TP1 : Amélioration d'images

Objectif(s)

L'amélioration d'images consiste à modifier la dynamique du signal d'origine afin de renforcer l'information utile dans une image. Une des techniques les plus utilisées est le réhaussement du contraste qui s'effectue sur l'histogramme de l'image.

Exercice 1 : Prise en main de Visual Studio et OpenCV

Tous les TPs de ce module se feront en C++ sous Visual Studio 2017 et nous utiliserons la bibliothèque de traitement d'images OpenCV afin de ne pas coder tous les algorithmes que nous utiliserons (nous coderons tout de même certains algorithmes déjà présents afin de mieux les comprendre). Nous aborderons également quelques fonctions d'affichage et de gestion d'événements qui faciliteront la manipulation des images.

1. Ouvrez Visual Studio et créez un projet C++ vide ;
2. Ajoutez un nouveau fichier c++ appelé `main.cpp` et commencez à écrire le code ci-dessous. Passez en mode 64 bits (`x64`) plutôt que 32 bits (`x86`). Les touches `CTRL+ESPACE` vous permettent de compléter automatiquement les mots-clés saisis (ou les noms de fonctions / variables déjà définies). N'hésitez pas à en user / abuser afin d'utiliser des noms de fonctions / variables longues et explicites. Vous pouvez ensuite compiler votre code avec les touches `CTRL+F7` et lancer l'exécutable produit avec les touches `CTRL+F5`.

```cpp
#include <iostream>

using namespace std;

// ----------------------------------------------------------------------
// Programme principal
// ----------------------------------------------------------------------
int main(int argc, char** argv) {
    // Affichage de texte dans la console
    cout << "Hello World !" << endl;
    system("pause");
    return EXIT_SUCCESS;
}
```

3. Configurez maintenant votre projet pour pouvoir utiliser la bibliothèque OpenCV en 64bits. Pour cela, vous devez ouvrir le `Gestionnaire de Propriétés` disponible dans le menu `Affichage->Autres fenêtres` et attacher les 2 fichiers `.props` fournis (1 pour chaque configuration : `Debug` et `Release`).

4. Maintenant, vous êtes prêt à travailler en utilisant la bibliothèque OpenCV : chargez et affichez l'image `Rubik's_cube_L.png`. Aidez-vous de cette page de tutoriel pour comprendre comment utiliser la bibliothèque :
   `https://docs.opencv.org/4.3.0/db/deb/tutorial_display_image.html`
5. Voici ci-dessous le code de chargement et d'affichage avec quelques modifications par rapport au tutoriel.

```
#include <opencv2/highgui.hpp>
#include <iostream>
#include <string>

using namespace std;

// ----------------------------------------------------------------------
// Programme principal
// ----------------------------------------------------------------------
int main(int argc, char** argv) {
    // chargement de l'image
    string name_image_in = "Rubik's_cube_L.png";
    cv::Mat image_in = cv::imread(name_image_in, cv::ImreadModes::IMREAD_COLOR);
    if (image_in.empty()) {
        cout << "-> ERREUR : le fichier [" << name_image_in << "] n'existe pas." << endl;
        return EXIT_FAILURE;
    }
    // affichage de l'image
    cv::namedWindow("image_in", cv::WindowFlags::WINDOW_AUTOSIZE);
    cv::imshow("image_in", image_in);
    cv::waitKey(0);
    cv::destroyAllWindows();
    return EXIT_SUCCESS;
}
```

Ne pas déclarer l'utilisation de l'espace de nom `cv` en début de programme vous obligera à apposer `cv::` devant chaque fonction OpenCV à utiliser mais vous permettra de découvrir plus facilement les fonctions existantes (avec les touches `CTRL+ESPACE`). Même concept pour les constantes d'OpenCV, ça vous permettra de découvrir ce qu'il existe d'autres.

Vous allez également pouvoir afficher une aide ponctuelle sur les différents paramètres de fonctions à saisir en frappant les touches `CTRL+MAJ+ESPACE` entre les parenthèses de la fonction à compléter (testez sur la fonction `imread()`).

Enfin, vous pouvez afficher les dépendances entre fichiers en faisant un `clic-droit` sur votre code puis en sélectionnant le menu `Générer le graphique des fichiers Include` : Pourquoi n'a-t-on inclus qu'un seul fichier de la bibliothèque OpenCV dans

le code ci-dessus ? Peut-on encore supprimer un fichier de la liste des `#include` ? Si oui, lequel ?

6. Convertissez l'image en niveau de gris et affichez-la. Pour cela, utilisez la fonction `cvtColor()` d'OpenCV. Tapez le nom de la fonction dans le champ `Search` de la doc d'OpenCV afin d'obtenir la documentation adéquate (faîtes de même pour toutes les fonctions évoquées dans la suite de ce TP) :
`https://docs.opencv.org/4.3.0/`
7. OpenCV dispose d'un gestionnaire d'événements permettant, par exemple, de capturer les clics de souris dans les fenêtres d'affichage. Grâce à la fonction `setMouseCallback()`, vous allez pouvoir appeler une fonction (à coder vous-même) permettant d'afficher dans la console la valeur du pixel pointé par la souris lorsque vous cliquez sur une image.

   Codez la fonction d'affichage de la valeur du pixel pointé par la souris (sur une image en niveau de gris), appelez cette fonction via `setMouseCallback()` et testez le tout sur l'image convertie en niveau de gris. L'accès à un pixel de l'image se fait par la fonction `at()` : utilisez toutes les infos fournies par la doc sur ces 2 fonctions afin de comprendre comment effectuer cela.
8. Modifiez à présent la fonction d'affichage de la valeur du pixel pointé par la souris afin qu'elle puisse être appelée aussi bien sur une image en niveau de gris que sur une image couleur. Appelez cette fonction via 2 `setMouseCallback()` afin de tester simultanément sur l'image couleur et sur l'image en niveau de gris. Vous pouvez récupérer le type de l'image (couleur ou niveau de gris) grâce à la fonction `type()`. Vous pouvez vous aider du tutoriel suivant pour l'accès aux pixels d'une image couleur :
`https://docs.opencv.org/4.3.0/d5/d98/tutorial_mat_operations.html`

Exercice 2 : Histogramme 1D sur les niveaux de gris

Nous allons à présent construire, visualiser et interpréter des histogrammes sur des images en niveaux de gris. Vous pouvez désactiver l'affichage de l'image couleur.

1. Ecrivez le code pour calculer l'histogramme de l'image en niveau de gris (pour cela, reprenez la formule de votre cours). Vous pouvez construire et initialiser votre histogramme par la fonction `zeros()`. Affichez ensuite les résultats de l'histogramme sous forme textuelle dans un premier temps.
2. Utilisez à présent la fonction `calcHist()` pour calculer l'histogramme en niveau de gris. Attention, les paramètres de cette fonction nécessitent de bien réfléchir au fonctionnement de cette fonction (d'où son codage en amont pour mieux cerner ce que l'on manipule). Affichez toujours les résultats de l'histogramme sous forme textuelle pour vérifier que votre code de la question précédente était correct.
3. Passez à présent à un affichage sous forme de graphique en bâtons de l'histogramme des niveaux de gris. Pour cela, générez une image de taille `256x256`. Vous pouvez utiliser la

fonction `minMaxLoc()` afin de récupérer le plus grand pic de l'histogramme et le ramener à la hauteur max de l'image afin d'afficher l'histogramme avec la plus grande amplitude possible. Vous pouvez utiliser la fonction `cvRound()` afin d'arrondir un flottant à l'entier le plus proche. La fonction `flip()` peut également vous être utile si votre histogramme n'est pas affiché dans le bon sens.

Que constatez-vous sur cet histogramme ?

4. Mettez en place la fonction `stretchHist()` permettant d'effectuer un `étirement d'histogramme`. Reprenez votre cours pour vérifier comment faire (celle-ci doit recalculer tous les pixels de l'image en niveaux de gris en étirant l'histogramme). Ensuite, pensez à raffraichir son affichage en relançant la fonction `imshow()`. Vous devez également recalculer l'histogramme sur cette nouvelle image afin de vérifier visuellement que l'histogramme a été étiré correctement. La reconstruction du graphique en bâtons de l'histogramme vous oblige à dupliquer le code => comme il n'est jamais bon de dupliquer son code, préférez écrire une fonction de construction du graphique en bâtons qui prend en paramètres l'histogramme de l'image en niveaux de gris. Utilisez cette fonction aux deux endroits de votre programme.

   Etes-vous satisfait du résultat visuel ? Est-il facile à interpréter ? Qu'est-ce qui pose problème à votre avis ?

5. La fonction `calcHist()` peut prendre un masque en paramètres afin de ne calculer l'histogramme que sur la partie non masquée. Que pourriez-vous inclure dans le masque ? Partez de plusieurs coins de l'image pour inclure tous les pixels de même intensité par un algorithme de segmentation par agrégation appelé `croissance de germe`. Pour cela, utilisez la fonction `floodFill()` en amont afin de construire le masque. Attention à la taille du masque qui doit être utilisé dans les fonctions `floodFill()` et `calcHist()`. L'utilisation de la fonction `adjustROI()` vous sera utile. Vous pouvez vérifier le contenu de votre masque en l'affichant comme une image classique.

   Intégrez ce masque dans tous vos calculs d'histogrammes afin de constater les changements. Y-a-t'il un impact sur le code d'étirement d'histogramme de la question précédente ? Si oui, corrigez le code afin de prendre en compte le masque lors de l'étirement d'histogramme (potentiellement, un bug s'y est glissé !). De quelles options disposons-nous pour prendre en compte ce masque ?

   Interprétez les différents pics visibles dans l'histogramme. Pourquoi un pic apparaît très étroit dans l'histogramme ? Comment interpréter la hauteur de ce pic par rapport à d'autres régions de l'image qui semblent aussi grandes ? Vous pouvez utiliser votre fonction de sélection de pixel sur l'histogramme afin de vérifier quelle valeur d'intensité est attribuée à chacun des pics présents (axe des x) et le nb relatif de pixels (axe des y).

6. Vous allez à présent effectuer une `égalisation d'histogramme` afin de voir quelles sont les différences par rapport à un `étirement d'histogramme`. Il existe la fonction `equalizeHist()` qui effectue celà à votre place. Testez-là, générez l'image et le graphique en bâtons de l'histogramme égalisés. Calculez et affichez également l'histogramme cumulé afin de vérifier qu'il est bien linéaire (écrire la fonction `calcHistCum()`).

Que pouvez-vous dire sur l'image ? sur l'histogramme ? et sur l'histogramme cumulé ?

7. Recodez vous-même une fonction effectuant une `égalisation d'histogramme` afin de palier les défauts constatés par la méthode originale. Reprenez votre cours pour vérifier comment faire (celle-ci doit recalculer tous les pixels de l'image en niveaux de gris en égalisant l'histogramme). Générez l'image et les graphiques en bâtons des histogrammes égalisé et cumulé.
   Cela a-t-il permis de corriger les défauts constatés précédemment ?

Exercice 3 : Balance des blancs sur les canaux BGR

Nous allons à présent travailler sur l'image en couleur. Vous pouvez mettre une pause après l'affichage des histogrammes de l'exercice 2 avec la fonction `waitKey()` puis supprimer toutes les fenêtres affichées avec la fonction `destroyAllWindows()` et enfin réactiver l'affichage de l'image couleur juste après afin de commencer sereinement l'exercice 3.
Nous allons voir comment effectuer une `balance des blancs` sur une image déjà acquise afin de retranscrire plus fidèlement les couleurs des objets composant la scène (suppression de la couleur de l'éclairage).

1. L'image affichée montre un arrière-plan légèrement bleuté, teinté par la couleur de l'éclairage (éclairage froid utilisé). Nous savons que l'arrière-plan est normalement sans teinte (gris). Si la `balance des blancs` avait été correctement effectuée lors de l'acquisition de l'image, nous aurions eu un arrière-plan complètement gris. Vous allez donc coder la fonction `applyWhiteBalance()` qui prend en paramètre l'image couleur, une zone rectangulaire significative de l'arrière-plan (genre `200x200`) et qui va transformer les couleurs de l'image afin de générer un arrière-plan le plus gris possible : pour cela il va falloir, au préalable, calculer la moyenne des couleurs de la zone rectangulaire (en utilisant la fonction `mean()`) puis modifier les 3 canaux afin d'obtenir le résultat escompté. Il existe beaucoup de méthodes d'application de la `balance des blancs`, ici nous allons coder une méthode simple basée sur une transformation linéaire des trois canaux.
   Attention à ne pas changer la `clarté` globale de votre image : pour cela, vous devez transformer votre zone rectangulaire en niveau de gris et estimer le ton de gris moyen de l'arrière-plan : cette valeur sera utilisée comme référence à atteindre pour les 3 canaux.
   Attention également à vérifier la `saturation` éventuelle de certains pixels lors de vos modifications de canaux : si `saturation` il y a, il faudra prévenir l'utilisateur des canaux saturés (envoi sur la console d'erreur via `cerr`) et utiliser la fonction `saturate_cast()` pour produire votre image finale. Pour cela, vous devrez au préalable séparer vos canaux via la fonction `split()`, rechercher le pixel de valeur max via la fonction `minmaxLoc()` et tester si ce pixel peut saturer après application de la `balance des blancs` (test sur chaque canal, indépendamment).

   Voici quelques explications complémentaires sur l'utilisation de la `balance des blancs` en photographie (`balance des blancs`, `balance des noirs`, `charte de gris neutre`,

`charte des couleurs`, différentes transformations utilisées) :
`https://blog.photo24.fr/balance-des-blancs/`
`http://sellig.zed.myriapyle.net/Couleur/balablan.htm`

Que constatez-vous sur les couleurs du rubik's cube ? de la table ? et de l'arrière-plan ?

2. OpenCV dispose de la fonction `balanceWhite()` qui permet d'appliquer une `balance des blancs` sur une image. Regardez attentivement la doc de cette fonction.
En quoi diffère-t-elle de celle que vous avez produite ? A votre avis, est-elle aussi efficace que la vôtre ? Pourquoi ?

Vous ne pouvez pas utiliser cette fonction car elle ne fait pas partie des fonctions de base installées. Recodez cette fonction en prenant les mêmes précautions que précédemment : ne pas changer la `clarté` globale de votre image et vérifier la `saturation` éventuelle des pixels modifiés.
Quelle méthode est la plus réaliste dans notre cas ? Est-ce conforme à vos attentes ?

3. Généralement, afin de ne pas biaiser le calcul de la `balance des blancs`, nous écartons les pixels proches de la `saturation` (pixels sur-exposés ou sous-exposés : il n'y en a pas dans la zone significative de l'arrière-plan, mais dans l'image globale ...). Modifiez l'appel de la fonction `balanceWhite()` afin d'écarter les pixels trop saturés dans votre calcul de `balance des blancs` (prise en compte des valeurs uniquement dans l'intervalle `[25..230]` pour les 3 canaux).
Pour faciliter vos calculs, vous pouvez réaliser vos `seuillages` grâce à la fonction `threshold()`. Etudiez bien la documentation de cette fonction afin de choisir le type de `seuillage` approprié. Vous devrez ensuite séparer vos trois canaux seuillés en trois images distinctes grâce à la fonction `split()` et utiliser la fonction `min()` afin de définir un `masque` comprenant uniquement les pixels dont les trois canaux sont dans l'intervalle (correspondant aux pixels non saturés). Ce `masque` pourra enfin être utilisé dans la fonction de calcul de moyenne `mean()`.
L'éviction des pixels saturés permet-elle d'améliorer la première version de la fonction ? Est-ce suffisant ?

Exercice 4 : Histogrammes 1D sur les canaux BGR

Nous allons à présent construire, visualiser et interpréter des histogrammes sur des images en couleur. Nous repartons de l'image couleur après application de la `balance des blancs` sur un patch d'arrière-plan.

Calculer un histogramme sur les 3 canaux simultanément revient à construire un `histogramme 3D` et le visualiser n'est pas aisé. Chaque axe représente un canal couleur et chaque point 3D doit représenter le nombre de pixels de la couleur de coordonnées `(B, G, R)` : pour cela il faut, par exemple, afficher chaque point avec une intensité lumineuse qui correspond à la densité de

pixels en ce point 3D. Une autre façon d'afficher ces points 3D est de leur donner un diamètre qui dépend de la densité de pixels en ce point et afficher chaque point comme une sphère de bon diamètre avec la couleur du point 3D central `(B, G, R)`. Dans les deux cas de figure, il faut passer par une interface 3D, par exemple, en affichant des primitives via OpenGL et gérer la rotation de la caméra dans cet environnement 3D. Il y a donc un peu de travail si on n'utilise pas de bibliothèque dédiée à l'affichage.

1. Dans ce TP, nous n'allons pas gérer une interface OpenGL mais plutôt utiliser la technique classique de `superposition d'histogrammes 1D` (1 histogramme par canal). Pour cela, vous allez devoir séparer les 3 canaux de l'image couleur via la fonction `split()` et utiliser la fonction `calcHist()` pour calculer l'histogramme sur chaque canal couleur séparément (sans utiliser de masque pour le moment). Visualisez ces 3 histogrammes dans 3 fenêtres séparées (par 3 graphiques en bâtons).
   Pouvez-vous réutiliser la fonction de construction de graphique en bâtons d'un histogramme sur une image en niveaux de gris codée dans l'exercice précédent ? Pourquoi ? Que faut-il faire pour la lisibilité du code ?
2. Réactivez le `masque` d'arrière-plan afin de recalculer des histogrammes qui prennent en compte ce masque. Maintenant, vous disposez d'un affichage plus lisible afin d'exploiter les différents pics sur chaque histogramme.
   Interprétez les différents pics des histogrammes. Les pics larges sont-ils tous faciles à interpréter dans l'image ? Et les pics très étroits ?
3. Visualisez maintenant ces 3 histogrammes en 1 seul en affichant 1 histogramme global (1 seul graphique en bâtons) : pour cela, il suffit d'afficher les bâtons de la couleur de chaque canal (les 3 couleurs peuvent ainsi être superposées sur un même bâton). Cela donne un affichage comme représenté dans Lightroom ici :
   `https://blog.japprendslaphotographie.com/charte-gris-neutre-comment-lutiliser-ameliorer-ses-photos/`
   Réfléchissez bien à la valeur max que vous allez prendre en compte sur l'axe `Y` de l'histogramme global.
   Cette valeur max a-t-elle une influence sur l'interprétation des pics faite précédemment ? Les couleurs superposées aident-elles à interpréter les pics plus facilement ?
4. Maintenant, calculez et visualisez l'histogramme global (1 graphique en bâtons) de l'image originale en couleur (sans application de la balance de blancs). L'idée est d'afficher côte-à-côte les histogrammes globaux (avec utilisation du masque d'arrière-plan) de l'image originale en `couleur` et de l'image avec application de la `balance des blancs`.
   En quoi sont-ils différents ?
5. Effectuez maintenant un `étirement d'histogramme` (avec utilisation du `masque` d'arrière-plan) sur chaque canal, sans modifier la `chromaticité` des pixels de l'image. Pouvez-vous réutiliser la fonction `strechHist()` codée précédemment ? Pourquoi ?
   Appliquez cette méthode sur l'image originale `couleur` et sur l'image avec application de la `balance des blancs`.
   Obtenez-vous les deux mêmes images ? Est-ce normal ? Que constatez-vous sur l'histo-

gramme global ?

6. Effectuez maintenant une `égalisation d'histogramme` (avec utilisation du `masque` d'arrière-plan) sur chaque canal. Pouvez-vous réutiliser la fonction `equalizeHist()` codée précédemment ? Pourquoi ?
   Appliquez cette méthode sur l'image originale `couleur` et sur l'image avec application de la `balance des blancs`.
   Obtenez-vous les deux mêmes images ? Est-ce normal ? Que constatez-vous sur l'histogramme global ?
7. Même si l'image égaliseé obtenue n'est pas réaliste, elle pourrait être utilisée pour rapidement détecter les faces rouges, vertes et bleues du Rubik's cube par un simple `seuillage` car les inter-réflexions lumineuses avec la table sont estompées sur les faces du cube et les canaux de couleur sont plus saturés, avec un peu moins de variance.
   Affichez une image résultat du `seuillage` avec les 3 canaux simultanément. Un `seuillage binaire` sera effectué par canal en utilisant la fonction `threshold()`. Les valeurs de `seuillage` seront ajustables grâce à 3 sliders attachés à la fenêtre d'affichage de l'image résultat du `seuillage` : pour effectuer cela, vous utiliserez la fonction `createTrackbar()`. Vous devrez coder une fonction `onChange()` permettant de mettre à jour l'image résultat à chaque variation d'un des 3 sliders, comme dans l'exemple ci-dessous :

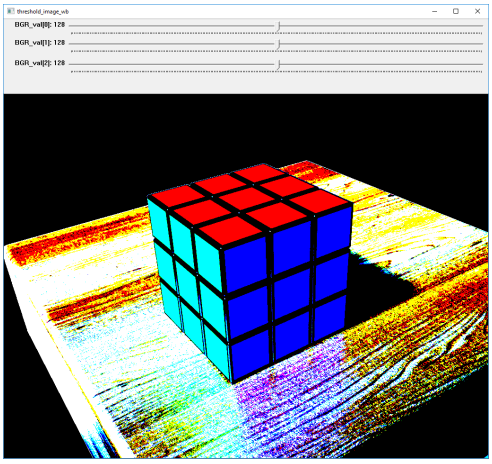

FIGURE 15 – Capture d'écran de l'image résultat du `seuillage binaire` simultané sur les 3 canaux de l'image `Rubik's_cube_L.png` après application de la `balance des blancs` sur un patch de l'arrière-plan.

Il faudra passer à la fonction `onChange()` :
- l'image à seuiller (afin de pouvoir seuiller un de ses canaux => séparation des canaux avec la fonction `split()`),
- l'image résultat du `seuillage` (afin de pouvoir mettre à jour l'affichage du `seuillage binaire` pour les trois canaux => fusion des canaux avec la fonction `merge()`)
- le `masque` d'arrière-plan (afin de ne pas le prendre en compte dans le `seuillage` => on peut appliquer un `masque` sur une image avec la fonction matricielle `mul()`).

Peut-être d'autres paramètres vous sembleront utiles (ce sera à vous de juger en codant la

fonction). Comme vous ne pourrez passer qu'une seule variable à la fonction `onChange()`, il faudra stocker l'ensemble des informations utiles dans une mini structure de données `struct{}`.

Arrivez-vous facilement à seuiller les 3 faces du Rubik's cube ? Testez également sur l'image étirée pour voir laquelle est la plus facile à seuiller.

Exercice 5 : Histogrammes 1D sur les canaux HLS

Nous allons à présent construire, visualiser et interpréter des `histogrammes` sur des images en couleur en séparant la `luminance` et la `chromaticité`. Divers `espaces de représentation des couleurs` existent pour séparer la `chromaticité` et la `luminance`, par exemple `L*a*b*`, `L*u*v*`, `HSV`, `HLS`, etc. Nous repartons de l'image couleur après application de la `balance des blancs` sur un patch d'arrière-plan.

1. Convertissez l'image dans l'espace colorimétrique `HLS` en utilisant la fonction `cvtColor()`. Que se passe-t-il si vous affichez l'image ainsi convertie ? Comment expliquez-vous ces changements de couleurs ?
   Séparez les 3 canaux de l'image `HLS` via la fonction `split()` et affichez les séparément (dans 3 fenêtres).
   Que remarquez-vous pour la table ?
2. Utilisez la fonction `calcHist()` pour calculer `l'histogramme` sur chaque canal séparément en prenant en compte le `masque` d'arrière-plan. Visualisez ces 3 `histogrammes` dans 3 fenêtres séparées (par 3 graphiques en bâtons).
   Pouvez-vous réutiliser la fonction de construction de graphique en bâtons d'un `histogramme` sur une image en niveaux de gris codée dans l'exercice précédent ? (regardez bien la documentation associée à la conversion dans le repère `HLS` pour justifier vos choix). Interprétez les pics des différents histogrammes. Retrouvez-vous l'information précédente sur la table ?
3. Effectuez à présent une `égalisation d'histogramme` uniquement sur le canal `L` de `HLS` (et uniquement sur le `masque` d'arrière-plan) afin de ne pas toucher à la `chromaticité` de l'image. Recomposez l'image obtenue avec la fonction `merge()` et visualisez-là.
   Que constatez-vous ?
4. Plusieurs pistes s'offrent à vous. Soit utiliser l'information de `teinte` pour séparer plus facilement les faces du Rubik's cube. Soit repérer la `teinte` de la table, en faire un `masque`, l'ajouter au `masque` d'arrière-plan afin de réutiliser le `seuillage binaire` en 3 canaux `BGR` de l'exercice précédant (ainsi la table ne sera plus un obstacle, les valeurs de seuils pourront être plus facilement ajustées).