TP Prise en main OpenCV – Calibration de caméra

Date : 22/11/2021
Durée : 4h (13h45-18h)
Objectifs : familiarisation avec OpenCV ; réalisation d'une calibration de caméra et redressement d'images

1- Préambule

OpenCV est une bibliothèque Open Source de vision par ordinateur très utilisée dans les domaines académiques et même dans l'industrie. Elle permet en effet d'utiliser de nombreux méthodes et algorithmes à la pointe dans les domaines tels que :

- le traitement d'images ;
- l'analyse vidéo ;
- la détection d'objets et/ou de points d'intérêts ;
- etc.

Nous vous proposons un document expliquant comment créer un projet Visual Studio 2019 pour OpenCV, voir document PDF correspondant et disponible sur le serveur pédagogique dans le dossier VSION.

OpenCV est une bibliothèque découpée en plusieurs modules, dans la séance d'aujourd'hui, nous allons utiliser les modules suivants :

- ***core*** : le module de base d'OpenCV contenant les classes de base (cv::Mat, etc.) ;
- ***highgui*** : module contenant les classes des interfaces utilisateurs (fenêtre, gestion clavier et souris, etc.) ;
- ***imgproc*** : module de traitement d'image (par exemple conversion d'une image couleur vers niveaux de gris, etc.) ;
- ***calib3d*** : module de calibration 3D et stéréoscopique (détection d'un échiquier, etc.).

Dans la suite de ce TP, nous allons utiliser **l'API** (Interface de Programmation) **C++** d'OpenCV, et non l'API C. Dans le cours, les captures d'écran prises à partir du chapitre du livre : « *Learning OpenCV – The Computer Vision library* » (également disponible sur le serveur pédagogique) utilisent l'API C et vous devrez donc être attentifs à bien utiliser les nouvelles versions des méthodes présentées.

Enfin, nous vous rappelons l'importance d'aller lire et regarder la documentation (très fournie) d'OpenCV afin d'obtenir les détails sur toutes les méthodes que vous allez utiliser, les différents paramètres et trouver des exemples d'utilisation de ces méthodes.

La documentation OpenCV pour la version installée sur vos machines est disponible à l'adresse suivante :

http://docs.opencv.org/index.html

2- Prise en main d'OpenCV

Dans cette partie, nous allons vous familiariser avec l'utilisation d'OpenCV. Pour ce faire, nous allons vous faire écrire une application qui va :

- afficher les images issues d'une caméra ou de différents fichiers en boucle jusqu'à ce que l'utilisateur appuie sur la touche '**echap**' ;
- convertir une image couleur en niveaux de gris ;
- offrir la possibilité à l'utilisateur de choisir entre l'affichage de l'image en couleur ou l'image en niveaux de gris ;
- libérer proprement les ressources avant de quitter votre programme.

2.1. Squelette de l'application

En reprenant le document expliquant comment créer un projet Visual Studio 2012 pour OpenCV et choisissez un nom adéquat (par exemple **OpenCV-Calibration**). Rajoutez ensuite un nouveau fichier `.cpp` à votre projet.

Commencez par inclure les entêtes OpenCV que nous allons utiliser :

```
// OpenCV core module (matrices, etc.)
#include <opencv2\core\core.hpp>

// OpenCV highgui: user interface, windows, etc.
#include <opencv2\highgui\highgui.hpp>

// OpenCV image processing (converting to grayscale, etc.)
#include <opencv2\imgproc\imgproc.hpp>
```

Nous allons également utiliser l'entête classique d'entrée/sortie C++ (`iostream`) et les espaces de nommages (namespace `cv` et `std`).

Vous aurez également besoin de définir une constante représentant le code ASCII de la touche '**Echap**' du clavier, que nous voulons pouvoir reconnaitre pour quitter l'application :

```
// the ASCCI code for the escape key
#define ESC_KEY 27
```

Voici (normalement) les codes à utiliser dans un swicth/Case retournés par la fonction cv ::waitKey sous Windows :

Touche	Valeur du « switch »
ESCAPE/Echap	27
Q	'Q'
U	'U'
SPACE/ESPACE	32

Bien que ce soit « moche », nous allons déclarer des variables globales dans ce TP car nous allons seulement réaliser un prototype et que ce n'est pas une vraie application. Nous pourrions également définir toutes nos variables au début de notre fonction principale (`main`) mais afin de limiter le nombre de lignes de cette dernière, nous optons ici pour les variables globales (vous pouvez toutefois faire comme vous le souhaitez). Pour le moment, contentons-nous d'une variable globale qui va nous permettre de récupérer une touche pressée par l'utilisateur :

```
// Either we use global variables (ugly but OK for a prototype)
// or we declare some variables in the main (also ugly)
// for clarity sake and brevity of the main function we'll use
global variables (shame)

// A key that we use to store the user keyboard input
char key;
```

Définissez maintenant la méthode principale de notre application :

```
// The main function
int main(int argc, char** argv) {

	// Starting the code
	// blablabla

	// exiting the program
	return EXIT_SUCCESS;
}
```

2.2. Affichage du flux vidéo

Dans un premier temps, nous voulons pouvoir récupérer le flux d'une caméra vidéo (assurez-vous d'en avoir une connectée à votre ordinateur). Pour ce faire, nous allons utiliser la classe `VideoCapture` du module **Highgui** d'OpenCV.

Définissez la variable suivante :

```
// Declaring a VideoCapture object
// it can open a camera feed
// or a video file if there is a codec on the machine
VideoCapture cap;
```

Un objet de type `VideoCapture` peut ouvrir le flux d'une caméra connectée à l'ordinateur, ou bien ouvrir un fichier vidéo. Nous allons l'utiliser dans le premier cas de figure ici, mais vous pourrez essayer par vous-même d'ouvrir un fichier vidéo.

Pour ce faire, nous allons utiliser la méthode `open` dont la documentation est ici : http://docs.opencv.org/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture

Cette méthode prend en paramètre soit :

- une **chaine de caractères** (`std::string`) contenant le nom du fichier vidéo que l'on souhaite ouvrir ;
- un **entier** représentant l'identifiant du dispositif (caméra) que l'on souhaite ouvrir (en schématisant : indice de la ou des caméra(s) connectée(s), commençant à 0).

Vérifiez ensuite que la capture a pu être effectuée. Pour ce faire, utilisez la méthode **`cap.isOpened()`** retournant un booléen permettant de savoir si tout s'est bien passé.

Modifiez votre programme pour demander en boucle à l'utilisateur un numéro de caméra tant que la capture n'est pas ouverte (ou -1 pour arrêter complètement l'application).

Avant toute chose, nous souhaitons manipuler des images issues de la caméra. Pour ce faire, déclarez deux objets de type `Mat` qui permettent de stocker des images en OpenCV.

```
// The current image
// retrieved from the video capture
// or read from a file
Mat image;
// The image converted into grayscale (see if we use it)
Mat gray_image;
```

Nous allons aussi avoir besoin d'une fenêtre pour afficher les résultats, pour ce faire, nous allons utiliser la méthode `namedWindow` du module **highgui**, qui permet de créer une fenêtre, laquelle est identifiée par une chaîne de caractères représentant son titre :

```
// Creating a window to display the images
windowName = "OpenCV Calibration";
namedWindow(windowName, CV_WINDOW_AUTOSIZE);
```

La fenêtre ne doit être créée qu'une seule fois !

Pour afficher une image dans une fenêtre préalablement créée, on utilisera la méthode `imshow` du module **highgui**. Cette méthode prend en paramètre le nom de la fenêtre dans laquelle on veut afficher, et l'image que l'on souhaite afficher. Par exemple, (on supposera que `image` contient bel et bien une image) :

```
// Showing the image in the window
imshow(windowName,image);
```

Enfin, pour extraire une image de la capture, il suffit d'utiliser l'opérateur >> de la classe `VideoCapture` :

```
// Getting the new image from the camera
cap >> image;
```

NB : vous pouvez également utiliser la méthode `read` (à la place de l'opérateur >>) de la classe `VideoCapture` :

```
// Getting the new image from the camera
cap.read(image);
```

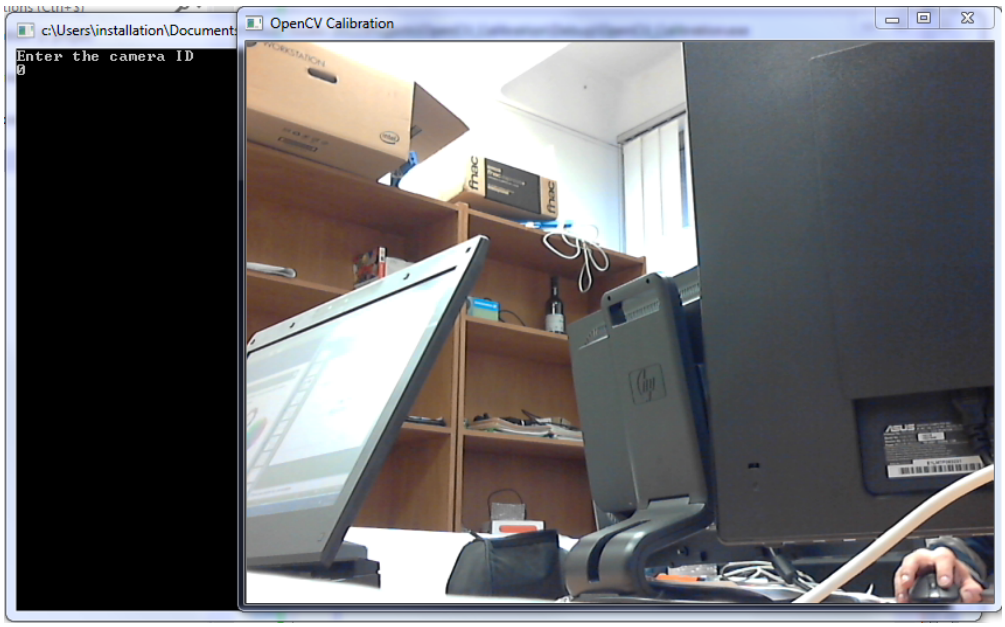
La dernière chose que vous devez mettre en œuvre est d'encapsuler la récupération d'une image, puis son affichage dans une boucle. Cette dernière doit s'arrêter lorsque l'utilisateur appuie sur la touche '**Echap**' de son clavier. Pour pouvoir récupérer les entrées clavier de l'utilisateur, nous allons utiliser une nouvelle méthode du module **highgui** : `waitKey`. Cette méthode prend comme argument un entier qui représente le délai pendant lequel la méthode va attendre un évènement utilisateur. **Si cet entier est négatif ou nul**, alors la méthode est bloquante, et `waitKey` attend indéfiniment pour l'évènement.

Enfin, afin de libérer les ressources que vous avez allouées pendant la création de cette application, n'oubliez pas d'appeler les méthodes suivantes, pour libérer successivement les fenêtres que vous avez créées, et la capture vidéo :

```
// Destroying the windows
destroyWindow(windowName);

// Releasing the video capture
cap.release();
```

Vous devriez maintenant avoir le résultat suivant (appelez-nous pour nous montrer le résultat) :



Nous allons rajouter une dernière étape à notre affichage de flux vidéo : la possibilité de transformer notre image en niveaux de gris, et de choisir si l'on souhaite afficher l'image en couleurs ou en niveaux de gris.

La conversion d'une image de couleur en niveaux de gris est très simple en OpenCV, il suffit d'utiliser la méthode `cvtColor` du module **imgproc**. Cette méthode prend en entrée l'image à convertir, l'image résultante et un paramètre déterminant les différentes options de conversion (voir la documentation pour plus de détails).

Ajoutez dans votre code la ligne qui permet de convertir la matrice `image` en une matrice `gray_image` en niveaux de gris.

Enfin, nous voulons que lorsque l'utilisateur appuie sur la touche **'g'** alors on affiche l'image en niveaux de gris. Si l'on appuie une nouvelle fois sur **'g'**, alors on réaffiche l'image en couleurs. Notez que la fonction `waitKey` retourne un caractère.

Appelez-nous pour nous montrer que tout fonctionne.

2.3. Affichage de fichiers (photos)

Dans cette deuxième partie, nous allons modifier notre programme afin d'afficher des fichiers image à la place d'un flux vidéo. Pour ce faire, vous utiliserez les images fournies sur le serveur pédagogique dans le dossier « calib_gopro ».

Pour lire un fichier image, nous allons utiliser la méthode `imread` du module **highgui**, que nous avons déjà vu dans le fichier expliquant la configuration de Visual Studio pour OpenCV.

Modifiez votre méthode main (pensez à le sauvegarder, ou créez un deuxième fichier .cpp) afin que votre boucle d'affichage du flux vidéo de la caméra, affiche en boucle les images contenues dans le dossier « calib_gopro ». Ces images sont toutes nommées **GOPR84**, suivi d'un numéro (précédé de 0 pour les numéros inférieurs à 10) puis possèdent l'extension **.JPG**. Il existe 27 images allant de GOPR8401.JPG à GOPR8427.JPG.

Pour cette partie, il se peut que vous deviez utiliser une méthode de conversion d'un nombre entier en chaine de caractères. Cela est faisable grâce à la méthode `to_string` de la bibliothèque standard de C++ (attention uniquement depuis la certification C++11 !) :

```
std::string msg = "bli";
int monEntier = 12;
bli += std::to_string(monEntier));
```

Afin d'illustrer l'intérêt de la calibration de caméra, vous devrez utiliser les photos du dossier **« calib_gopro »**, et les comparer avec les résultats que vous obtiendrez sur les autres fichiers d'exemples présents sur Hippocampus.

3- Calibration d'une caméra en OpenCV

Maintenant que vous avez pris en main OpenCV et utilisé quelques fonctions des principaux modules de cette bibliothèque, nous avons assez de connaissance pour réaliser la calibration d'une caméra.

3.1 Détection d'un échiquier

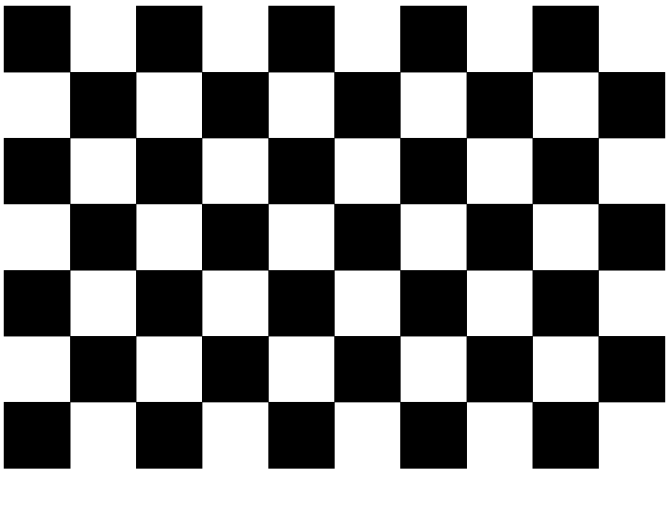
Comme présenté en cours, la calibration d'une caméra se fait en détectant un ensemble de points caractéristiques d'une cible particulière (les coins intérieurs d'un échiquier) puis en calculant les paramètres intrinsèques et la position et rotation des points caractéristiques. Cette détection se fait par le biais de la méthode `findChessboardCorners` du module **calib3d**.

Cette méthode prend les paramètres suivants :

- l'image dans laquelle nous allons tenter de trouver l'échiquier ;
- la taille (au format `cv::Size` du nombre de coins de l'échiquier à trouver en largeur et en largeur) ;
- un vecteur de `cv::Point2f` qui sont les points correspondants aux coins de l'échiquier trouvés dans l'image (ce vecteur est donc rempli par la fonction) ;
- un drapeau qui permet de spécifier le type de seuillage utilisé par la méthode.

Elle retourne un **booléen** qui dit si oui ou non un échiquier de la taille précisée a été détecté dans l'image.

L'exemple suivant d'échiquier est traditionnellement utilisé pour la calibration de caméra en OpenCV, notez que le nombre de coins intérieurs de cet échiquier, et donc requis par la méthode `findChessboardCorners` est de :
9 coins en largeur et de 6 en hauteur !



Voici un exemple d'appel à la méthode via l'API C++ :

```
cv::Size board_sz = cv::Size(numCornersHor, numCornersVer);
vector<cv::Point2f> corners;
bool found = findChessboardCorners(image, board_sz, corners,
CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);
```

Afin de vérifier la bonne détection de l'échiquier, nous allons utiliser la méthode `drawChessboardCorners` du module **calib3d** qui nous permet d'afficher les coins détectés dans une image.

Pour ce faire, lorsque l'échiquier a bel et bien été trouvé par `findChessboardCorners`, nous allons devoir affiner la détection des coins de l'échiquier. Ceci est réalisé en deux étapes :

1- la conversion de l'image couleur en image en niveaux de gris ;
2- le raffinage des positions des coins de l'échiquier en utilisant la méthode `cornerSubPix` du module **imgproc**. Cette étape n'est pas obligatoire, mais elle permet d'obtenir des meilleurs résultats de calibration.

Allez lire la documentation de la méthode `cornerSubPix`, afin de bien comprendre ce qu'elle fait. Voici un exemple d'utilisation de cette méthode :

```
cornerSubPix(gray_image, corners, cv::Size(11, 11), cv::Size(-1, -
1), cv::TermCriteria(CV_TERMCRIT_EPS | CV_TERMCRIT_ITER, 30, 0.1));
```

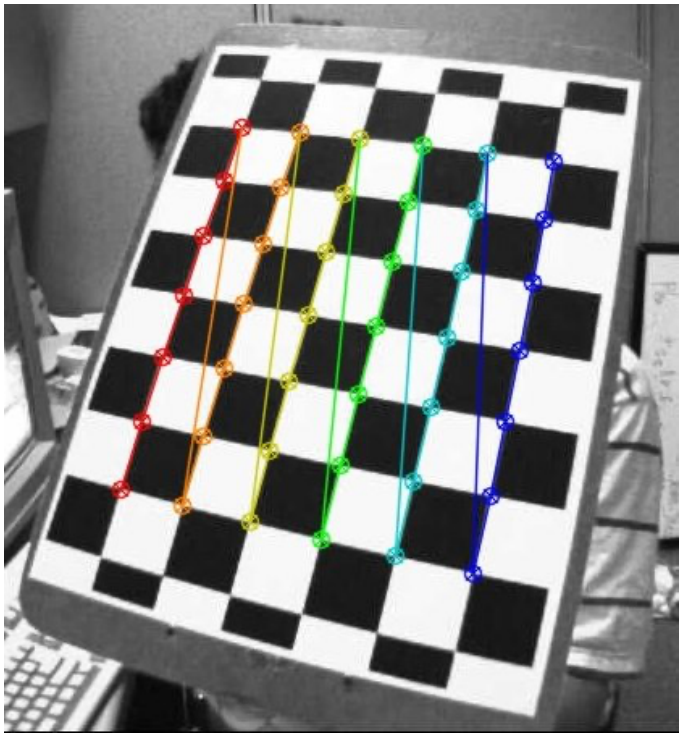
Une fois les coins raffinés, nous pouvons les afficher grâce à la méthode `drawChessboardCorners`, qui prend en paramètre :

- l'image dans laquelle on souhaite afficher les coins ;
- la taille de l'échiquier (en nombre de coins internes en largeur et hauteur sous forme de `cv::Size`) ;
- les coins de l'échiquier (dont les positions on éventuellement été raffinées par appel à la méthode `cornerSubPix`) ;
- un booléen indiquant si l'échiquier a été trouvé ou non (ce qui correspond au retour de la méthode `findChessboardCorners`).

Voici donc un appel à cette méthode pour dessiner les coins détectés dans l'image couleur utilisée pour détecter l'échiquier :

```
drawChessboardCorners(image, board_sz, corners, found);
```

Modifiez votre code pour que vous arriviez à détecter l'échiquier fourni dans le flux vidéo de votre caméra.



3.2 Calibration d'une caméra

Maintenant que nous avons détecté les coins internes de l'échiquier dans l'image obtenue par notre caméra, nous allons pouvoir réaliser la calibration. Ceci est effectué en appelant la méthode `calibrateCamera` du module **calib3d**.

Cette méthode fonctionne telle que présenté en cours.
Les paramètres de cette méthode sont les suivants :

- les **points objet** : ce sont des points 3D (i.e. `cv::Point3f`) (X, Y, Z) qui décrivent le patron recherché dans les images (ici notre échiquier). Sans perte de généralité, et comme on travaille sur des plans pour calibrer, cette grille de points correspondant simplement aux coordonnées des coins internes de l'échiquier (X,Y) et nous pouvons arbitrairement attribuer la valeur 0 pour Z. Ces points peuvent être calculés une fois pour toute car leurs coordonnées seront toujours les mêmes ;
- les points image : un ensemble de points 2D (i.e. `cv::Point2f`) correspondant aux coins détectés dans l'image donc détectés avec `findChessboardCorners` (et potentiellement raffinés par un appel à `cornerSubPix`) ;
- la taille de l'image utilisée pour la détection de l'échiquier ;
- la matrice de caméra (`cv::Mat`) : correspondant à la matrice intrinsèque de la caméra qui est calculée par cette fonction ;
- les coefficients de distorsion (`cv::Mat`) de 4, 5 ou 8 éléments : calculés par la fonction ;
- un ensemble de matrices (`vector<cv::Mat>`) représentant les rotations estimées pour chaque vue du patron (i.e. de l'échiquier) ;
- un ensemble de matrices (`vector<cv::Mat>`) représentant les translations estimées pour chaque vue du patron (i.e. de l'échiquier).

Allez lire la documentation de la méthode !

Voici un exemple de déclarations de variables permettant l'appel de la méthode `calibrateCamera` pour une image et en supposant que vous avez bien remplis les **vecteurs** `object_points` et `image_points` :

```
// Enough images to calibrate
cv::Mat intrinsic = cv::Mat(3, 3, CV_32FC1);
cv::Mat distCoeffs;
vector<cv::Mat> rvecs;
vector<cv::Mat> tvecs;
vector<vector<cv::Point3f>> object_points;
vector<vector<cv::Point2f>> image_points;

intrinsic.ptr<float>(0)[0] = 1;
intrinsic.ptr<float>(1)[1] = 1;

calibrateCamera(object_points, image_points, image.size(),
intrinsic, distCoeffs, rvecs, tvecs);
```

NB : Notez que `object_points` et `image_points` contiennent respectivement les coordonnées des coins physiques (`vector<cv::Point3f>`) de l'échiquier et l'équivalent des coordonnées 2D de ces coins dans une image (`vector<cv::Point2f>`). Ainsi, si vous voulez utiliser plusieurs images pour pouvoir obtenir une meilleure calibration, nous allons stocker ces correspondances pour chacune des images que vous allez utiliser dans la calibration (d'où le vecteur de vecteur).

Modifiez votre code pour pouvoir réaliser la calibration de votre caméra et ainsi récupérer les matrices intrinsèques (`intrinsic`) et les coefficients de distorsion (`distCoeffs`) de la caméra.

3.3 Redressement d'une image

Comme nous venons de calculer la matrice intrinsèque et les coefficients de distorsion de notre caméra, nous sommes maintenant capables de redresser une image. Pour ce faire, nous allons utiliser la méthode `undistort` du module **calib3d**.

Cette méthode prend en paramètres :

- l'image à redresser ;
- l'image redressée résultat ;
- la matrice intrinsèque de la caméra ;
- les coefficients de distorsion.

Déclarez une nouvelle matrice (`imageUndistorted`) pour stocker l'image redressée, puis faites appel à cette méthode :

```
undistort(image, imageUndistorted, intrinsic, distCoeffs);
```

Modifiez votre code pour afficher l'image redressée.

Faire une « vraie » application

Modifiez votre code pour pouvoir demander à l'utilisateur les informations suivantes :

- l'identifiant de la caméra utilisée ;
- le nombre de coin intérieur à l'échiquier dans la largeur ;
- le nombre de coin intérieur à l'échiquier dans la hauteur ;
- le nombre d'images à utiliser pour calculer la matrice intrinsèque et les coefficients de distorsion.

Ensuite modifiez votre application pour qu'elle corresponde à deux boucles consécutives :

1- une boucle de calibration dans laquelle l'utilisateur doit appuyer sur une touche (par exemple '**Espace**') pour choisir les images à garder pour la calibration. Le nombre d'images à collecter doit correspondre au nombre entré par l'utilisateur ;
2- une boucle d'affichage du résultat : image redressée ou image originale (suivant un choix de l'utilisateur, appui sur une touche).

4- Calibration d'une caméra en OpenCV

Modifiez votre code pour pouvoir utiliser comme source de calibration les images contenues dans le dossier « calib_gopro » au lieu de la caméra.

En effet, vous verrez que les caméras que nous possédons sont très bien calibrées, et que le résultat de vos calibrations sera probablement plus mauvais que l'image non redressée. Ce ne sera pas le cas avec la camera GoPro utilisée pour les images.

Vous pouvez aussi utiliser les autres images mises à votre disposition pour faire des comparaisons !