MODULE 3

La classe Mat

Objectifs de ce module :

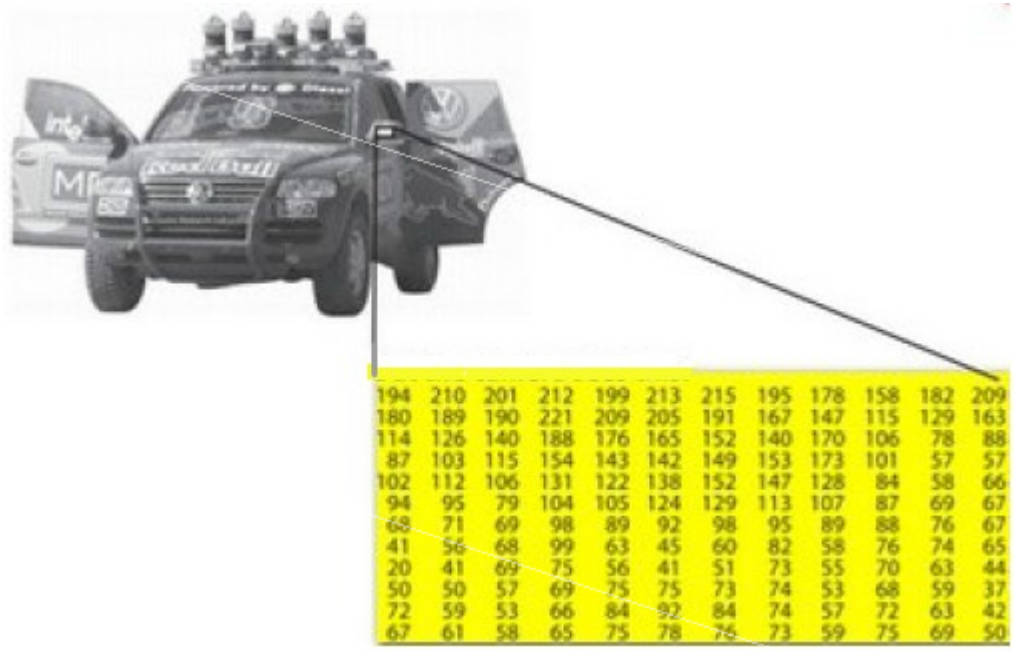
- ✓ ***Comprendre les éléments de la classe Mat***
- ✓ ***Stocker une image avec la classe Mat***
- ✓ ***Manipuler un objet de la classe Mat***

Table des matières

Sujets *Page*

Introduction

Il existe plusieurs façon de digitaliser une image du monde réel : caméras digitales, WebCam, scanners, rayon-x, sonar pour ne nommer que ceux-ci. Peu importe le moyen, ce nous voyons à la fin est une image. Cependant, ce que nous faisons vraiment lorsque nous enregistrons une image est d'enregistrer les valeurs numériques de chacun des points de l'image.



Par exemple, dans l'image précédente, ce que nous percevons comme le miroir de la voiture est en fait une matrice contenant les valeurs des intensités de chacun des pixels de cette portion de l'image. La question est vraiment de savoir comment emmagasiner ces valeurs pour pouvoir les traiter et les interpréter adéquatement. Au niveau informatique, une image sera toujours représenter par une matrice contenant les informations de cette matrice elle-même et les valeurs d'intensités des pixels. OpenCV est une bibliothèque de vision par ordinateur dont l'objectif est de rendre plus facile la manipulation et le traitement de ces informations.

La classe Mat

OpenCV est un projet qui a vu le jour vers les années 2000-2001. Cependant, la bibliothèque était utilisée, à cette époque, à l'aide du langage C. La structure en C nommée IplImage était fortement utilisée car elle représentait une image en mémoire (Plusieurs projets OpenCV utilisent cette structure encore aujourd'hui).

La plus grande difficulté était alors de faire la gestion efficace de la mémoire. En fait, on supposait que l'usager (le programmeur) était responsable d'allouer ou de vider la mémoire correctement. Bien que ce ne soit pas une difficulté en soit dans les programmes plus petit, le problème devient tout autre lorsque l'application engraisse et que, par conséquent, le code devient plus volumineux. Nous sommes alors pris avec le fait que nous passerons plus de temps à faire de la bonne gestion de mémoire que de nous concentrer sur l'objectif de résoudre le problème de vision réel.

Heureusement, le langage C++ nous a donné le concept de classe et les versions 2.0 ou supérieure d'OpenCV en font usage abondamment maintenant.

Les dessous de la classe Mat

La classe Mat permet de représenter une image en mémoire de même que les informations relatives à cette image. Mais les avantages sont plus nombreux encore :

- L'allocation des objets de type Mat est automatiquement réalisée selon la grosseur de l'image analysée.
- Le vidage de la mémoire est aussi automatique lorsque l'image n'est plus utilisée.
- L'opérateur d'assignation (=) et le constructeur de copie ont pour effet de copier uniquement l'en-tête de la matrice.

L'objet de la classe Mat apporte une différence majeure par rapport aux versions précédentes d'OpenCV. Vous n'avez plus besoin d'allouer manuellement l'espace mémoire ni de vider la mémoire allouée à un objet lorsque vous n'avez plus besoin de cet objet. Bien qu'il soit encore possible de gérer vous-même la mémoire, la plupart des fonctions d'OpenCV vont allouer automatiquement la mémoire. En plus, si vous passez en paramètre un objet déjà alloué en mémoire, celui-ci sera réutilisé. Autrement dit, la mémoire est utilisée de façon optimale.

Mat est une classe qui possède les informations concernant l'entête (la dimension (le nombre de colonne et de rangée, la méthode de rangement en mémoire et à quelle adresse les pixels sont situés en mémoire) ainsi qu'un pointeur sur les éléments qui contiennent les pixels eux-mêmes.

Lorsque vous copiez une image dans un autre objet de classe Mat, vous copiez évidemment tous les pixels de l'image originale dans un autre endroit en mémoire. Ceci est très couteux en terme d'utilisation de la mémoire surtout si les images sont grosses. Afin d'éviter un tel gaspillage de la mémoire, OpenCV n'effectue pas comme tel la copie des pixels originaux mais alloue plutôt une adresse qui pointe au même endroit que l'adresse de l'objet original. De plus, les opérateurs de copie ne copierons que l'entête pas les valeurs des pixels eux-mêmes.

Exemple:

```
Mat  A, C  // Créer seulement les entêtes
A = imread(argv[1], CV_LOAD_IMAGE_COLOR);
//Alloue la place en mémoire pour garder les valeurs des pixels et remplie l'entête.

Mat B(A)  // Constructeur de copie  Seulement l'entête de A est copié dans B.

C = A  // Opérateur d'assignation.  Là aussi, seulement l'entête est copié dans C.
```

Dans les exemples précédents, tous les objets pointent sur la même matrice de pixels. Cependant, bien qu'ils ont chacun un entête indépendant, une modification à la matrice de pixels entraînera une modification pour les autres également.

Vous vous demandez peut-être à quoi bon plusieurs objets mais qui pointe tous au même endroit sur la matrice de pixel. L'avantage devient clair lorsqu'on fait référence à un sous-ensemble de la matrice de pixel. Par exemple, pour créer une région d'intérêt (Region Of Interest en anglais), vous n'aurez qu'à créer un nouvel entête avec les valeurs :

```
Mat ImgDest (ImgSource, Rect(10,10,100,100) );
//On utilise un rectangle pour signifier la portion d'intérêt.

Mat  ImgDest = ImgSource(Range:all(),  Range(1,3));
// En utilisant les rangées et colonnes
```

Voyons un exemple pour nous convaincre de tout ça :

```
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main()
{
Mat ImgSource;

ImgSource = imread("Penguins.jpg", CV_LOAD_IMAGE_COLOR);

Mat ImgDest1(ImgSource);

Mat ImgDest2(ImgDest1);

Mat ImgDest3 = ImgDest2;

Mat ImgDest4 = ImgDest1(Range(100,200), Range(0,500));

Mat ImgDest5 = ImgSource(Rect(100, 300, 300, 300));

namedWindow("Img Source");
namedWindow("Img Destination 1");
namedWindow("Img Destination 2");
namedWindow("Img Destination 3");
namedWindow("Img Destination 4");
namedWindow("Img Destination 5");

imshow("Img Source", ImgSource);
imshow("Img Destination 1", ImgDest1);


imshow("Img Destination 2", ImgDest2);


imshow("Img Destination 3", ImgDest3);


imshow("Img Destination 4", ImgDest4);

imshow("Img Destination 5", ImgDest5);

cout << "Refcout: " << *(ImgSource.refcount) << "Adresse du premier pixel: " << hex << (int)ImgSource.data << endl;
waitKey(0);


return 0;
}
```

Dans l'exemple précédent, vous vous demandez peut-être qui sera en charge de vider la mémoire correctement lorsque celle-ci ne sera plus nécessaire. En fait, c'est le dernier objet à utiliser la mémoire qui sera en charge de le faire. À cet effet, il existe un élément dans la classe Mat qui garde un compte du nombre d'objet qui fait référence à la matrice de pixel. Cet élément se nomme « refcount ». Lorsqu'on fait une copie d'un objet Mat, ce compte est incrémenté et, à l'inverse, décrémenté lorsqu'on n'utilise plus cet objet. Lorsque ce compteur arrive à zéro, la mémoire qui contenait la matrice de pixel est vidée.

Regardons ce qui se passe dans l'exemple précédent :

La ligne 8 assigne un entête de base à l'image source.
La ligne 9 charge le fichier avec la fonction imread et remplit les informations de l'entête.

Les lignes suivantes ne font que créer un entête spécifique à l'objet mais font pointer à la même adresse de la matrice de pixel.

Par exemple, la ligne 11 : Mat ImgDest2(ImgDest1);
Utilise le constructeur de copie. Le compte de « refcount » passe à 3 puisqu'en plus de l'image source, il y avait aussi l'objet ImgDest1 qui faisait passer le « refcount » à 2.

La ligne 13 : Mat ImgDest4 = ImgDest1(Range(100,200), Range(0,500));
Permet de définir une portion de l'image ImgDest1 dont les rangées vont de 100 à 200 et les colonnes de 0 à 500.

La ligne 14 : Mat ImgDest5 = ImgSource(Rect(100,300,300,300));

Permet de définir une Image (ImgDest5) en fonction d'une région de l'image source (ImgSource). Les deux premières valeurs (100,300) spécifie le premier pixel en prendre en considération alors que la paire suivante (300,300) signifie le nombre de pixel en hauteur et en largeur. On a donc, à la fin, un rectangle dont le point supérieur gauche débute à (100, 300) et dont le dernier pixel en bas à droite est à 300 de plus en x et à 300 de plus en y, c'est-à-dire, à la coordonnée (400, 600).

Cependant, il est possible de copier la matrice de pixel dans un autre objet de type Mat. Pour ce faire, il existe 2 méthodes qui se nomme « clone » et « copyTo ». Voici un exemple :

```
Mat ImgDest6 = ImgDest3.Clone();
```

Maintenant si on effectue une transformation aux pixels de l'image ImgDest3 (comme un blur), tous les objets Mat qui pointent à cette même matrice seront transformées également puisqu'ils pointent tous à la même adresse en mémoire sauf ImgDest6 puisque c'est le seul objet à avoir reçu une copie grâce à la méthode « Clone ». On peut le constater dans l'image ci-dessous :

L'image de gauche est une copie de l'image de droite et même si l'image de droite a été transformée par un « blur », l'image de gauche reste intacte.

Création d'un objet Mat

Par son constructeur

Nous avons vu précédemment qu'un objet de la classe Mat est en grande majorité utilisé pour représenter une image en mémoire. Il s'agit d'utiliser la fonction « imread » pour l'assigner.

On peut voir les valeurs de la matrice en utilisant l'opérateur « << » (le même que celui que vous utilisez dans un « cout » en langage C++). Cependant, veuillez prendre note que cet opérateur ne fonctionne que pour les matrices en 2 dimensions. Bien que l'usage principal soit de représenter une image en mémoire, l'objet Mat peut aussi être utilisé pour représenter une simple matrice de nombre.

Vous pouvez créer un objet Mat de plusieurs façons :

Par son constructeur:

```
Mat M(2,2, CV_8U, Scalar(255));
cout << "M = " << endl << " " << M << endl << endl;
```

avec le résultat suivant:

```
M =
 [255, 255;
  255, 255]
```

On obtient bel et bien une matrice de 2 x 2 dont chaque élément sur 1 octet non signé (CV_8U) vaut 255.

La convention est la suivante en ce qui a trait au nombre d'octet qui représente une valeur:

CV_[Le nombre de bits par élément][Signé(S) ou non signé(U)]C[Le nombre de canal]

Voici les types OpenCV les plus rencontrés :

CV_8UC1 -> Chaque pixel est représenté avec 1 octet (8) non signé (U) et sur un seul canal (C1). Convient parfaitement aux images en ton de gris.

CV_8UC3 -> Chaque pixel est représenté avec 1 octet(8) non signé (U) et sur 3 canaux (C3). Convient parfaitement aux images en couleur.

CV_16SC1 -> Chaque valeur est représentée avec 2 octets(16) signé (S) et sur 1 canal (C1). Convient surtout aux images temporaires provenant de calcul comme les convolutions de Sobel ou Laplace.

CV_32FC -> Chaque valeur est représentée avec 4 octets (32) de type « float » (F) et possédant un canal (C).

CV_64FC -> Chaque valeur est représentée sur 8 octets (64) de type double et possédant un seul canal.

Par la fonction Create

```
Mat M;
M.create(4,4, CV_8UC(2));
cout << "M = "<< endl << " " << M << endl << endl;
```

```
M =
 [205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205]
```

La fonction "create" possède 3 constructeurs tels que montré ci-dessous:

void Mat::**create**(int **rows**, int **cols**, int **type**)¶
void Mat::**create**(Size **size**, int **type**)¶
void Mat::**create**(int **ndims**, const int* **sizes**, int **type**)¶

paramètres :

ndims : un entier qui représente la dimension.
rows : un entier qui représente le nombre de colonne de la matrice.
cols : un entier représentant le nombre de colonne de la matrice.
size : La dimension de la matrice exprimé avec la méthode Size. Ex : Size(3,3) pour une matrice 3 x 3.
Sizes : Un tableau d'entier qui spécifie les dimensions de la matrice.
Type : Le type de chacune des valeurs de la matrice. (Un des types tels CV_8U, CV_32FC1, etc...)

Par un tableau en C/C++

On initialise un tableau par les dimensions de la matrice et on associe ce tableau lors de l'appel du constructeur. Voici un exemple :

```
int  Dimension[3] = {2,2,2};

Mat  M1(3, Dimension, CV_8UC1, Scalar::all(0) );
```

Le constructeur ici précise qu'il y aura 3 dimensions de 2 colonnes par 2 rangées par 2 de hauteurs. Chaque élément est une valeur codée sur 8 bits et ayant 1 seul canal. Chaque élément est initialisé à la valeur 0 avec « Scalar ::all(0) ».

Initialisateurs de matrice spéciaux

Les initialisateurs zeros(), ones() et eyes() provoque l'assignation des matrices suivantes :

Zeros() :

```
Mat O = Mat::zeros(2, 2, CV_32F);
cout << "O = " << endl << " " << O << endl << endl;
```

ones() :

```
Mat O = Mat::ones(2, 2, CV_32F);
cout << "O = " << endl << " " << O << endl << endl;
```

eye():

```
Mat E = Mat::eye(4, 4, CV_64F);
cout << "E = " << endl << " " << E << endl << endl;
```

Pour des matrices 3 x 3 qui représente des matrices de convolution, l'écriture ci-dessous est tout à fait indiquée :

```
Mat Sharpen = (Mat_<double>(3,3) << 0, -1, 0,
                                    -1, 5, -1,
                                    0, -1, 0);
cout << "Sharpen = " << endl << " " << Sharpen << endl << endl;
```

```
C =
 [0, -1, 0;
  -1, 5, -1;
  0, -1, 0]
```