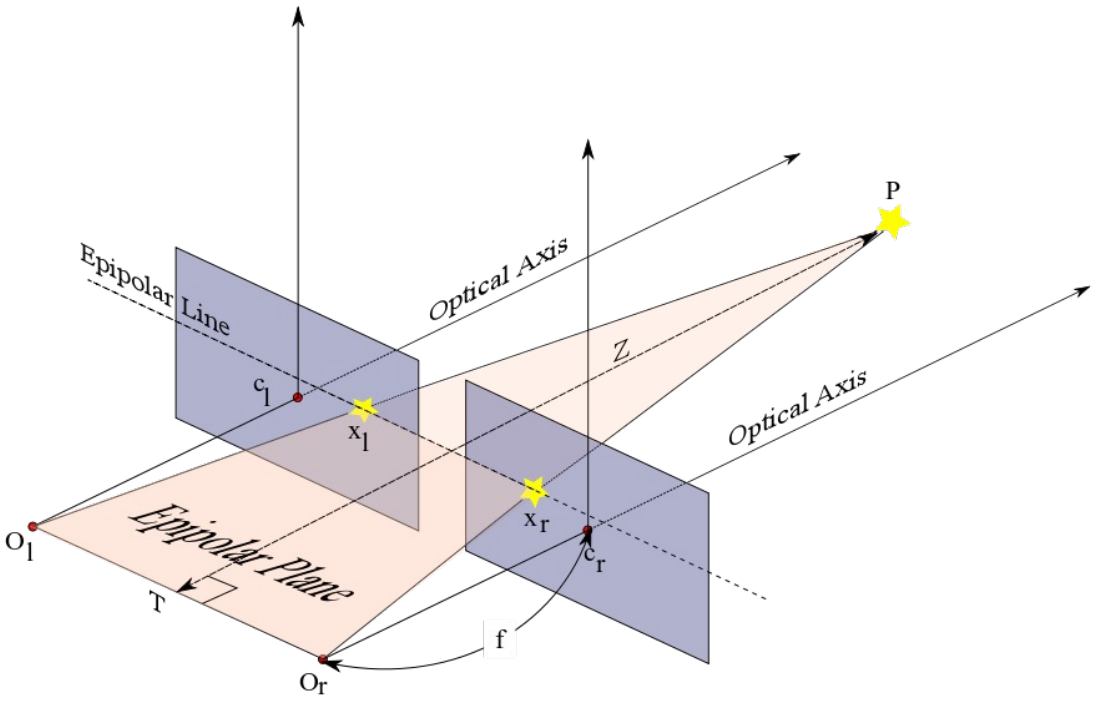

Illustration 9: Stereo Camera System. T is the translation vector. R is the rotation matrix between cameras. Xl and xr are the projections of the point P on the image planes.

On the figure 9, a point p is projected into the image planes as xl and xr, they are corresponding points. A Stereo Camera system satisfy the following property, called Epipolar Costraint:

$$\begin{bmatrix} x_l \\ y_l \end{bmatrix}^T \times [F] \times \begin{bmatrix} x_r \\ y_r \end{bmatrix} = 0$$

Formula 4: Fundamental Matrix

Where xl,yl are the coordinates of a point on the left camera and xr and yr are the coordinates of the corresponding point on the right camera. The matrix F is called "Fundamental Matrix" and describes the geometry of the stereo system. This matrix is related to the rotation and translation vector and is an important parameter to be calculated during calibration.

Class Camera parameters

This class abstracts the intrinsics and distortion parameters, it contains a CvMatrix with the intrinsics of the camera and a CvMatrix with the distortion vector:

```
class CameraParams
{
public:
      CameraParams();
      ~CameraParams();
      /** The camera intrinsics matrix.
       *  it is a 3x3 matrix with following information:
       *  [ fx,  0, cx ]
       *  [ 0 , fy, cy ]
       *  [ 0 ,  0,  1 ]
       *  where fx is the focal distance / pixel width
       *  and fy is the focal distance / pixel height. cx and cy are the
       *  coordinates of the center of the image sensor in pixels.
       */
      CvMat M; /// the camera matrix.
      /** The camera distortion vector.
       *  We use the Brown distortion model (see Brown DC (1966).
       * "Descentering distortion of lenses.".
       * Photogrammertric Engineering. 7. 444-462.). It models the
       * spheric distortion as:
       * x_corrected = x(1+k_1r^2 + k_2r^4 + k_3 r^6) and
       * y_corrected = y(1+k_1r^2+k_2r^4+k_3r^6)
       * while the tangential distortion is modeled as:
       * x_corrected = x+[2p_1y+p_2(r^2+2x^2)] and
       * y_corrected = y + [p_1(r^2+2y^2)+2p_2x]
       * D is the vector with the parameters: [k_1, k_2, p_1, p_2, k_3]
       */
      CvMat D; /// the distortion vector.
      /** Read the calibration parameters from a file
       * @param filename
       * @param camera_name a short string used to identify the camera
       */
      void read_from_file(string filename, string camera_name);
      /** Write the calibration parameters into a file */
      void save_to_file(string filename, string camera_name);
      /** The image size */
      CvSize imageSize;
      bool operator==(const CameraParams &params);
      bool operator!=(const CameraParams &params);
};
```

Table 1: Camera Parameters

The camera parameters class provides methods to save and load the camera parameters into or from a file. The file is saved as xml with the format shown in Table 2.

M is the Intrinsics Matrix
D is the distortion vector
S is the size of the image

The function write_to_file saves the camera parameters in a xml-file. Filename is the name to save and camera_name is the name of the camera (for identification purposes).

The function read_from_file reads the camera parameters from a xml file. The function raises an std::runtime_error if no file was found or if the file does not contains valid parameters or the given camera name.

```
<?xml version="1.0"?>
<opencv_storage>
<M type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>d</dt>
  <data>
    9.4716864244621524e+02 0. 3.4244995557175929e+02 0.
    9.4893500995732086e+02 2.2398901064337085e+02 0. 0. 1.</data></M>
<D type_id="opencv-matrix">
  <rows>1</rows>
  <cols>5</cols>
  <dt>d</dt>
  <data>
    -4.1614275561557418e-01 9.4780522032785020e-01
    1.6150871103398968e-03 -2.3470251725877544e-03
    -4.1034296438943114e+00</data></D>
<S type_id="opencv-matrix">
  <rows>2</rows>
  <cols>1</cols>
  <dt>i</dt>
  <data>
    640 480</data></S>
</opencv_storage>
```

Table 2: Sample xml camera parameters file

Class Stereo Camera parameters

this class abstract the stereo parameters in a similar way as the CameraParameters class. It is a placeholder for the parameters of the left and right cameras and also for the translation and rotation vectors between cameras.

Class calibration

As explained on the later sections, the calibration procedure tryes to calculate the intrinsics and distortion parameters of the camera and in the case of stereo cameras the rotation and translation matrix , using for this purpose some images of a calibrated pattern better know as calibration rig.

```
/** This class performs the calibration of a camera. We use a planar calibration
 *  rig
 */
class Calibration
{
public:
      /** This function returns the intrinsic parameters from a array of
       *  points that represents the coordinates of the corners in the calibration
       *  rig.
       *  @param left_icorners the points in the left image of the calibration rig
       *  @param right_icorners the points in the right image of the calibration rig
       *  @param board size  the number of columns and rows of the calibration rig.
       * @return the camera parameters
       */
      CameraParams calibrate_camera(vector<CvPoint2D32f> left_icorners,
                      vector<int> &npoints, CvSize square_size, CvSize board_size,
                                      CvSize image_size);
```

```
	/** This function returns the intrinsic parameters of the camera from a list of
	 * images of the calibration rig. (All the images must be of the same size)
	 * @param left_images the filename of the left images
	 * @param right_images the filename of the right images
	 * @param board_size the number of rows and columns of the board
	 * @return the calibration parametesr
	 */
	CameraParams calibrate_camera(vector<string> left_images_path, CvSize square_size,
CvSize board_size);
};
```

This class defines two methods:

1. calibrate_camera(vector<string>, CvSize square_size, CvSize board_Size) : it take a string vector with the path to the images of the calibration rig. Square size are the size of each square of the checkerboard and board size is the size of the checkerboard (in columns and rows).
2. calibrate_camera(vector<CvPoint2D32f> left_icorners, vector<int> &npoints, CvSize square_size, CvSize board_size, CvSize image_size); this method calculates the camera parameters using the coordinates of the corners on the checkerboard.

The following example ilustrates better how to calibrate a camera using images of a calibration rig.

```
/// the following example looks for images of a calibration rig saved on the /images
/// directory. The calibration rig is a checkerboard with 6 rows and 9 cols, and
/// each square has a width of 25 mm and a height of 25 mm.
#include <iostream>
#include <dscalib.h>
using namespace std;
using namespace ds;
/// Performs the camera calibration with a know set of images, to
/// test the implementation
int main(int argc, char** argv)
{
	string exe_name(argv[0]);
	string exe_path = exe_name.substr(0,exe_name.find_last_of('/'));
	/// define the image paths
	vector<string> left_images, right_images;
	for(int i = 0; i < 6; i++)
	{
		std::stringstream index;
		index << i;
		left_images.push_back(exe_path + "/images/left-" + index.str() + ".jpg");
		right_images.push_back(exe_path + "/images/right-" + index.str() + ".jpg");
	}

	CvSize board_size = cvSize(6,9); /// 6 rows and 9 cols
	CvSize square_size = cvSize(25,25); /// 25 mm square
	/// test stereo calibration
	StereoCalibration scalib;
	StereoCameraParams sparams, sref_params;
	cout << "Testing stereo calibration ... ";
	sparams = scalib.calibrate_cameras(left_images,right_images,square_size,board_size);
	sparams.save_to_file(exe_path + "/images/sparams.xml");
	sref_params.read_from_file(exe_path + "/images/sparams.xml");
```

```
	//sref_params.save_to_file(exe_path + "/images/sparams_n.xml");
	if(sparams == sref_params)
	{
		cout << "Stereo Calibration OK" << endl;
	}
	else
	{
		cout << "Stereo Calibration FAIL" << endl;
		return -1;
	}
	cout << "Calibration error: " << scalib.error << endl;
	/// test the mono camera calibration
	Calibration calib;
	CameraParams params, ref_params;
	cout << "Testing mono calibration ... ";
	params = calib.calibrate_camera(left_images, square_size, board_size);
	params.save_to_file(exe_path + "/images/mparams.xml", "nikon");
	ref_params.read_from_file(exe_path + "/images/mparams.xml", "nikon");
	if(params==ref_params)
		cout << "Calibration OK" << endl;
	else
	{
		cout << "Calibration FAIL" << endl;
		return -1;
	}
	//params.undistort(simage);
	return 0;
}
```

Class StereoCalibration

The stereocalibration class works similar to the calibration class, but it needs a two sets of images, one from the left camera and one from the right camera. It returns a StereoCameraParameters object which contains the intrinsic and distortion parameter of each camera and the rotation, translation and fundamental matrix of the stereo camera system.

For stereo systems, a method to determine if the calibration was all right, is to use the equation 4. In fact if the calibration was ok, corresponding points must satisfy eq. 4. If this is not true, then there was an error during the calibration and it must be repeated, with a new set of images.

Undistortion

Once obtained the distortion parameters with the class calibration, the next step is to use them in order to correct the images in a procedure called undistortion. Figure 10 shows the results of the undistortion procedure. Notese that in figure 10(b) the pincushion distortion shown in figure 10(a) is no more present.

```
	/** This class is used to correct mono camera distortions*/
	class MonoRectify
	{
		public:
			/** Constructor
```

```
                     * @param params Camera parameters.
                     */
                    MonoRectify(CameraParams params);
                    ~MonoRectify();
                    /** Undistort a image using the given camera parameters
                     *  @param lsrc pointer to the original Image
                     *  @param rdst pointer to the undistorted image. Must be the same size
and type as original.
                     */
                    void UndistortImage(IplImage *src, IplImage* dst);
                    /** Set a new set of camera parameters.
                     *  @param params New camera params
                     */
                    void set_camera_params(CameraParams params);
             private:
                    CameraParams params;
                    CvMat *mx, *my;//, *R, *P;
      };
```

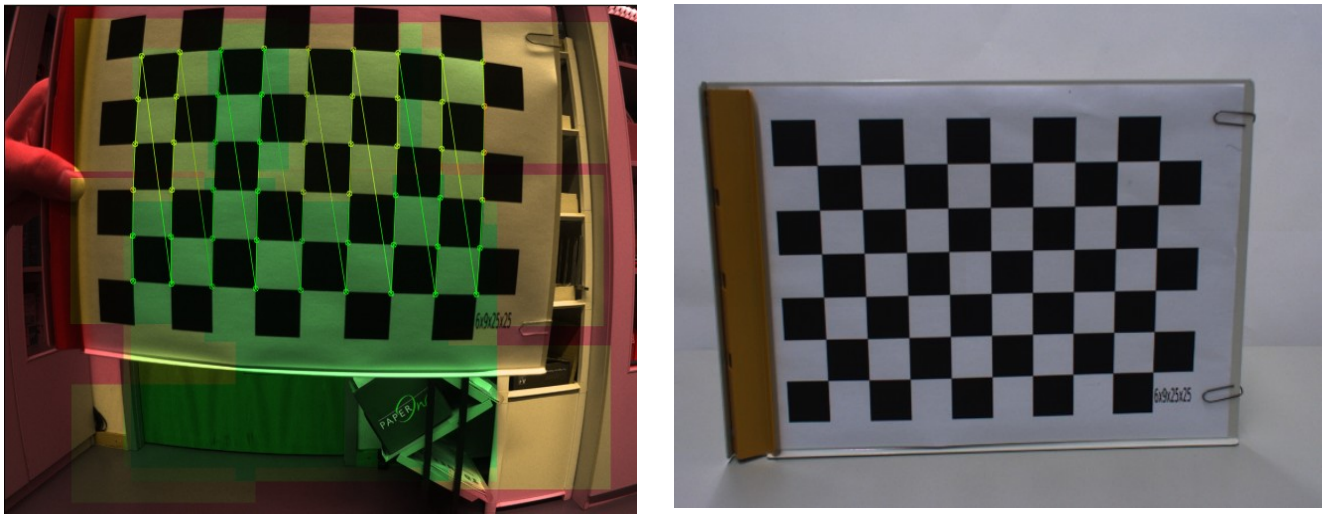

Illustration 10: (a) image from camera and (b) undistorted image.

Image rectification, disparity and 3D reconstruction.

In computer vision, 3D reconstruction is to recover the 3D information of a scene from a set of images. This is accomplished, in the case of stereo vision, by using triangulation.

In fact, as shown in figure 9, if we know the coordinates of two corresponding points, say xl and xr, the distance Z between the cameras and the real object P can be calculated with the following formula:

$$Z = \frac{B \times f}{disparity}$$

Formula 5: Depth and disparity relationship

Where B is the baseline or separation between cameras, f is the focal distance and disparity is the difference between coordinates of xl and xr.

On each camera, the center of the coordinate system is where the optical axis cuts the image plane, as shown on fig. 9 with the points cl and cr. Thus the coordinates of xl are relative to cl and the coordinates of xr are relative to cr.

The coordinates of the point P (on figure 9) relative to the link camera are calculated as:

$$Z = \frac{B \times f}{disparity}$$
$$Y = \frac{(y - c_y) \times Z}{f}$$
$$Z = \frac{(x - c_x) \times Z}{f}$$

cx and cy are from the intrinsics matrix of the link camera (see section camera parameters), x and y are the coordinates of the projected point xl relative to the center of the sensor in the right camera(see figure 9), and disparity is calculated as (xl-xr).

Thus a important step before the scene can be reconstructed is to find corresponding points in order to calculate the disparity. This task is accomplished (in this library) using a block matching approach, where a small block of pixels in one image is compared with another block in the second image, as shown in figure 11.

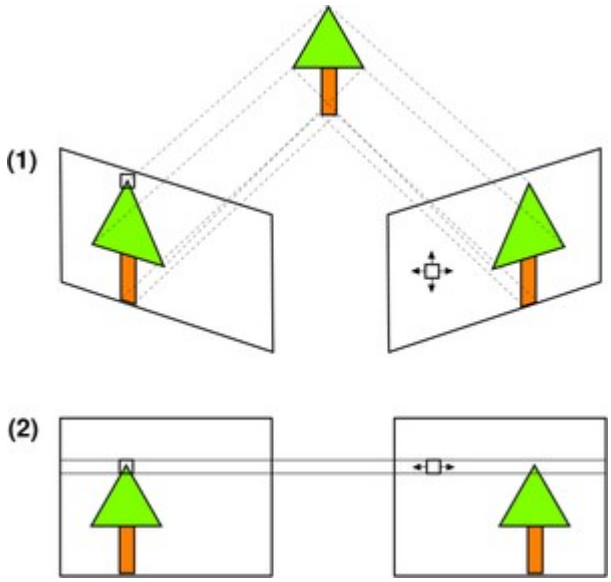

Illustration 11: Image before(1) and after(2) rectification.

To simplify the corresponding point search, the Imagen must be rectified. It means that the image must be reprojected to a standard coordinate system such as the corresponding epipolar lines are aligned. This task is performed by the StereoRectify object. It provides the UndistortImagePair method which

performs undistortion and rectification of the original images.

```
class StereoRectify
{
      public:
            /** Constructor
             *  @params params Stereo Camera Parameters.
             */
            StereoRectify(StereoCameraParams& params);
            ~StereoRectify();
            /** Undistort a image pair using the given camera parameters
             *  @param lsrc pointer to the original left Image
             *  @param rsrc pointer to the original right image
             *  @param ldst pointer to the undistorted left image. Must be the
same size and type as original.
             *  @param rdst pointer to the undistorted right image. Must be the
same size and type as original.
             */
            void UndistortImagePair(IplImage *lsrc, IplImage* rsrc, IplImage*
ldst, IplImage* rdst);
            /** Set a new set of camera parameters.
             *  @param params New stereo camera params
             */
            void set_camera_params(StereoCameraParams& params);
            StereoCameraParams& get_camera_params();
            CvMat *Q;
};
```

Disparity

As shown in figure 9, disparity is the difference between coordinates of the corresponding points on left and right images. If we drawn these values of disparity for each corresponding points on an stereo pair, we obtain a image like figure 12, where each pixel value represents the value of disparity at that point. This image is called disparity map (or because the disparity is related to depth as shown in equation 5, it is also called depth map).

Due to eq. 5 depth can be only calculated on those areas where the field of view of booth cameras intersects. This region is called horopter, as shown in figure 13. The distance (depth) at witch the horopter starts depends of the configuration of the stereo system (lens, distance between cameras) and can be calculated as follows:

$$H_{depht} = \frac{f \times (B + S)}{S}$$

where f is the focal distance, B is the baseline and S is the width of the sensor.

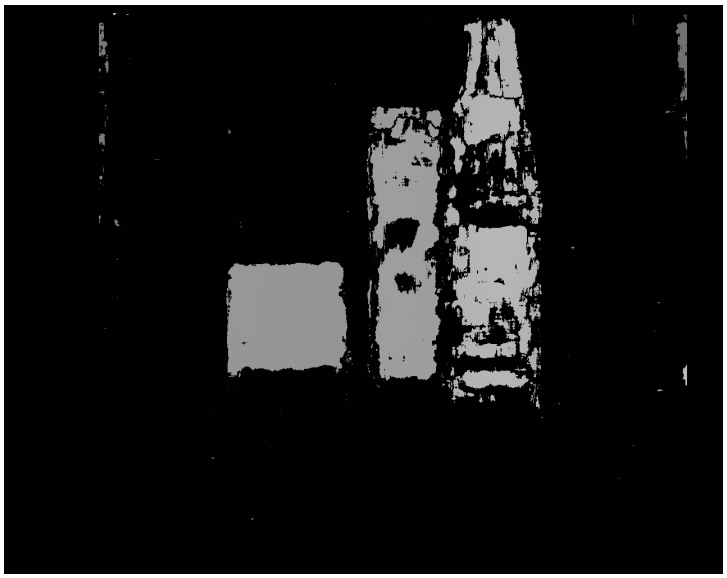

Illustration 12: Disparity map

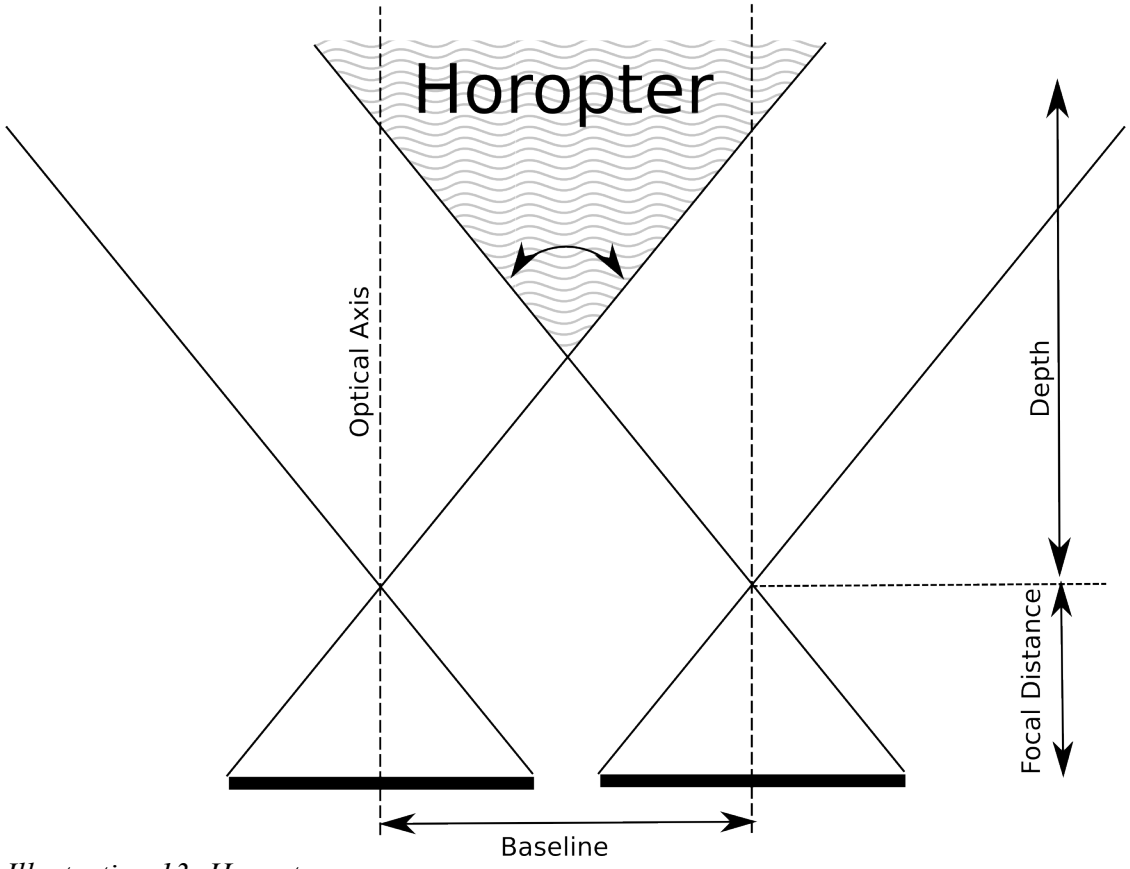

Illustration 13: Horopter

Dsvision provides the object disparity in order to calculate disparity.

```
class Disparity
{
	public:
		/** Constructor
		 * @param params the parameters of the stereo system
		 * @param image_size the size of the images
		 */
		Disparity(CvSize image_size);
		~Disparity();
		/** Calculates the disparity for a new pair of images
		 * @param left the left image.
		 * @param right the right image
		 **/
		void calculate_disparity(IplImage* left, IplImage* right);
		/** The disparity map as a 8bit per pixel image */
		IplImage* disp8;
		/** The disparity map as a 16 bit per pixel matrix */
		CvMat*  disp16;
		/** The disparity map size */
		CvSize size;
		void set_prefilter_size(int size){BMState->preFilterSize = size; };
		void set_prefilter_cap(int cap){BMState->preFilterCap = cap; };
		void set_SAD_window_size(int size){BMState->SADWindowSize = size;};
		void set_min_disparity(int disp){BMState->minDisparity = disp; };
		void set_num_of_disparities(int num){BMState->numberOfDisparities = num;};
		void set_texture_threshold(int th){BMState->textureThreshold = th; };
		void set_uniqueness_ratio(int ratio){BMState->uniquenessRatio = ratio; };
```

```
                    int get_prefilter_size(){return BMState->preFilterSize; };
                    int get_prefilter_cap(){return BMState->preFilterCap; };
                    int get_SAD_window_size(){return BMState->SADWindowSize; };
                    int get_min_disparity(){return BMState->minDisparity; };
                    int get_num_of_disparities(){return BMState->numberOfDisparities;};
                    int get_texture_threshold(){return BMState->textureThreshold; };
                    int get_uniqueness_ratio(){return BMState->uniquenessRatio; };

            private:
                    CvStereoBMState *BMState;
    };
```

This is a wrapper around the function cvFindStereoCorrespondenceBM from OpenCV. The disparity is calculated with subpixel accuracy and the result saved as a 16bit per pixel image (disp16). For visualization it is convenient to convert this image into a 8 bit per pixel image, which is stored into the disp8 variable.

The method calculate_disparity takes two undistorted and rectified images (the output of stereo_rectify) and stores the calculated disparity map on the variables disp16 and disp8, as 16 bit per pixel and 8 bit per pixel images.

Eq. 5 shows that depth resolution is not the same for all the depht range as for distances near the cameras, a small change in the depth produces a big change in the disparity, while for distances far from the cameras it is necessary a big change in the depth in order to produce a small change in the disparity. This means that the resolution of the stereo system changes in relation with the distance to the object observed, as explained in equation 6.

$$Range\ Resolution = \frac{Z^2}{b \times f} \times \Delta d$$

Formula 6: Range resolution

On Eq. 6 b is baseline, f is the focal distance, Z is the measured depth and (delta)d is the minimum disparity change (normally 1 pixel, but better resolution can be obtained using subpixel interpolation). For example, at a distance of 200 mm with a baseline of 6 mm, a focal distance of 5mm and a pixel size of 12 um the range resolution is:

$$Range\ Resolution = \frac{200^2}{6 \times 5} \times 1\ pixel \times 0{,}012 = 16\ mm$$

wich means that at a distance of 200 mm, the resolution is 16 mm, or the distance will be 200 mm ± 8mm.

Range resolution is not the same as range accuracy, which is a measure of how well the range computed by stereo compares with the actual range (Fig. 14). Range accuracy is sensitive to errors in camera calibration, including lens distortion and camera alignment errors.

Parameters for the block matching can be changed with the set methods, but for the most cases, the default values must perform well.

The only parameters that must be adjusted for new image sizes and stereo geometry are number of disparities (num_of_disparities) and minimum disparity (min_disparities), as shown in Example 1.

Example 1.
Accuracy determination for a stereo camera system.

	Measured Distance	[mm]
Real Distance	Mean	Standard Deviation
200	208.35	2.51
225	233.76	2.07
250	256.76	5.82
275	286.2	2.83
300	301.45	3.23
325	325.26	4.72
350	344.8	4.60
375	373.63	4.25
400	397.92	7.69
425	419.25	3.94
500	491.83	2.74
525	516.32	1.94
550	538.83	1.68
575	563.2	3.40
600	585.13	2.70

Table 3: Accuracy measurements of a stereo system (baseline = 5mm, focal distance = 6mm)

Example 1.
Accuracy determination for a stereo camera system:
Baseline = 44.46 mm
Focal Distance = 3 mm
Pixel Width = 4.2 um.

1. Calculate the disparity range:
   Z = Baseline*Focal distance / disparity. (Fig. 15)
2. Calculate the Range resolution.
   Range Resolution = Z*Z/(b*f) * Δd (Fig. 16)
3. Measure the actual accuracy (Fig. 14, table 3)

min_disparity and num_of_disparities move the horopter (the disparity search space) from its theoretical minimum (see fig. 13).

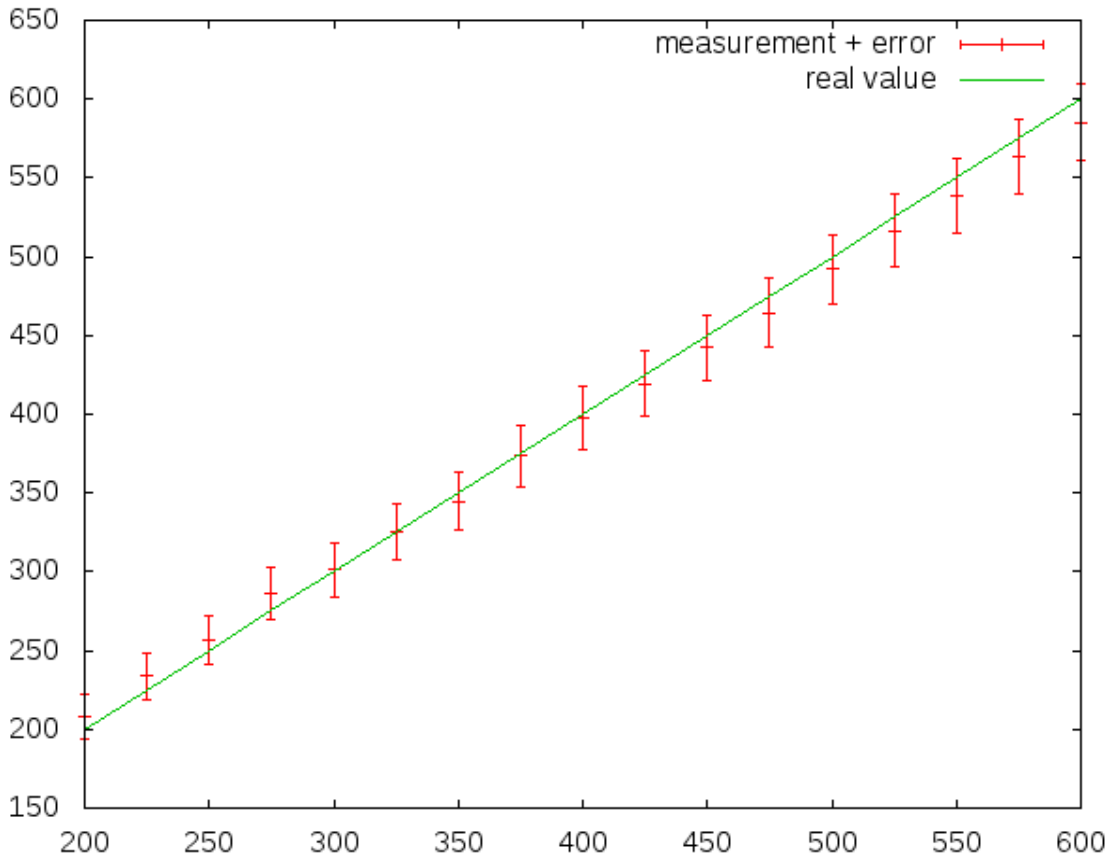

Illustration 14: Real depth vs Measured depth from stereo system (mm).(table 3)

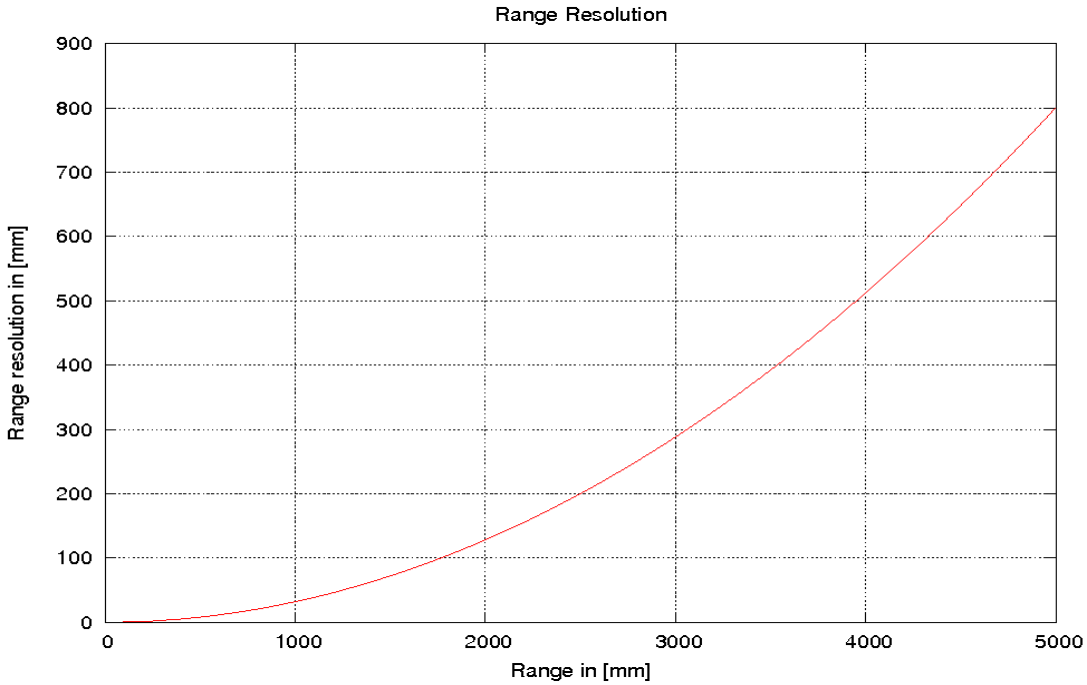

Illustration 15: Teoretical Range Resolution for example 1

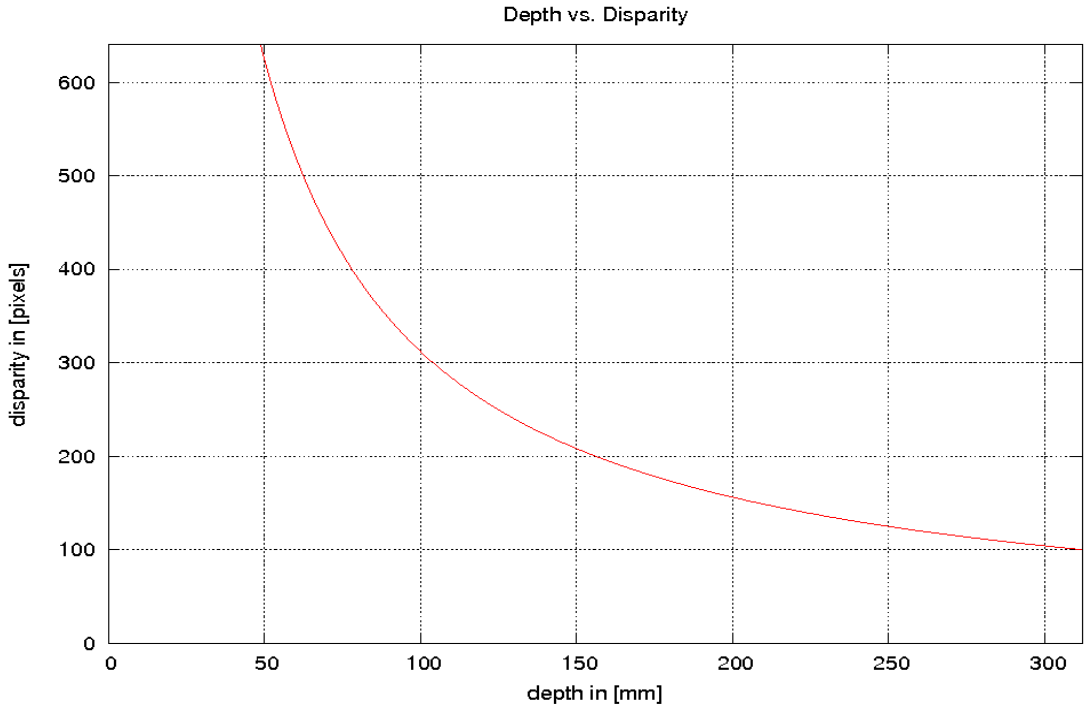

Illustration 16: Measured disparity as function of depth for example 1

```
#include <iostream>
#include <dscalib.h>
#include <dsundistort.h>
#include <dsstereo.h>
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
using namespace ds;
int main(int argc, char** argv)
{
        /// extract the working path
        string exe_name(argv[0]);
        string exe_path = exe_name.substr(0,exe_name.find_last_of('/'));
        StereoCameraParams params;
       //Load the stereo parameters from a file
        params.read_from_file(exe_path + "/images/sparams.xml");
       /// Load images from directory
        IplImage* left = cvLoadImage((exe_path + "/images/left-0.jpg").c_str());
        IplImage* right = cvLoadImage((exe_path + "/images/right-0.jpg").c_str());
        IplImage* uleft = cvCloneImage(left);
        IplImage* uright = cvCloneImage(right);
       /// initialize the rectificator
        StereoRectify sr(params);
       /// undistort and rectify image pair
        sr.UndistortImagePair(left,right,uleft,uright);
       /// initialize the disparity object
        Disparity disp(params.leftcam.imageSize);
       /// calculate the disparity
        disp.calculate_disparity(uleft,uright);
       /// Show the disparity map
        cvShowImage("disparity", disp.disp8);
        cvWaitKey(0);
        disp.size = cvSize(640,480);
        disp.calculate_disparity(uleft,uright);
        cvShowImage("disparity", disp.disp8);
        cvWaitKey(0);
        return 0;
 }
```

Table 4: Using the disparity class in order to calculate the disparity map

Descriptors

Surf and Sift descriptors are part of this library. We use the SURF implementation from OPENCV and SIFT from IVT. The following example shows how to use them in order to calculate descriptors.

```
#include <dsdescriptor.h>
#include <dssurf.h>
#include <iostream>
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>
#include <dsstereo.h>

using namespace std;
using namespace ds;

int main(int argc, char** argv)
{
        string exe_name(argv[0]);
```

```
        string exe_path = exe_name.substr(0,exe_name.find_last_of('/'));
        IplImage* image = cvLoadImage((exe_path + "/images/l-eft-t.bmp").c_str());
        IplImage* ref_image = cvLoadImage((exe_path + "/images/r-ight-t.bmp").c_str());

        DSSurf sc(1, 1, 700, false);
        bool ft = true;
        EpipolarFilter filter(.1);

//while(true)
{
        DsDescriptorArray image_desc = sc.find_descriptors(image);
        DsDescriptorArray ref_desc = sc.find_descriptors(ref_image);

        cout << "found: " << image_desc.size() << " descriptors" << endl;
        cout << "found: " << ref_desc.size() << " descriptors" << endl;

        DsDescriptorMatches matches = find_matches(image_desc, ref_desc, NearestNeighbor);
        cout << "found: " << matches.size() << " matches" << endl;
        if(ft)
        {
                filter.set_fundamental_matrix(matches);
                ft = false;
        }
        DsDescriptorMatches fmatches = filter.filter_matches(matches);

        cout << "found:f: " << fmatches.size() << endl;
        //draw_descriptors(image, image_desc);
        IplImage *result = draw_matches(image, ref_image, fmatches);
        IplImage *r1 = draw_matches(image, ref_image, matches);

        cvShowImage("descriptors", r1);
        cvShowImage("filtered descriptors", result);
        cvWaitKey(30);
        cvReleaseImage(&result);
        cvReleaseImage(&r1);
}
        return 0;
}
```

Segmentation

Two basic segmentation algorithms are implemented. Connected component analysis, and color segmentation.

Connected component analysis segment the (grayscale-)images into regions with the same (or similar) pixel value, where color segmentation selects those regions of the (color-)imagen with the same color.

The class ds::Blob implements the connected components analysis. It has following interface:

getMask(); returns a CvMat with ones where the blobs exists and null where it does not.
GetCenter(); returns the 2d coordinates of the blob.
GetContour(); returns a set of points that describes the contour of the blob.
GetBoundingRect(): Return the bounding box of the blob.

Visualization

Visualization of 3D data (Point Clouds) is specially usefull when working with tri-dimensional spaces. The library provides an utilitiy for data visualization called dsViewer. It is bassed on the PointCloudViewer from the StairVision Library (http://ai.stanford.edu/~sgould/svl/) with facilities for data visualization through the network.

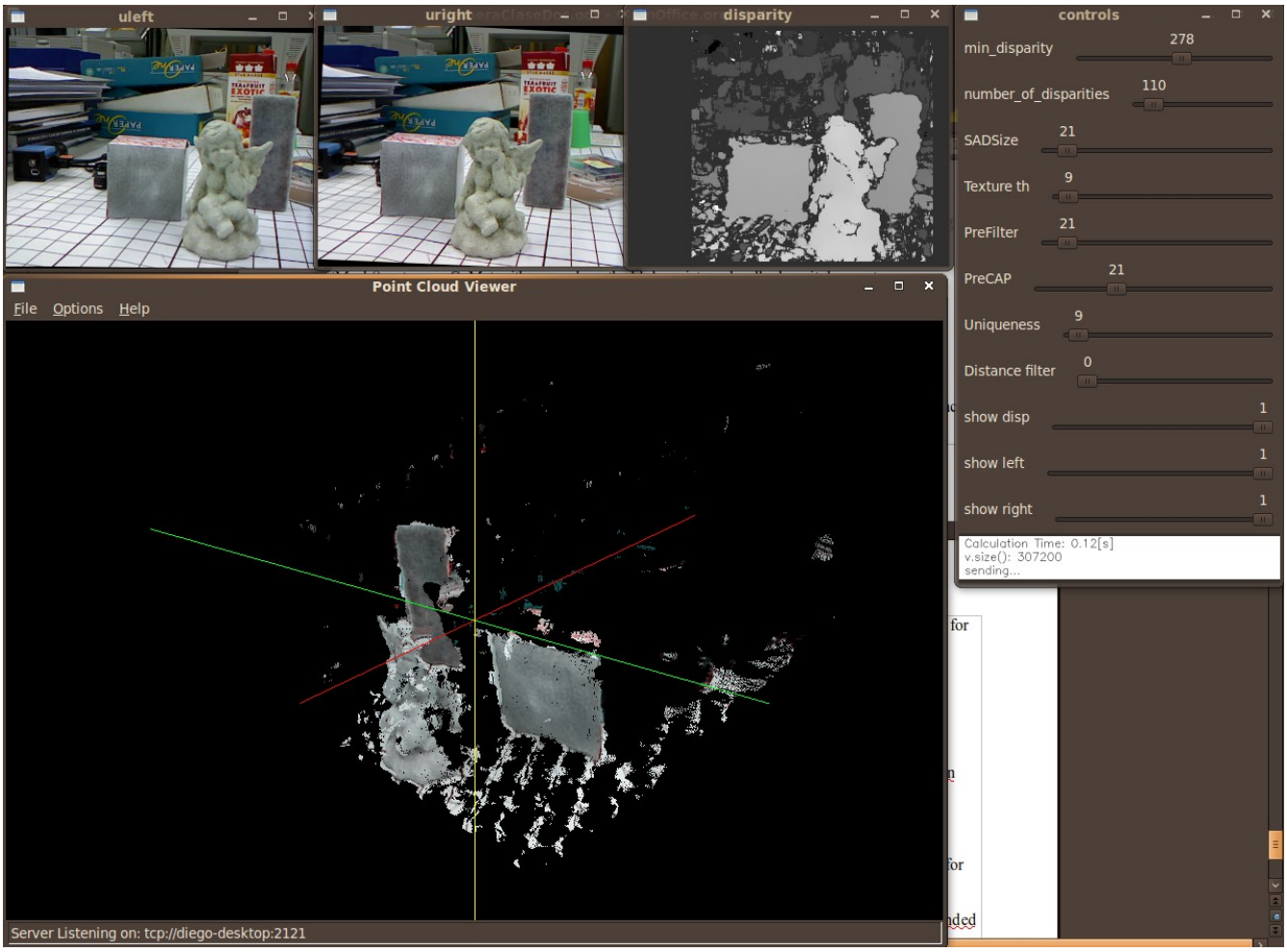

Illustration 17: 2D and 3D visualization Example

The communication of data for visualization occurs through the network.

Data Communication

Simple data communication is provided using the yami (yet another message interface). It is used for data transmision fundamentally for visualization purposes.

Usage of the library is ilustrated in the dsviewer application. It shows how a Point cloud can be sended for visualization in a remote computer.

Better documentation about the yami library can be found at:

http://www.inspirel.com/yami4/