Autonomous Robot: Grid Localization

Guillaume Lemaître
Heriot-Watt University, Universitat de Girona, Université de Bourgogne
g.lemaitre58@gmail.com

I. Introduction

In this paper, we will present a short introduction and utilization of the library TAG Tatille. This library is used to manipulate data given from IP camera. First, we will present the implementation to grab images and properties of the acquisition and image. Then, we will present an implementation to convert the color input image to a gray image. We will conclude with an implementation which allows to compute the gray and color gradient image.

II. Acquisition and properties

A. Image acquisition

Before any acquisition, we need to make a routine to connect to the device. First, an open live channel has to be opened using PC and camera IP address and the following function:

```
char m_IpPc [20] = "84.88.155.17";
char m_IpDev[20] = "84.88.155.15";
HANDLE hdata = NULL;

hdata = itf_clOpenLiveChannel(m_IpDev,
    m_IpPc);
```

Then, a data channel has to be opened using the same parameter as before:

```
HANDLE          chDeviceData;

chDeviceData = itf_clOpenDataChannel(
    m_IpDev, m_IpPc,0);
```

After, these two steps, acquisition of images is available. This acquisition can be done with the following code:

```
int result;

result = itf_clStartSnapshot(chDeviceData)
    ;
```

Figure 1. Example of grabbed image

In order to read the image grabbed, the following code is used:

```
char*   localImage;

result=itf_clReadImage(hdata, (void**)&
    localImage);
```

The variable *result* return a value regarding if the data are corrupted or not. An example of grabbed image is presenting on the picture 1.

B. Frame rate

The following function gives the time elapsed since the program started:

```
time = GetTickCount();
```

In order to compute the frame rate, when an image is grabbed, the elapsed time between the previous grabbed image and the actual grabbed image is computed. To compute the frame rate we have to apply the following formula:

$$F_r = \frac{1000}{I_t - I_{t-1}}$$

where I_t and I_{t-1} are the time when we grabbed the actual image and the previous image. Mean frame rate are equally computed due to the fluctuation of the *Moment frame rate*. The code below shows the implementation to compute the frame rate.

```
//Information frame rate
//Put a counter to know the elaps time
    between the last time and now
time = GetTickCount();
if (numFrames > 1)
{
        cout << "Time elapsed: " << time -
            time_1 << endl;
        double rate = time - time_1;
        mean = mean_1 + (double(time -
            time_1) - mean_1)/(numFrames -
             1);
        mean_1 = mean;
        cout << "Mean frame rate: " <<
            (1000/mean) << "img/s" << endl
            ;
        cout << "Moment frame rate: " <<
            (1000.00/(rate)) << "img/s" <<
             endl;
}

time_1 = time;
```

During the experimentation, the **maximum frame rate** achieved was about **7.3** images per seconds.

C. Image properties

TAG Tatille library permits to give several information of the image. In this section, we will present the different functions which provide some information about the picture.

1) $TIL\_get\_image\_pixel\_type$ function: This function allows to know the coding type of the image. The implementation is as follow:

```
char* localImage;
long pixel_type;

TIL_get_image_pixel_type(&pixel_type, (
    Tbanco*)localImage);
```

An integer is assigned to the variable $pixel\_type$. This integer have a corresponding type given by the documentation of the library and indicated in the table I.

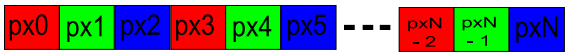

Figure 2. Structure of the data buffer

We will work in next steps with a **RGB888** pixel type. Hence, the structure of the buffer will be as shown on the figure 2

2) $TIL\_get\_image\_bits\_pixel$ function: This function allows to know the number of bits per pixels. The relation is given by the type previously presented. The implementation is as follow:

```
char* localImage;
long bpp;

TIL_get_image_bits_pixel(&bpp,(Tbanco*)
    localImage);
```

The number of bits per pixel was **24**.

3) $TIL\_get\_image\_width$ and $TIL\_get\_image\_height$ functions: This function allows to know the width and height of the image. The implementation is as follow:

```
char* localImage;
long width_im;
long height_im;

TIL_get_image_width(&width_im,(Tbanco*)
    localImage);
TIL_get_image_height(&height_im,(Tbanco*)
    localImage);
```

The dimension of the image was **640** pxs (width) by **480** pxs (height).

4) $TIL\_get\_image\_stream\_size$ function: This function allows to know the size of the data buffer. Basically, we can compute the size of the stream as follow:

$$size = height \times width \times 3$$

The implementation is as follow:

```
char* localImage;
long size_buffer

TIL_get_image_stream_size(&size_buffer,(
    Tbanco*)localImage);
```

The size of the buffer is our example is **921600**.

Pixel Type	Integer corresponding
$YGRAY\_8$	0
$RGB565\_INTERLEAVED\ RGB\_565$	1
$RGB323\_INTERLEAVED\ RGB\_323$	2
$RGB888\_INTERLEAVED\ RGB\_888$	3
$BAYER\ BAYER$	4
$MOSAIC\ MOSAIC$	5
$YCBCR420\ YCBCR420$	6
$BGR888\_INTERLEAVED\ BGR888$	7
$CRYCBY422\_INTERLEAVED\ CRYCBY422$	8
$RGBY\_INTERLEAVEDRGBY\_32BIT$	9

Table I
TABLE OF CORRESPONDENCES FOR PIXEL TYPE VARIABLE

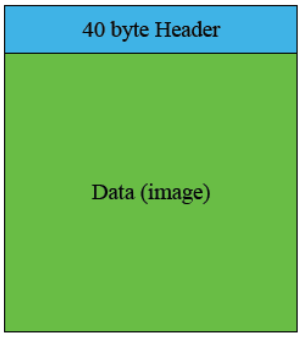

Figure 3. Structure of an image in the TAG Tatille library

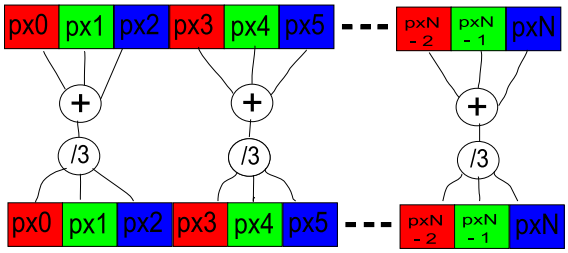

Figure 4. Representation of the conversion from color image to gray scale image

III. GRAY SCALE IMAGE

First, before to start the conversion, we have to allocate an empty image to save the image. In order to save correctly, the type of the gray image has to be the same than the color image. If the color image type is RGB88, we have to save the gray image as a RGB888 image. In the TAG Tatille library, an image have the organization presented on the figure 3.

The procedure to create an image is to allocate a buffer of the size of the streaming plus the header. Then we have to copy the original header inside the header of the empty image. The following code is an implementation of this aspect:

```
int offsetHeader = 40;
char* grayImage = (char*) itf_Malloc(3*
    width_im*height_im + offsetHeader);

for (int i = 0 ; i < offsetHeader ; i++)
{
        grayImage[i] = localImage[i];
}
```

In order to work only with data intensity image, we use the function $TIL\_get\_image\_stream\_ptr$ which give the pointer of the data buffer. To compute the intensity of the gray scale image, we use the following formula:

$$IG_{p,p+1,p+2} = \frac{IC_p + IC_{p+1} + IC_{p+2}}{3}$$

Where IG is the intensity of the gray level and IC the intensity of the color image.

The function which allows to convert the color image in gray scale image is presented in the appendix A-A.

The figure 4 represents the manipulation realized to obtain a correct image while the figure 5 shows an example of gray level image saved.

IV. GRAY SCALE AND COLOR GRADIENTS

To simplify the computation, we created a function which convert the data buffer in a two dimensional array. This function allows equally to split the different channels. This function is shown in the appendix A-B. This method has a problem that we lose time during the computation of the matrix. However, this method is more intuitive for next steps to explain filtering. Figure 6 represents the transformation realized.

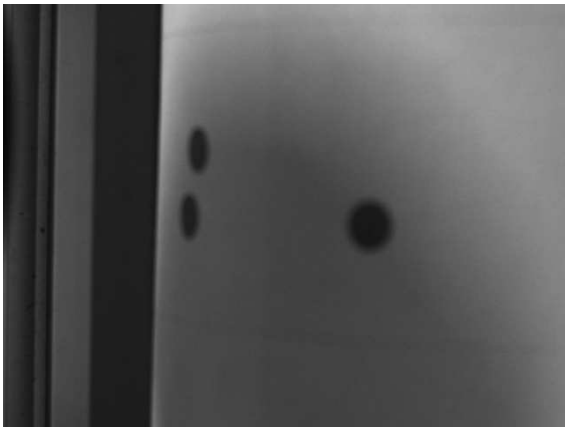

Figure 5. Result of an gray scale image

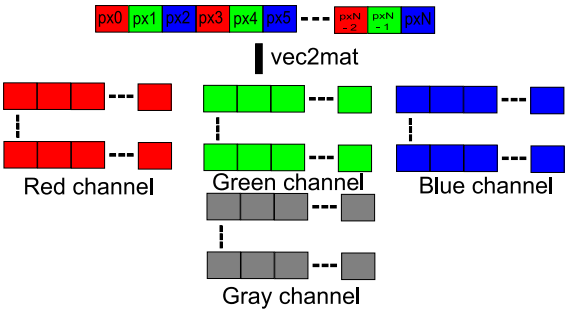

Figure 6. Transformation from vector to matrix

A. Operator filter

To compute the gradient of an input image, we have to choose an operator. We chose Sobel operator. The Sobel masks are as follow:

1	0	-1
2	0	-2
1	0	-1

Table II
SOBEL MASK X

-1	-2	-1
0	0	0
1	2	1

Table III
SOBEL MASK Y

B. Gray scale gradient

In order to compute the gradient of the gray scale image, we have to compute the gradient in x and the gradient in y. To compute these gradients, we have to do the convolution between the Sobel masks and the matrix image:

$$G_x = S_x * Im$$
$$G_y = S_y * Im$$

Then, the gradient is equal to the magnitude of G_x and G_y. We can formalize as follow:

$$G = \sqrt{G_x^2 + G_y^2}$$

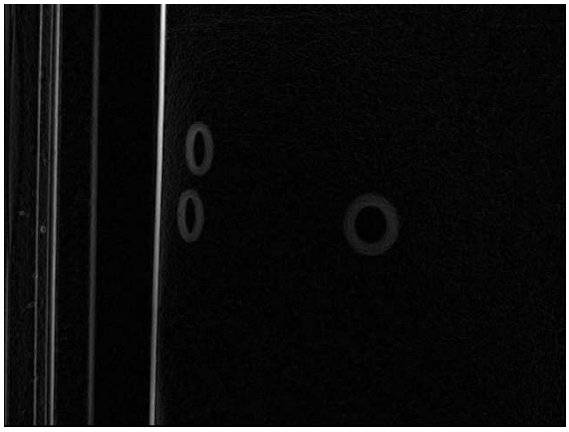

Figure 7. Gray scale gradient image

Figure 7 shows the result of the gray scale gradient.

To save the image, we copy the value of the gradient in the channel red, green and blue of the image to respect the image type.

The frame rate during the computation of the gradient is **4.9** images per seconds.

C. Color gradient

For the color gradient, the method is exactly the same as below. We compute the gradient for the different channels red, green and blue. Then, we copy the intensity of the different gradient at the right inside the data buffer in order to save correctly the image. The obtain image is shown in the figure 8.

The frame rate during the computation of the gradient is **2.1** images per seconds.

We can observe that the frame rate is decreasing when we realize more image processing. We can also notice that the operation of the color gradient is three

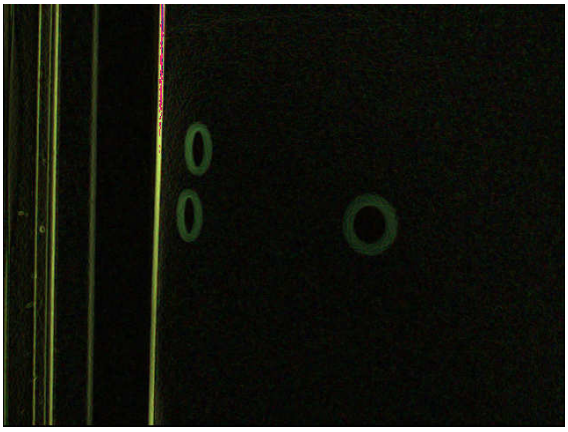

Figure 8. Color gradient image

times longer due to realized three times the convolution operation.

V. Conclusion

In this paper, we presented a short introduction and utilization of the library TAG Tatille. First, we presented the implementation to grab images and properties of the acquisition and image. Then, we presented an implementation to convert the color input image to a gray image. Finally, we concluded with an implementation which allows to compute the gray and color gradient image.

APPENDIX A
CODE

A. rgb2gray function

```
//////////////////////////////////////////////////////////////
// Function to convert a rgb image in grayscale image
//////////////////////////////////////////////////////////////

void rgb2gray(UBYTE *&rgbstr, UBYTE *&graystr, int size_buffer)
{
        for (int i = 0 ; i < size_buffer ; i=i+3)
        {
                graystr[i] = (rgbstr[i]+rgbstr[i+1]+rgbstr[i+2])/3;
                graystr[i+1] = (rgbstr[i]+rgbstr[i+1]+rgbstr[i+2])/3;
                graystr[i+2] = (rgbstr[i]+rgbstr[i+1]+rgbstr[i+2])/3;
        }
}
```

B. vec2mat function

```
//////////////////////////////////////////////////////////////
// Function to convert the data vector in data matrix
//////////////////////////////////////////////////////////////

void vec2mat(UBYTE *&str, int **& mat, int height_im, int width_im, int type)
{
        for (int i = 0 ; i < height_im ; i++)
        {
                for (int j = 0 ; j < width_im ; j++)
                {
                        mat[i][j] = str[(i*width_im)*3+(j*3 + type)];
                }
        }
}
```

C. computeGradient function

```
//////////////////////////////////////////////////////////////
// Function to compute the gray scale image gradient
//////////////////////////////////////////////////////////////

void computeGradient(int **&gradientx, int **&gradienty, int **&imagein, UBYTE *&
    gradientimage, int height_im, int width_im)
{
        int counter = 3;
        for(int i = 1 ; i < (height_im -1) ; i++)
        {
                for (int j = 1 ; j < (width_im - 1) ; j++)
                {
                        double vac = ((imagein[i-1][j-1]*gradientx[0][0] + imagein[i-1][j]*
                            gradientx[0][1] + imagein[i-1][j+1]*gradientx[0][2] + imagein[i
                            ][j-1]*gradientx[1][0] + imagein[i][j]*gradientx[1][1] + imagein
                            [i][j+1]*gradientx[1][2] + imagein[i+1][j-1]*gradientx[2][0] +
                            imagein[i+1][j]*gradientx[2][1] + imagein[i+1][j+1]*gradientx
                            [2][2]))*(imagein[i-1][j-1]*gradientx[0][0] + imagein[i-1][j]*
                            gradientx[0][1] + imagein[i-1][j+1]*gradientx[0][2] + imagein[i
                            ][j-1]*gradientx[1][0] + imagein[i][j]*gradientx[1][1] + imagein
```

```
                        [i][j+1]*gradientx[1][2] + imagein[i+1][j-1]*gradientx[2][0] +
                        imagein[i+1][j]*gradientx[2][1] + imagein[i+1][j+1]*gradientx
                        [2][2])+(imagein[i-1][j-1]*gradienty[0][0] + imagein[i-1][j]*
                        gradienty[0][1] + imagein[i-1][j+1]*gradienty[0][2] + imagein[i
                        ][j-1]*gradienty[1][0] + imagein[i][j]*gradienty[1][1] + imagein
                        [i][j+1]*gradienty[1][2] + imagein[i+1][j-1]*gradienty[2][0] +
                        imagein[i+1][j]*gradienty[2][1] + imagein[i+1][j+1]*gradienty
                        [2][2])*(imagein[i-1][j-1]*gradienty[0][0] + imagein[i-1][j]*
                        gradienty[0][1] + imagein[i-1][j+1]*gradienty[0][2] + imagein[i
                        ][j-1]*gradienty[1][0] + imagein[i][j]*gradienty[1][1] + imagein
                        [i][j+1]*gradienty[1][2] + imagein[i+1][j-1]*gradienty[2][0] +
                        imagein[i+1][j]*gradienty[2][1] + imagein[i+1][j+1]*gradienty
                        [2][2]);
                gradientimage[counter] = gradientimage[counter+1] = gradientimage[
                    counter+2] = (UBYTE) sqrt(vac);
                counter = counter + 3;
                if (j == width_im - 2)
                {
                        counter = counter + 6;
                }
        }
    }
}
```

D. computeGradientColor

```
////////////////////////////////////////////////////////////////
// Function to compute color gradient image
////////////////////////////////////////////////////////////////

void computeGradientColor(int **&gradientx, int **&gradienty, int **&redChannel, int **&
    greenChannel, int **&blueChannel, UBYTE *&gradientimage, int height_im, int width_im)
{
        int counter = 3;
        for(int i = 1 ; i < (height_im -1) ; i++)
        {
                for (int j = 1 ; j < (width_im - 1) ; j++)
                {
                        double gradred = ((redChannel[i-1][j-1]*gradientx[0][0] + redChannel
                            [i-1][j]*gradientx[0][1] + redChannel[i-1][j+1]*gradientx[0][2]
                            + redChannel[i][j-1]*gradientx[1][0] + redChannel[i][j]*
                            gradientx[1][1] + redChannel[i][j+1]*gradientx[1][2] +
                            redChannel[i+1][j-1]*gradientx[2][0] + redChannel[i+1][j]*
                            gradientx[2][1] + redChannel[i+1][j+1]*gradientx[2][2]))*(
                            redChannel[i-1][j-1]*gradientx[0][0] + redChannel[i-1][j]*
                            gradientx[0][1] + redChannel[i-1][j+1]*gradientx[0][2] +
                            redChannel[i][j-1]*gradientx[1][0] + redChannel[i][j]*gradientx
                            [1][1] + redChannel[i][j+1]*gradientx[1][2] + redChannel[i+1][j
                            -1]*gradientx[2][0] + redChannel[i+1][j]*gradientx[2][1] +
                            redChannel[i+1][j+1]*gradientx[2][2])+(redChannel[i-1][j-1]*
                            gradienty[0][0] + redChannel[i-1][j]*gradienty[0][1] +
                            redChannel[i-1][j+1]*gradienty[0][2] + redChannel[i][j-1]*
                            gradienty[1][0] + redChannel[i][j]*gradienty[1][1] + redChannel[
                            i][j+1]*gradienty[1][2] + redChannel[i+1][j-1]*gradienty[2][0] +
                             redChannel[i+1][j]*gradienty[2][1] + redChannel[i+1][j+1]*
                            gradienty[2][2])*(redChannel[i-1][j-1]*gradienty[0][0] +
                            redChannel[i-1][j]*gradienty[0][1] + redChannel[i-1][j+1]*
                            gradienty[0][2] + redChannel[i][j-1]*gradienty[1][0] +
                            redChannel[i][j]*gradienty[1][1] + redChannel[i][j+1]*gradienty
                            [1][2] + redChannel[i+1][j-1]*gradienty[2][0] + redChannel[i+1][
                            j]*gradienty[2][1] + redChannel[i+1][j+1]*gradienty[2][2]);
```

```
                        gradientimage[counter] = (UBYTE) sqrt(gradred);
                        double gradgreen = ((greenChannel[i-1][j-1]*gradientx[0][0] +
                            greenChannel[i-1][j]*gradientx[0][1] + greenChannel[i-1][j+1]*
                            gradientx[0][2] + greenChannel[i][j-1]*gradientx[1][0] +
                            greenChannel[i][j]*gradientx[1][1] + greenChannel[i][j+1]*
                            gradientx[1][2] + greenChannel[i+1][j-1]*gradientx[2][0] +
                            greenChannel[i+1][j]*gradientx[2][1] + greenChannel[i+1][j+1]*
                            gradientx[2][2]))*(greenChannel[i-1][j-1]*gradientx[0][0] +
                            greenChannel[i-1][j]*gradientx[0][1] + greenChannel[i-1][j+1]*
                            gradientx[0][2] + greenChannel[i][j-1]*gradientx[1][0] +
                            greenChannel[i][j]*gradientx[1][1] + greenChannel[i][j+1]*
                            gradientx[1][2] + greenChannel[i+1][j-1]*gradientx[2][0] +
                            greenChannel[i+1][j]*gradientx[2][1] + greenChannel[i+1][j+1]*
                            gradientx[2][2])+(greenChannel[i-1][j-1]*gradienty[0][0] +
                            greenChannel[i-1][j]*gradienty[0][1] + greenChannel[i-1][j+1]*
                            gradienty[0][2] + greenChannel[i][j-1]*gradienty[1][0] +
                            greenChannel[i][j]*gradienty[1][1] + greenChannel[i][j+1]*
                            gradienty[1][2] + greenChannel[i+1][j-1]*gradienty[2][0] +
                            greenChannel[i+1][j]*gradienty[2][1] + greenChannel[i+1][j+1]*
                            gradienty[2][2])*(greenChannel[i-1][j-1]*gradienty[0][0] +
                            greenChannel[i-1][j]*gradienty[0][1] + greenChannel[i-1][j+1]*
                            gradienty[0][2] + greenChannel[i][j-1]*gradienty[1][0] +
                            greenChannel[i][j]*gradienty[1][1] + greenChannel[i][j+1]*
                            gradienty[1][2] + greenChannel[i+1][j-1]*gradienty[2][0] +
                            greenChannel[i+1][j]*gradienty[2][1] + greenChannel[i+1][j+1]*
                            gradienty[2][2]);
                        gradientimage[counter + 1] = (UBYTE) sqrt(gradgreen);
                        double gradblue = ((blueChannel[i-1][j-1]*gradientx[0][0] +
                            blueChannel[i-1][j]*gradientx[0][1] + blueChannel[i-1][j+1]*
                            gradientx[0][2] + blueChannel[i][j-1]*gradientx[1][0] +
                            blueChannel[i][j]*gradientx[1][1] + blueChannel[i][j+1]*
                            gradientx[1][2] + blueChannel[i+1][j-1]*gradientx[2][0] +
                            blueChannel[i+1][j]*gradientx[2][1] + blueChannel[i+1][j+1]*
                            gradientx[2][2]))*(blueChannel[i-1][j-1]*gradientx[0][0] +
                            blueChannel[i-1][j]*gradientx[0][1] + blueChannel[i-1][j+1]*
                            gradientx[0][2] + blueChannel[i][j-1]*gradientx[1][0] +
                            blueChannel[i][j]*gradientx[1][1] + blueChannel[i][j+1]*
                            gradientx[1][2] + blueChannel[i+1][j-1]*gradientx[2][0] +
                            blueChannel[i+1][j]*gradientx[2][1] + blueChannel[i+1][j+1]*
                            gradientx[2][2])+(blueChannel[i-1][j-1]*gradienty[0][0] +
                            blueChannel[i-1][j]*gradienty[0][1] + blueChannel[i-1][j+1]*
                            gradienty[0][2] + blueChannel[i][j-1]*gradienty[1][0] +
                            blueChannel[i][j]*gradienty[1][1] + blueChannel[i][j+1]*
                            gradienty[1][2] + blueChannel[i+1][j-1]*gradienty[2][0] +
                            blueChannel[i+1][j]*gradienty[2][1] + blueChannel[i+1][j+1]*
                            gradienty[2][2])*(blueChannel[i-1][j-1]*gradienty[0][0] +
                            blueChannel[i-1][j]*gradienty[0][1] + blueChannel[i-1][j+1]*
                            gradienty[0][2] + blueChannel[i][j-1]*gradienty[1][0] +
                            blueChannel[i][j]*gradienty[1][1] + blueChannel[i][j+1]*
                            gradienty[1][2] + blueChannel[i+1][j-1]*gradienty[2][0] +
                            blueChannel[i+1][j]*gradienty[2][1] + blueChannel[i+1][j+1]*
                            gradienty[2][2]);
                        gradientimage[counter + 2] = (UBYTE) sqrt(gradblue);
                        counter = counter + 3;
                        if (j == width_im - 2)
                        {
                                counter = counter + 6;
                        }
                }
        }
}
```

E. Main function, declarations and parameters

```
// TAG.cpp : Defines the entry point for the console application.
//
#include <windows.h>
#include <iostream>
#include <tchar.h>
#include <stdio.h>
#include <conio.h>
#include "itf_global.h"
#include "itf_utility.h"
#include "itf_interface.h"
#include "TIL_utility.h"
#include "itf_GigaEth.h"
#include "itf_ethernet.h"
#include "TAGLib.h"
#include "Iphlpapi.h"
#include <string>
#include <math.h>

using namespace std;

//////////////////////////////////////////////////////////////////////////////
//      TAG.cpp
//              Code for a single threaded application grabbing images from Tattile TAG
    Camera.
//              Without using the TAG filter.
//              Based on TAGConsole project.
//
//              RTI Module, MASTER VIBOT 2009.
//
//              University of Girona, Robert Mart (marly@eia.udg.edu)
//
//////////////////////////////////////////////////////////////////////////////


char m_IpPc [20] = "84.88.155.17";
char m_IpDev[20] = "84.88.155.15";
//////////////////////////////////////////////////////////////////////////////

#define GRAY    0
#define RED             0
#define GREEN   1
#define BLUE    2

typedef struct {
        char description[255];
        char mac[255];
        char bind[255];
}bindType;

int                     yDisplay;
int                     xDisplay;
BOOL            m_enableFilter;
HANDLE          chDeviceData;
HANDLE          hFilter = NULL;
HANDLE          m_hStopEvent;
bool            running;
int                     m_bind;
SYSTEMTIME      timeS, timeE;
HANDLE hdata = NULL;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
        long result;

        system("cls");
        printf("\n\n\n");

        printf("\n--- Parameters ----------------------------------------\n");
        printf("Pc IP Address   = %s\n", m_IpPc);
        printf("TAG IP Address  = %s\n", m_IpDev);
        printf("\n");

        int port = 0;

        hdata = itf_clOpenLiveChannel(m_IpDev, m_IpPc);
        if(hdata== NULL)
        {
                printf("clOpenLiveChannel failed\n");
                return 0;
        }

        // Open communication channel
        chDeviceData = itf_clOpenDataChannel(m_IpDev, m_IpPc,0);
        if(chDeviceData == 0)
        {
                printf("itf_clOpenDataChannel failed\n");
                result = TAGClose(hFilter);
                itf_Close(&hdata);
                return 0;
        }

        #define CHECK4ERROR
                                \
        if(result != 0)
                                \
        {
                                        \
        printf("Result = %d Line %d\n",result,__LINE__);        \
        result = itf_clStartStopDevice(chDeviceData, 0);        \
        itf_Close(&chDeviceData);                                                       \
        itf_Close(&hdata);
                        \
        return 0;
                                \
        }

//////////////////////////////////////////////////////////////////////////////
// TODO
// if necessary you can set parameters on TAG using following setup functions
//////////////////////////////////////////////////////////////////////////////
//      result = itf_setExposureTimeAsTrigger(chDeviceData,T_GEN_DISABLE);      CHECK4ERROR;
        // Disable Exposure time as trigger
//      result = itf_clEnableFilter55(chDeviceData,0);
          CHECK4ERROR;    // Disable 5x5 filter
//      result = itf_clEnableTestImage(chDeviceData,0);
          CHECK4ERROR;    // Disable test mode
//      result = itf_clSetWindowing(chDeviceData,0,0,1600,1200);
   CHECK4ERROR;    // Force to full screen
```

```
//      result = itf_setHisto(chDeviceData,0);
                CHECK4ERROR;    // Disable histogram
//      result = itf_clSetNormalize(chDeviceData,0);
            CHECK4ERROR;    // Disable normalization
//      result = itf_setGrabMode(chDeviceData,T_NORMAL);
            CHECK4ERROR;    // Set Grab Mode to Normal
//      result = itf_setGrabTriggerDelay(chDeviceData,0);
            CHECK4ERROR;    // Disable Trigger Delay
//      result = itf_setExposureDelayOnTrigger(chDeviceData,0);
    CHECK4ERROR;    // Set Trigger Delay to 0
//      result = itf_setWhiteBalance(chDeviceData,T_GEN_DISABLE);
    CHECK4ERROR;    // Disable white balance
//      result = itf_setExposureMask(chDeviceData,T_GEN_DISABLE);
    CHECK4ERROR;    // Disable Exposure mask
//      itf_GainShutterValues Pars;
//      Pars.clamp          = 255;  // 0-255
//      Pars.gain           = 254;  // 0-600
//      Pars.shutter        = 7670; // 66-100000 microseconds
//      Pars.strobo         = 10;   // 10-1000 microseconds
//      Pars.trigger        = 1;    // 0-1 enable
//      Pars.triggerFront   = 0;    // 1 = rising edge, 0 = falling edge
//      result = itf_clSetMainPars(chDeviceData,&Pars);
            CHECK4ERROR;    // Setup clamp,gain,shutter & trigger

        // rti: reduce bandwidth
        /*result = itf_setGrabMode(chDeviceData,T_BANDWIDTH);
        CHECK4ERROR;
        result = itf_setGrabBandwidth(chDeviceData,5);
        CHECK4ERROR;
        */


        BOOL                            loop = TRUE;
        unsigned long           bytesRead=12;
        char*                           localImage;
        TAGLostImageInfos_t lostImage;
        int                                     noFrames=0;
        int                                     lostFrames=0;
        int                                     numFrames=0;
        int                                     offsetHeader = 40;

        //Information pictures definition
        long                            bpp;
        long                            pixel_type;
        string                  pixel_type_str;
        long                            height_im;
        long                            width_im;
        long                            size_buffer;

        //Definition color image
        int **                          redChannel;
        int **                          greenChannel;
        int **                          blueChannel;

        //Definition gray image
        char *                          grayImage;
        int**                           grayMatrix;
        int                                     temp = 0;

        //Mask for gradient
        char *                          gradientImage;
```

```
char *                          gradientImageColor;
int**                           gradientx;
int**                           gradienty;
int                                     win_size = 3;
int                                     gradientIndex = offsetHeader;

//Definition of copydata
char *                          copyImage;

//Time definition
DWORD time, time_1 = 0;
double mean, mean_1 = 0;


// Send command to start device (only continuous mode.
//result = itf_clStartStopDevice(chDeviceData, 1);


printf("\n--- Test results ------------------------------------\n");

// Get and print test starting date/time
GetSystemTime(&timeS);
printf("Test start on %d/%d/%d - %.2d:%.2d:%.2d\n",
        timeS.wYear, timeS.wMonth, timeS.wDay, timeS.wHour , timeS.wMinute , timeS.
            wSecond);

printf("Frame\tLen\t\tLost\tNoData\tAddr\n");

// Loop until a key is pressed
char filename[100];
long dx,dy;

while( !_kbhit() )
{
        result = itf_clStartSnapshot(chDeviceData); //instead of continous
            acquisition.
        if(result != 0)
        {
                printf("itf_clStartSnapshot failed\n");
                result = itf_clStartStopDevice(chDeviceData, 0);
                itf_Close(&chDeviceData);
                result = TAGClose(hFilter);
                _getch();
            return 0;
        }
        // Read image
        result=itf_clReadImage(hdata, (void**)&localImage);

        // Show results
        switch (result)
        {
                case ITF_READ_NO_DATA:
                        noFrames++;
                        break;
                case ITF_FRAME_LOST:
                        lostFrames++;
                        break;
                case ITF_OK:
                        numFrames++;
                        // received ok
                        sprintf(filename,"img%02d.bmp",numFrames);
```

```
printf("\nWriting Image: %s\n", filename);
TIL_image_to_disk(filename,(Tbanco*)localImage);

//
    ////////////////////////////////////////////////////////////////////

//////////////////////// Informartion Images
    ////////////////////////////////////
//
    ////////////////////////////////////////////////////////////////////


//Information frame rate
//Put a counter to know the elaps time between the last time
     and now
time = GetTickCount();
if (numFrames > 1)
{
        cout << "Time elapsed: " << time - time_1 << endl;
        double rate = time - time_1;
        mean = mean_1 + (double(time - time_1) - mean_1)/(
            numFrames - 1);
        mean_1 = mean;
        cout << "Mean frame rate: " << (1000/mean) << "img/s
            " << endl;
        cout << "Moment frame rate: " << (1000.00/(rate)) <<
             "img/s" << endl;
}

time_1 = time;

//Features images
//Find pixel type
TIL_get_image_pixel_type(&pixel_type, (Tbanco*)localImage);
switch(pixel_type)
{
        case 0:
                pixel_type_str = "GRAY_8";
                break;
        case 1:
                pixel_type_str = "RGB_565";
                break;
        case 2:
                pixel_type_str = "RGB_323";
                break;
        case 3:
                pixel_type_str = "RGB_888";
                break;
        case 4:
                pixel_type_str = "BAYER";
                break;
        case 5:
                pixel_type_str = "MOSAIC";
                break;
        case 6:
                pixel_type_str = "YCBCR420";
                break;
        case 7:
                pixel_type_str = "BGR888";
                break;
        case 8:
```

```
                pixel_type_str = "CRYCBY422";
                break;
        case 9:
                pixel_type_str = "RGBY_32BIT";
                break;
}
cout << "Pixel type: " << pixel_type_str << endl;

//Bits per pixel
TIL_get_image_bits_pixel(&bpp,(Tbanco*)localImage);
cout << "Bits per pixel: " << bpp << endl;

//Image dimensions
TIL_get_image_width(&width_im,(Tbanco*)localImage);
TIL_get_image_height(&height_im,(Tbanco*)localImage);
cout << "Width image: " << width_im << endl;
cout << "Height image: " << height_im << endl;

//Stream size image
TIL_get_image_stream_size(&size_buffer,(Tbanco*)localImage);
cout << "Buffer size: " << size_buffer << endl;

//
    ////////////////////////////////////////////////////////////

//// Image Processing
    ////////////////////////////////////////
//
    ////////////////////////////////////////////////////////////


//Allocation data
grayImage = (char*) itf_Malloc(3*width_im*height_im +
    offsetHeader);
gradientImage = (char*) itf_Malloc(3*width_im*height_im - 4
    + offsetHeader);
gradientImageColor = (char*) itf_Malloc(3*width_im*height_im
     - 4 + offsetHeader);
grayMatrix = new int* [height_im];
for(int i = 0 ; i < height_im ; i++)
{
        grayMatrix[i] = new int[width_im];
}
redChannel = new int* [height_im];
for(int i = 0 ; i < height_im ; i++)
{
        redChannel[i] = new int[width_im];
}
greenChannel = new int* [height_im];
for(int i = 0 ; i < height_im ; i++)
{
        greenChannel[i] = new int[width_im];
}
blueChannel = new int* [height_im];
for(int i = 0 ; i < height_im ; i++)
{
        blueChannel[i] = new int[width_im];
}

//Create mask gradient
gradientx = new int* [win_size];
```

```
for (int i = 0 ; i < win_size ; i++)
{
        gradientx[i] = new int[win_size];
}
gradientx[0][0] = -1;
gradientx[1][0] = -2;
gradientx[2][0] = -1;
gradientx[0][2] = 1;
gradientx[1][2] = 2;
gradientx[2][2] = 1;
gradientx[0][1] = 0;
gradientx[1][1] = 0;
gradientx[2][1] = 0;

//Create mask gradient
gradienty = new int* [win_size];
for (int i = 0 ; i < win_size ; i++)
{
        gradienty[i] = new int[win_size];
}
gradienty[0][0] = 1;
gradienty[1][0] = 0;
gradienty[2][0] = -1;
gradienty[0][2] = 1;
gradienty[1][2] = 0;
gradienty[2][2] = -1;
gradienty[0][1] = 2;
gradienty[1][1] = 0;
gradienty[2][1] = -2;

//Copy the header for image copy
for (int i = 0 ; i < offsetHeader ; i++)
{
        grayImage[i] = localImage[i];
}
for (int i = 0 ; i < offsetHeader ; i++)
{
        gradientImage[i] = localImage[i];
}
for (int i = 0 ; i < offsetHeader ; i++)
{
        gradientImageColor[i] = localImage[i];
}

switch(pixel_type)
{
case 1:
        break;
case 3:

        //Find the streaming
        UBYTE * pixel_localImage;
        TIL_get_image_stream_ptr(&pixel_localImage,(Tbanco*)
            localImage);
        UBYTE * pixel_grayImage;
        TIL_get_image_stream_ptr(&pixel_grayImage,(Tbanco*)
            grayImage);
        UBYTE * pixel_gradientImage;
        TIL_get_image_stream_ptr(&pixel_gradientImage,(
            Tbanco*)gradientImage);
        UBYTE * pixel_gradientImageColor;
```

```
                TIL_get_image_stream_ptr(&pixel_gradientImageColor,(
                    Tbanco*)gradientImageColor);

                //Convert RGB to grayscale
                rgb2gray(pixel_localImage, pixel_grayImage,
                    size_buffer);

                //Create the gray image in gray matrix
                vec2mat(pixel_grayImage, grayMatrix, height_im,
                    width_im, GRAY);

                //Create a matrix
                vec2mat(pixel_localImage, redChannel, height_im,
                    width_im, RED);

                //Create a matrix
                vec2mat(pixel_localImage, greenChannel, height_im,
                    width_im, GREEN);

                //Create a matrix
                vec2mat(pixel_localImage, blueChannel, height_im,
                    width_im, BLUE);

                //Computation of the gradient
                computeGradient(gradientx, gradienty, grayMatrix,
                    pixel_gradientImage, height_im, width_im);

                //Computation of the gradient
                computeGradientColor(gradientx, gradienty,
                    redChannel, greenChannel, blueChannel,
                    pixel_gradientImageColor, height_im, width_im);

                break;
        }

        sprintf(filename,"grayimg%02d.bmp",numFrames);
        TIL_image_to_disk(filename,(Tbanco*)grayImage);
        cout << "Saved Gray Image: " << endl;

        sprintf(filename,"gradimg%02d.bmp",numFrames);
        TIL_image_to_disk(filename,(Tbanco*)gradientImage);
        cout << "Saved Gray Image: " << endl;

        sprintf(filename,"gradimgcol%02d.bmp",numFrames);
        TIL_image_to_disk(filename,(Tbanco*)gradientImageColor);
        cout << "Saved Gray Image: " << endl;

        //Desallocation
        for(int i = 0 ; i < height_im ; i++)
        {
                delete []grayMatrix[i];
        }
        delete []grayMatrix;

        for (int i = 0 ; i < win_size ; i++)
        {
                delete []gradientx[i];
        }
        delete []gradientx;

        for (int i = 0 ; i < win_size ; i++)
```

```
                        {
                                delete []gradienty[i];
                        }
                        delete []gradienty;

                        for(int i = 0 ; i < height_im ; i++)
                        {
                                delete []redChannel[i];
                        }
                        delete []redChannel;

                        for(int i = 0 ; i < height_im ; i++)
                        {
                                delete []greenChannel[i];
                        }
                        delete []greenChannel;

                        for(int i = 0 ; i < height_im ; i++)
                        {
                                delete []blueChannel[i];
                        }
                        delete []blueChannel;

                        itf_Free(grayImage);
                        itf_Free(localImage);
                        itf_Free(gradientImage);
                        itf_Free(gradientImageColor);


                        break;
                default:
                        printf("\nUnknown Result=%d\n", result);
                        break;
        }

        printf("\r%d\t%d\t\t%d\t%d\t0x%x              ",
                numFrames,
                bytesRead,
                lostFrames,
                noFrames,
                (int)localImage);

}
_getch();

// Get test ending date/time
GetSystemTime(&timeE);
printf("\n\nStopping Live on TAG\n");
// Send command to stop device
result = itf_clStartStopDevice(chDeviceData, 0);

printf("Wait a Little\n");
Sleep(1000);

// Close communication channel
printf("Close command channel on TAG\n");
itf_Close(&chDeviceData); // close the data channel

printf("Call hdata()\n");
result = itf_Close(&hdata); // close the live channel
```

```
        // Print test starting/ending date/time
        printf("\n\n\n");
        printf("Test start on %d/%d/%d - %.2d:%.2d:%.2d\n",
                timeS.wYear, timeS.wMonth, timeS.wDay, timeS.wHour , timeS.wMinute , timeS.
                    wSecond);
        printf("Test end   on %d/%d/%d - %.2d:%.2d:%.2d\n",
                timeE.wYear, timeE.wMonth, timeE.wDay, timeE.wHour , timeE.wMinute , timeE.
                    wSecond);

        _getch();
        return 0;
}
```