PICTWIN

–

TRAITEMENT DE L’IMAGE

Projet de fin d’année
(2015-2016)
DUT ANALYSE & PROGRAMMATION

Table des matières

Introduction

Je tiens à remercier Thomas VUARCHEX pour son aimable autorisation à utiliser sa création comme élément de travail pour ce projet de fin d'année.
La finalité de se projet est la mise en œuvre quelques possibilités de traitement d'image pour résoudre un casse-tête. Ça sera ainsi l'occasion de voir comment une image est représentée numériquement et comment, par cette représentation, on peut appliquer des traitements : conversion colorimétrique, sélection d'une région de l'image, comparaison de textures pour en déterminer les similitudes et détection de zones d'intérêt.
Le langage Python a été choisi pour sa lisibilité et une documentation très fournie. L'intégration de bibliothèques de traitement d'image, de visualisation de données graphiques et de calculs scientifiques ont aussi constitué un argument de choix. Nous aurons l'occasion d'expliquer les fonctions utilisées ; le but étant d'aboutir à un script qui résout le casse-tête tout en limitant l'intervention de l'usager.

Problématique

Caractéristiques du casse-tête:

Décrivons tout d'abord ce qui caractérise notre objet d'étude. Il s'agit d'un casse-tête de type "recherche de paires". La difficulté pour le joueur réside dans le tri d'une profusion de textures et de couleurs. Ainsi « 390 bulles différentes » sont réparties sur un fond de couleur uniforme.

Ces "bulles" aux formes aléatoires (convexes ou concaves) sont délimités par une bordure au trait plein sombre. Elles représentent des textures uniques, à l'exception des cinq paires similaires que le joueur doit identifier. Les bulles sont bien délimitées et espacées les unes des autres, ce qui facilite leur identification, extraction et comparaison.

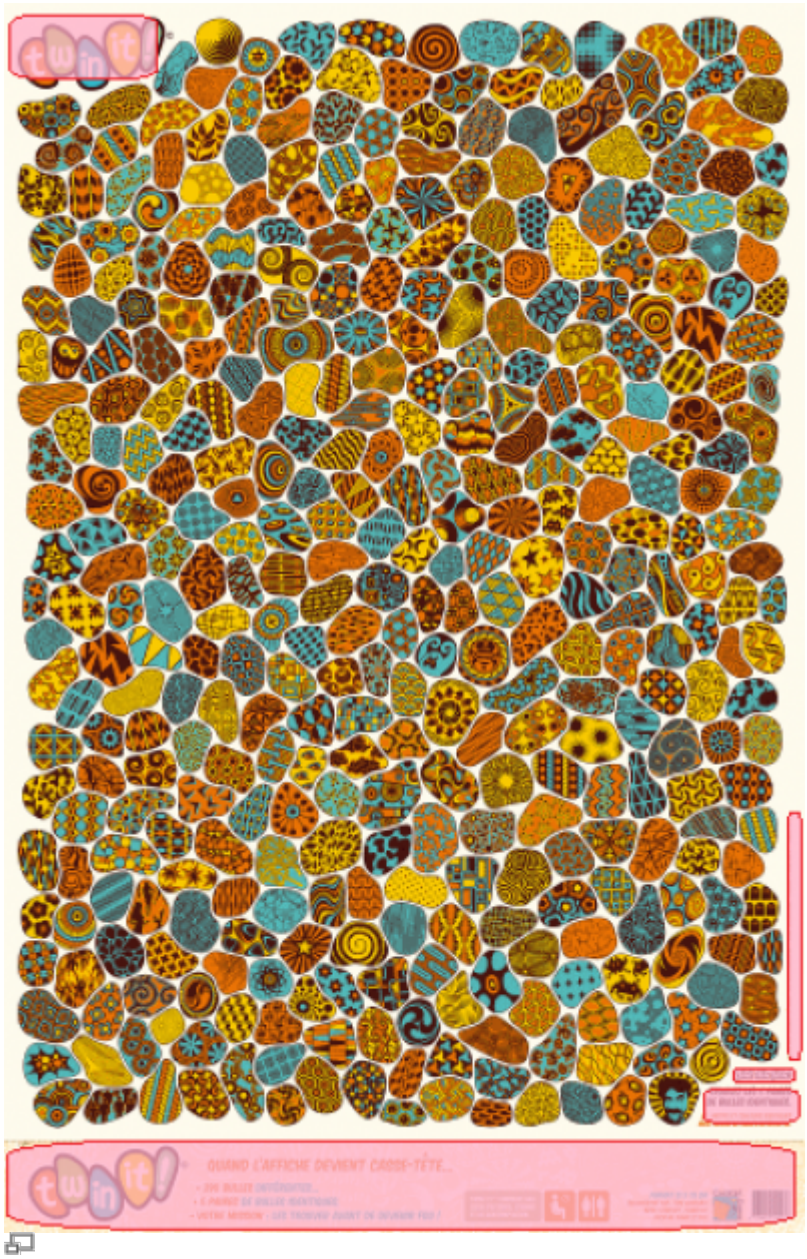

Bulle jumelle1


Bulle jumelle2

Les bulles jumelles n'ont pas obligatoirement la même forme, seule le motif les lie. Même si l'échelle est constante (le motif n'étant pas étiré d'une bulle à l'autre), la rotation ainsi qu'une légère translation restent possibles. De plus, malgré la profusion de textures, certaines sont très proches les une des autres. A charge pour nous de relever ces défis et de déterminer les solutions pour y

remédier.

Remarquons aussi que certaines régions du poster ne nous sont pas utiles. Il nous faut donc penser à éliminer ces zones qui pourraient poser problème. Il s'agit du nom du jeu situé dans le coin supérieur gauche ainsi que du bandeau de présentation situé dans la partie inférieure.



Zones à nettoyer: Logo + bandeau inferieur + annotations

Concepts fondamentaux

Sans entrer dans des considérations physiques qui dépassent notre sujet on définira l'image comme la représentation graphique d'un objet ou d'une scène. De plus les images auquelles nous nous intéresserons sont uniquement à deux dimensions, ce qui est suffisant pour aborder les concepts fondamentaux du traitement d'image.

Qu'es-ce qu'une image

L'image est mathématiquement définit par une fonction à deux variables réelles: **I(x,y)**, ces variables étant les coordonnées de points sur un plan cartésien. La fonction représentant l'**intensité** en ce point.
L'image inclut fréquemment des sous-images, appelées **ROI** (pour *region of interest*) qui correspondent aux divers objets répartis sur la scène. Dans notre cas, les 390 bulles constituent autant de ROI qui focalisent notre attention. La formation d'une image peut être le résultat de la transformation d'une scène réelle par un capteur:

- capteur chimique, comme notre œil sensible aux ondes électromagnétiques situé 400µm et 800µm ou comme un film photographique
- capteur thermique
- capteur photoélectrique, comme les dispositifs CCD/CMOS
- ou autres: IRM, imagerie sismique...

En l'occurrence, notre sujet est le fruit d'une création assistée par ordinateur. Bien souvent, les infographistes composent à l'aide d'objets géométriques individuels. L'imagerie vectorielle permet de manipuler[1)] ces objets sans perte de qualité.

La représentation informatique d'une image est par définition *discrète*, c'est à dire que ses éléments constitutifs sont nécessairement disjoints les uns des autres. Nous faisons alors appel à deux sortes de discrétisations:
l'une spatiale (**échantillonnage**) et la seconde, colorimétrique (**la quantification**). Ainsi la transformation d'un signal analogique tel que des ondes électromagnétiques en une somme de données numériques implique une perte d'information, même si celle-ci n'est pas visible à l'œil nu.

Notons que le champs d'étude de l'imagerie numérique ne se limite pas à la conversion de signaux analogiques en une structure de données manipulables par les machines. Il s'étend aussi à la mesure de ces données ainsi qu'à une analyse sémantique produisant une description de haut niveau de ce qui est représenté. C'est le cœur de l'indexation et de la recherche d'informations selon l'interprétation du contenu.

Le pixel

Pour la machine, l'image est un ensemble de données. Elles sont conceptuellement organisées en lignes et en colonnes. Nous parlerons alors d'image matricielle car la structure de données utilisées sera un tableau à deux dimensions. L'intersection d'une ligne et d'une colonne est dénommée *pixel*. Chacun de ces pixels possédera un type de valeur en fonction du type de l'image.

Image binaire

La plus basique étant l'image binaire dont les pixels n'auront que deux valeurs possibles. Généralement codé sur un 1 ou 8 bits.

1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1
1	1	1	0	0	0	1	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	1

Image en nuances de gris

L'image en nuances de gris composée de pixels de valeurs représentant une **luminosité** et réparties sur une échelle d'amplitude plus ou moins grande. On représente ci-dessous une échelle à 16 niveaux de gris qui peut être codée par 4 bits. Plus une échelle possède de niveaux, et plus le nombre de bits nécessaires à son codage est important.

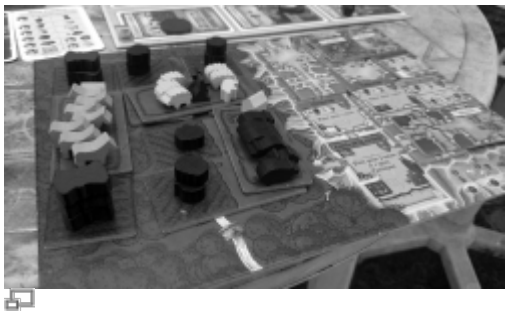
Image en couleur

Quant aux images couleurs, plusieurs modes de représentation existent, en ce qui nous concerne nous aborderons le mode RGB. Leurs pixels possèdent des valeurs vectorielles[2)]: l'**espace de couleurs** est ainsi former par l'utilisation des trois couleurs primaires, le rouge, le vert et le bleu. On parle de couleurs additives. Une couleur donnée sera obtenue en équilibrant l'intensité de chaque composante. Comme pour les nuances de gris, chacune des 3 valeurs est répartie sur une échelle ayant une **profondeur de couleur** plus ou moins importante. Par exemple, une image RVB codée en 8 bits par couche[3)], contiendra 256 nuances par couche (pour un totale de 61 777 216 couleurs disponibles). Il est ainsi possible de décomposer une image en chacune des 3 couches de couleurs, en ne conservant qu'une seule des 3 composantes pour chaque pixel:
Ci-dessus, on illustre l'extraction pour chaque pixel de sa composante rouge, verte et blue. Plus la valeur de la composante est élevée dans l'image source, plus la luminance corespondant à ce pixel dans l'image de résultat sera intense.

Les 3 couches RGB d'une image couleur

Couche rouge

Couche verte



Couche bleu

La résolution

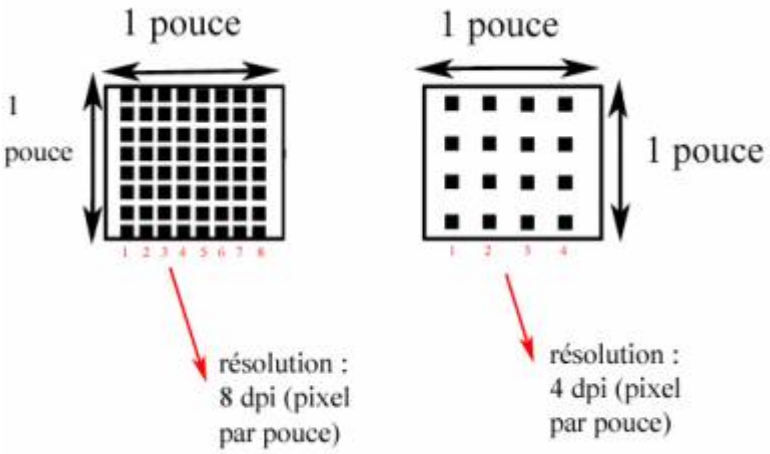
Les dimensions d'une image donnée en nombre de pixel permettent de déterminer une densité de pixel par unité de taille, généralement le *pouce*(**ppp**). Les images suivantes ont différentes résolutions mais avec la même taille d'affichage afin d'illustrer l'impact de la différence de densité de pixels:

400x200

200x100

100x50

Schématiquement cela donne :



Déterminer la résolution

L'image avec laquelle nous allons travailler aura une résolution de 300ppp. Pour des dimensions de (1632×2500), ce qui nous donne une largeur égale à 1632/300, soit 5.44 pouces et une hauteur égale à 2500/300, soit 8,33 pouces environs.

Stockage de l'image

Nous allons étudier, à titre d'exemple, quelques formats classiques de stockage d'images.

Bitmap

le **BMP** (pour bitmap) qui est un format développé par Microsoft et IBM. Le fichier est composé par trois parties: l'**en-tête** du fichier, la **palette** de couleur et enfin les **données** propres à l'image. Ce format permet de stocker des images numériques, monochromes ou en couleur, statiques à deux dimensions. Ainsi le *header* va contenir les informations générales concernant le fichier et son utilisation[4)], la palette indique la profondeur de coloration[5)]. Puis, en dernier, on retrouve le tableau de pixels, qui sont codés en partant du coin inférieur gauche de l'image, et qui occupent un nombre d'octets multiples de 4 à chaque ligne (compensé par l'ajout de colonnes si nécessaire). Pour une image en 24bits, chaque pixel est codé par 3 octets en *little-endian*[6)], avec successivement le niveau de bleu, vert puis rouge.

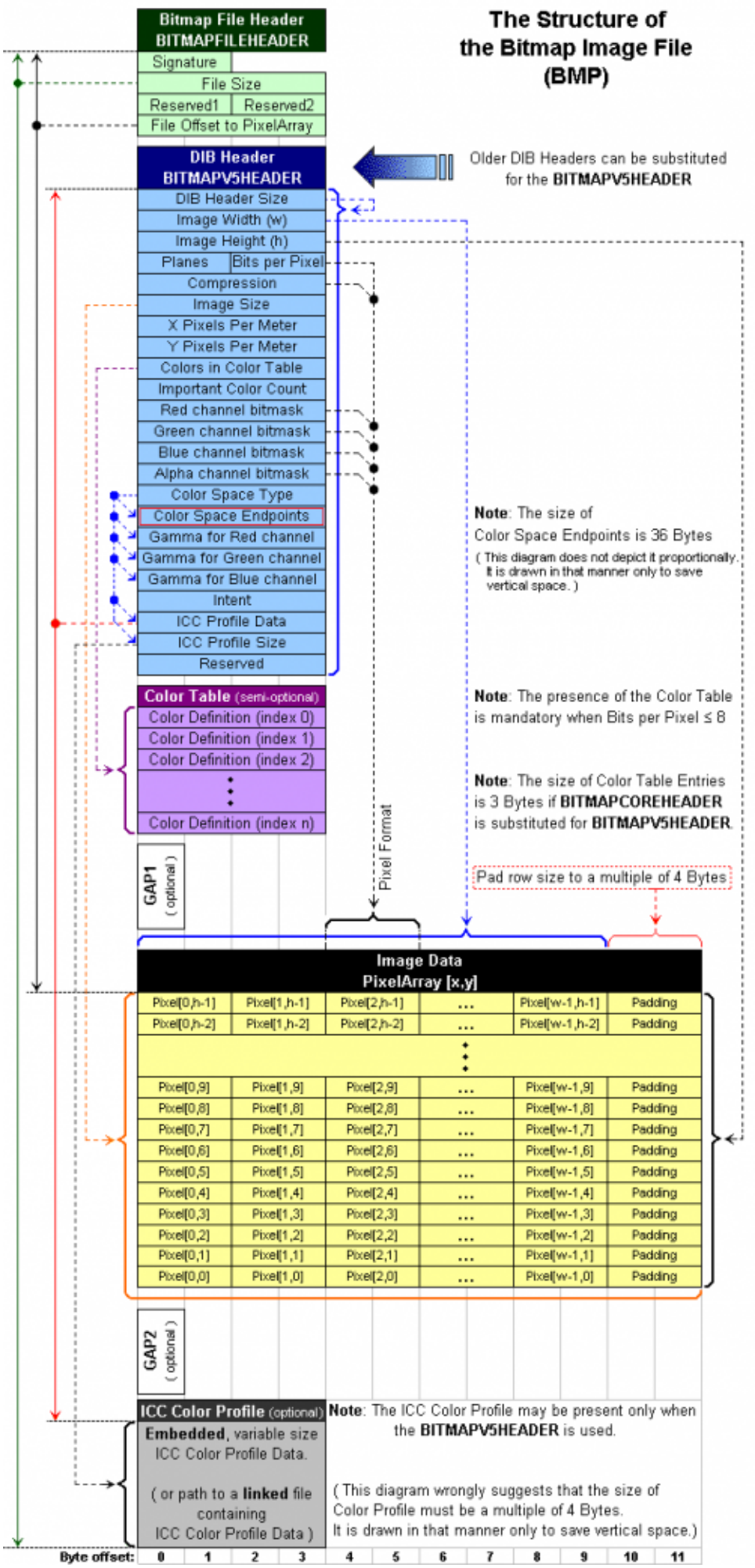

The Structure of the Bitmap Image File (BMP)
Bitmap File Header
BITMAPFILEHEADER
Signature
File Size
Reserved1
Reserved2
File Offset to PixelArray
DIB Header
BITMAPV5HEADER
Older DIB Headers can be substituted for the BITMAPV5HEADER
DIB Header Size
Image Width (w)
Image Height (h)
Planes
Bits per Pixel
Compression
Image Size
X Pixels Per Meter
Y Pixels Per Meter
Colors in Color Table
Important Color Count
Red channel bitmask
Green channel bitmask
Blue channel bitmask
Alpha channel bitmask
Color Space Type
Color Space Endpoints
Gamma for Red channel
Gamma for Green channel
Gamma for Blue channel
Intent
ICC Profile Data
ICC Profile Size
Reserved
Note: The size of Color Space Endpoints is 36 Bytes
(This diagram does not depict it proportionally. It is drawn in that manner only to save vertical space.)
Color Table (semi-optional)
Color Definition (index 0)
Color Definition (index 1)
Color Definition (index 2)
Color Definition (index n)
Note: The presence of the Color Table is mandatory when Bits per Pixel ≤ 8
Note: The size of Color Table Entries is 3 Bytes if BITMAPCOREHEADER is substituted for BITMAPV5HEADER
GAP1 (optional)
Pixel Format
Pad row size to a multiple of 4 Bytes
Image Data
PixelArray [x,y]
Pixel[0,h-1] Pixel[1,h-1] Pixel[2,h-1] ... Pixel[w-1,h-1] Padding
Pixel[0,h-2] Pixel[1,h-2] Pixel[2,h-2] ... Pixel[w-1,h-2] Padding
Pixel[0,9] Pixel[1,9] Pixel[2,9] ... Pixel[w-1,9] Padding
Pixel[0,8] Pixel[1,8] Pixel[2,8] ... Pixel[w-1,8] Padding
Pixel[0,7] Pixel[1,7] Pixel[2,7] ... Pixel[w-1,7] Padding
Pixel[0,6] Pixel[1,6] Pixel[2,6] ... Pixel[w-1,6] Padding
Pixel[0,5] Pixel[1,5] Pixel[2,5] ... Pixel[w-1,5] Padding
Pixel[0,4] Pixel[1,4] Pixel[2,4] ... Pixel[w-1,4] Padding
Pixel[0,3] Pixel[1,3] Pixel[2,3] ... Pixel[w-1,3] Padding
Pixel[0,2] Pixel[1,2] Pixel[2,2] ... Pixel[w-1,2] Padding
Pixel[0,1] Pixel[1,1] Pixel[2,1] ... Pixel[w-1,1] Padding
Pixel[0,0] Pixel[1,0] Pixel[2,0] ... Pixel[w-1,0] Padding
GAP2 (optional)
ICC Color Profile (optional)
Embedded, variable size ICC Color Profile Data.
(or path to a linked file containing ICC Color Profile Data)
Note: The ICC Color Profile may be present only when the BITMAPV5HEADER is used.
(This diagram wrongly suggests that the size of Color Profile must be a multiple of 4 Bytes. It is drawn in that manner only to save vertical space.)
Byte offset: 0 1 2 3 4 5 6 7 8 9 10 11

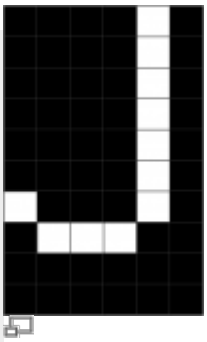
Illustration tirée de Wikipédia de la structure d'un fichier BMP

Formats PNM

Bien entendu d'autres formats classiques existent comme le *portable pixmap file format* (PPM), le *portable graymap file format* (PGM) et le *portable bitmap file format* (PBM). On retrouve une structure similaire, avec un **nombre magique** relatif au type du format et au stockage des données, les **dimension** de l'image, une **profondeur de couleur** s'il y a lieu et les **données** propres à l'image.

Ci-dessous l'exemple d'un fichier PBM dont la première ligne (P1) indique que les données seront stockées en ascii, la seconde est un commentaire, suivi par les dimensions (nb. de colonnes et nb. de lignes) et enfin les données propres à l'image. Le premier entier correspondant à la donnée du pixel du coin haut-gauche.

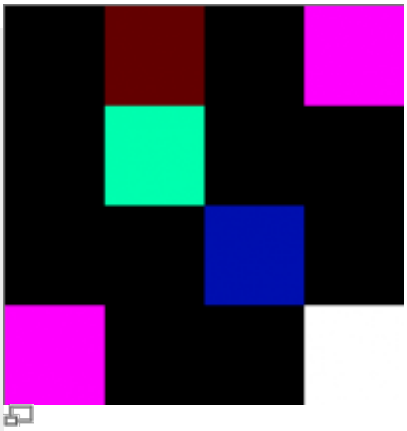
```
P1
# Dessine_moi_un_J.pbm
6 10
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
0 0 0 0 1 0
1 0 0 0 1 0
0 1 1 1 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```



Interprétation graphique du fichier *Dessine_moi_un_J.pbm*

Idem pour l'exemple ci-dessous où on trouve la profondeur de couleur située à la 4éme ligne et précisant la valeur maximum que peut prendre chaque composante de pixel. Un pixel étant codé par un triplet qui représente les composantes RVB.

```
P3
#Un_carré.ppm
# Le P3 signifie que les couleurs
sont en ASCII,
# par 4 colonnes et 4 lignes,
4 4
# ayant 255 pour valeur maximum, et
qu'elles sont en RGB.
255
0  0  0     100 0 0       0 0 0
255 0 255
0  0  0     0 255 175     0 0 0
0 0 0
0  0  0     0 0 0        0 15 175
```



Interprétation graphique du fichier *Un_carré.ppm*

```
0 0 0
255 0 255   0 0 0         0 0 0
255 255 255
```

JPEG/JFIF

L'image que nous allons manipuler est au format JPEG. Celui-ci est issu du travail du *Joint Photographic Experts Group* qui établit un standard d'encodage pour image numérique fixe. Ainsi va être définit une méthode de compression de l'image et son algorithme de décodage. Pour ce faire, plusieurs astuces sont exploités. Plutôt que de mémoriser une succession de données similaires: “VERT VERT VERT VERT ROUGE ROUGE ROUGE JAUNE JAUNE”, on ne mémorise que le nombre de répétitions “4:VERT 3:ROUGE 2:JAUNE”. Ce type de compression est sans perte mais ne garantit pas une compression optimale.
JPEG utilise différents moyens de compression comme la réduction du nombre de couleurs disponibles. Ainsi des teintes très proches les unes des autres vont être *rassemblées*[7)], après cet échantillonnage, l'image est découpé en blocs de 8×8 pixels qui vont être considérés comme une fonction numérique à deux variables, en une somme de fonctions cosinus oscillant à des fréquences différentes (On applique une fonction appelée *transformation discrète en cosinus*). Chaque bloc est ainsi décrit en une carte de fréquences et en amplitudes plutôt qu'en pixels et coefficients de couleur. La valeur d’une fréquence reflète l’importance et la rapidité d’un changement, tandis que la valeur d’une amplitude correspond à l’écart associé à chaque changement de couleur“[8)]. Enfin après encodage de l'image, la méthode de Huffman va permettre d'attribuer aux pixels des mots de code (un index) dont la longueur varie selon la fréquence d'apparition de ces pixels.
La compression de données est essentielle pour faciliter l'échange des images en limitant la consommation de bande passante par exemple. Et grâce à la compression JPEG, il est possible de “contrôler” la perte d'information due à la compression.
Le format de fichier embarquant un flux codé en JPEG est en réalité appelés JFIF (pour *JPEG File Interchange Format*), meme si par abus on parle de fichier JPEG.

L'image en Python

Python est un langage dynamique, libre et pour lequel il existe beaucoup de bibliothèques de calcul et visualisation scientifique. Par rapport au C ou Java, le code est plus concis et plus lisible rendant donc le développement clair et plus rapide. Python est aussi une alternative au développement scientifique sous Matlab, avec l'avantage d'être un langage orienté-objet, ce qui rend certains aspects du développement plus faciles.
A l'heure actuelle deux branches du langages existent. La branche 3.X apporte des changement qui sont incompatibles avec les version antérieurs (2.7 et antérieurs). Ces changements se font au bénéfice d'une plus grande cohérence dans la syntaxe, avec des changements de noms de modules, une réorganisation des objets. Ceprendant, nous utiliserons python2 car certaines bibliothèques scientifiques ne sont pas encore compatibles avec python3.
L'environnement de développement utilisé sera Spyder qui est fourni en installant PythonXY. C'est une distribution qui regroupe nombres d'outils scientifiques de développement, d'analyse, de calcul, de *designe* d'application. Seront donc directement disponibles, des modules tels que Matplotlib et NumPy. Nous ajouterons par ailleurs la bibliothèque OpenCV (version3.1) qui implémente un très grand nombre d'algorithmes d'analyse d'images.

Manipulation

La lecture et la manipulation d'une image en python se font par l'intégration de fonctionnalités de certaines bibliothèques telles que PIL, NumPy ou openCV. Cette dernière va en fait utiliser NumPy et ses optimisations d'opérations sur les structures de données tels que les tableaux multi dimensionnels. On charge une variable avec le résultat de la fonction cv2.imread() qui comprend le chemin d'accés au fichier ainsi qu'un second argument (-1, 0, ou 1) indiquant respectivement si l'image est lue:

- sans changement par rapport aux données du fichiers et en conservant les données du 4éme canal relatif à la transparence(-1 équivalent à cv2.IMREAD_UNCHNAGED), les 3 premiers canaux étant ceux des composantes RVB,
- si le fichier est lu en mode nuance de gris (0 équivalent à cv2.IMREAD_GRAYSCALE),
- ou si il est lu en mode couleur (1 équivalent à cv2.IMREAD_COLOR).

Importation de bibliothèques et lecture d'image

```
import cv2
import numpy as np

#Read original file:
my_image = cv2.imread('src/twinIt.jpg',1)
print (my_image.shape)
```

Une fois la variable chargée, il est possible d'accéder aux données de chacun des pixels en indiquant ses coordonnées. On peut ainsi retourner ces données et même les modifier. Notons qu'OpenCV manipule les couleurs de pixel sous un format particulier : Bleu Vert Rouge, cela est du comme nous l'avons vu dans la partie précédente au type de codage en *big endian*. On peut aussi retourner les propriétés de notre image

Acces pixel

```
my_pixel = my_image[10,25]
print (my_pixel)
my_image[10,25]=(255,255,255)
```

Il est déconseillé de parcourir l'ensemble des pixels d'une image par une boucle *for* car cela enlève tout l'intérêt que représente NumPy pour ce type d'opérations.
Il est possible de sélectionner une ROI de l'image: Prenons un sommet de coordonnées **(x,y)**, et son opposé par rapport au centre du rectangle **(x+w,y+h)** avec **w** et **h** correspondant aux dimensions de notre ROI. La syntaxe utilisée sera my_image[y:y+h, x:x+w].

Une ROI

```
my_roi = my_image[0:4,0:10]
```

Nous aurons aussi besoin d'afficher des images. Pour ce faire, il existe la methode cv2.imshow(), qui aura préalablement besoin d'un contenant[9)], qui sera créer grace aux fonction cv2.namedWindow(titre, option), cv2.moveWindow(titre,posX,posY) et cv2.resizeWindow(titre,largeur,hauteur). Il ne faut donc pas oublier de définir une condition afin de fermer l'affichage.

Afficher une image

```
cv2.namedWindow('titre', cv2.WINDOW_KEEPRATIO)
cv2.moveWindow('titre', 0, 0)
cv2.resizeWindow('titre', 800, 1200)

cv2.imshow('titre')
#Close all window
if cv2.waitKey(0) & 0xff == 27:
    cv2.destroyAllWindows()
```

Transformation morphologique

Pour résoudre notre casse-tête on va faire appel à certains algorithmes de transformation morphologique. Ces opérations nous seront notamment utiles afin de créer des masques et permettre d'extraire des ROI en détourant les zones où se trouvent nos *bulles* des zones d'arrière plan. Mais il nous faut avant tout convertir notre image couleur en une image en nuances de gris. Pour ce faire nous allons utiliser la méthode cv2.cvtColor() qui va appliquer aux 3 composantes BGR les coefficients de la formule suivante afin d'obtenir une valeur (comprise entre 0 et 255) équivalante à la luminosité de notre pixel.
Prenons pour exemple un pixel "bleu ciel", avec une valeur RGB=[0,128,255]. La conversion en nuance de gris se fera ainsi:

```
Y=0.299 * R + 0.587 * G + 0.114 * B
Y=0.299 * 0 + 0.587 * 128 + 0.114 * 255
Y=104
```

Nous sommes ainsi passés d'un pixel à 3 canaux à un pixel à 1 canal.

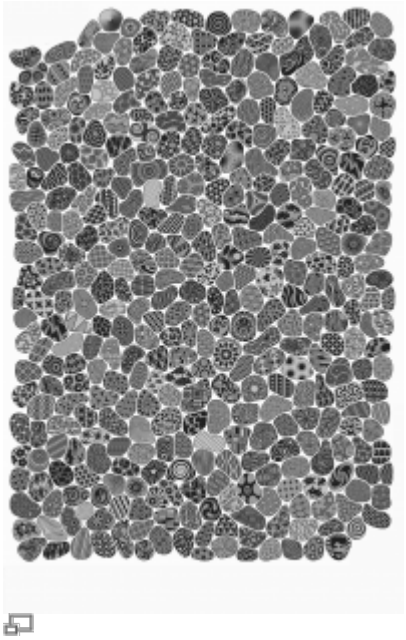
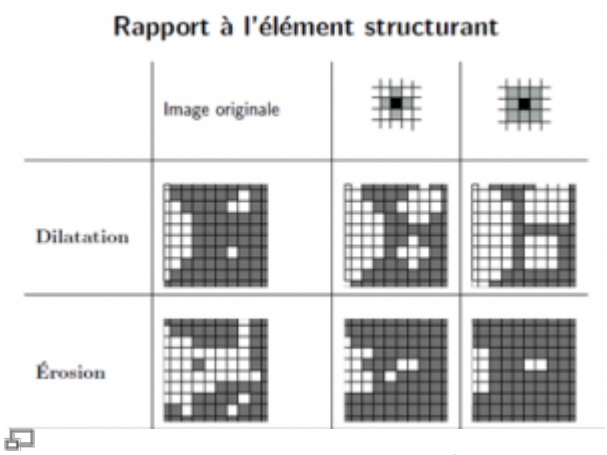


Image convertie en niveaux de gris

Création d'un masque

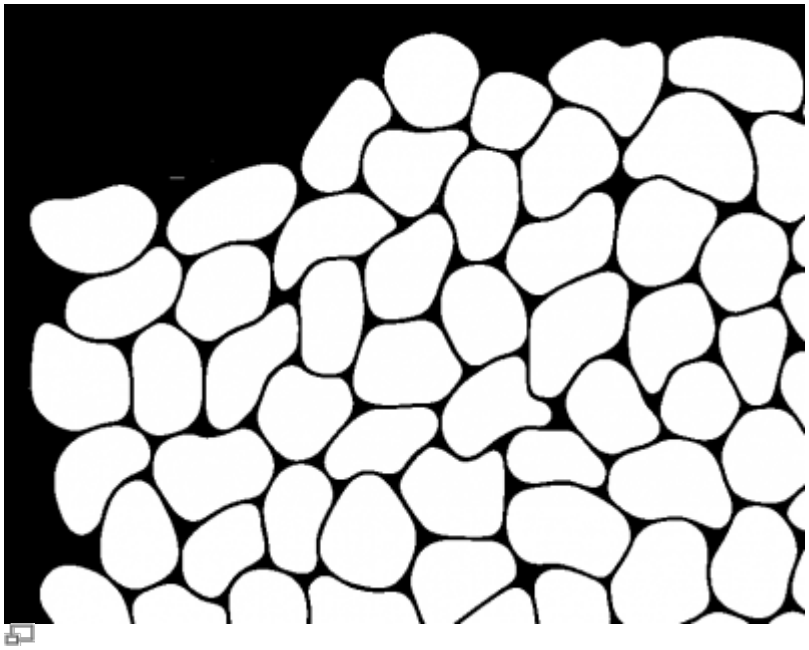
Notre image ainsi convertie pourra ensuite subir une binarisation (aussi appelée *binary thresholding*). Ceci sera exécuté grâce à la fonction cv2.threshold() à laquelle nous fournissons 4 arguments: l'image source, le seuil, la valeur d'intensité pour chaque pixel dépassant le seuil, une option. Ainsi on retourne une image de la même dimension où chaque pixel de destination à une valeur dépendante de la valeur du pixel source et du seuil. L'option cv2.THRESH_BINARY_INV va nous permettre de crée un négatif de l'image source, avec un arrière plan noir et un plan avant (les bulles) en blanc.

Notre image monochromatique comporte cependant quelques défauts; Il s'agira ensuite de les atténuer grâce à la méthode cv2.morphologyEx(). Celle-ci prend en troisieme argument, un tableau à deux dimensions, correspondant à l'**élément structurant** (appelé *kernel*). Ce dernier servira à modifier la morphologie des objets représentés en combinant deux opérations : la dilatation et l'érosion.
Illustrons le déroulement d'une érosion avec un élément structurant. On parcourt chaque pixel de l'objet, en centrant le *kernel* sur le pixel courant. Si le pixel courant possède au moins une connexion avec l'arrière plan, alors ce pixel sera érodé, sinon le pixel n'est pas modifié.
La dilatation suit un processus inverse, en translatant le *kernel* pixel à pixel, les pixels d'arrière plan et possédant une connexion avec un objet du premier plan recevront la valeur de cette objet.

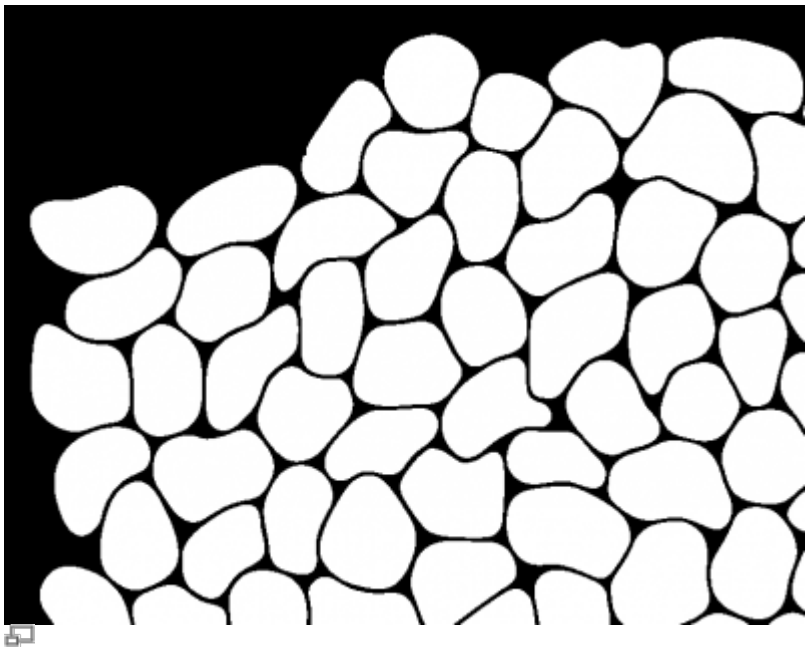


Comparatif: érosion et dilatation[10)]

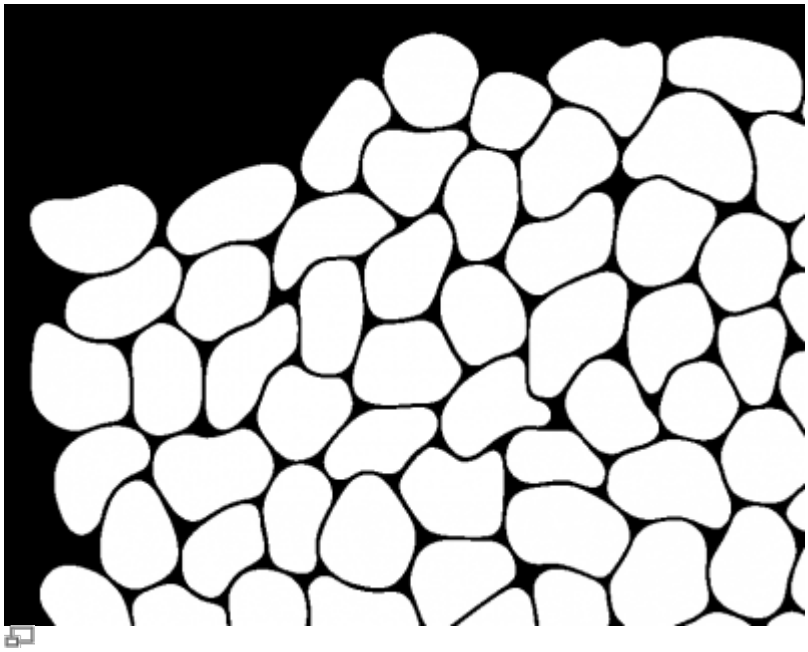
Nous pouvons aussi comparer l'évolution du traitement en fonction des *kernel* utilisés:



Binarisation avec seuil à 249: On remarque la présence de pixels blancs dans la zone d'arrière plan



Masquage avec élément structurant en 3×3: On peux noter un crénelage grossier au niveau de certaines bulles



Masquage avec élément structurant en croix On voit des contours plus nets et un arrière plan noir uniforme

Nous retrouvons ce processus dans la fonction mask() qui a été définie dans le module identify.py. Les deux procédedés (érosion et dilatation) sont en fait combiner dans la methode cv2.morphologyEx(). Ce que “cache” l'argument cv2.MORPH_OPEN est en réalité l'application d'une érosion. Tous les objets la subissant, cela permet de nettoyer les imperfections qui pourraient se trouver sur l'arriere plan, suivies par l'application d'une dilatation afin de conserver la taille des objets qui nous interesses. L'illustration ci-dessous montre l'avant-aprés cv2.morphologyEx():

Ouverture: érosion suivi de dilatation[11)]

Création d'un masque: identify.py

```
    def mask(self, mode=None):
        """Creat a mask of a given image:
        Use cv2.threshold() with cv2.THRESH_BINARY_INV mode to get a binary
image
        with white forground and black background
        Use cv2.morphologyEx() with cv2.MORPH_OPEN option to remove BG
noises
        """
        #Convert original image to grayscale option
        img_gray = cv2.cvtColor(self.img, cv2.COLOR_BGR2GRAY)
        #Bounding detection:
```

```
        #Threshold = 249, as brightness background area between 250 and 255
        ret, mask_inv = cv2.threshold(img_gray, 249, 255,
cv2.THRESH_BINARY_INV) #FG=White BG=Black
        #Morphological Gradient
#        kernel = np.ones((3,3),np.uint8)
        kernel = np.array([[0, 1, 0], [1, 1, 1], [0, 1, 0]], dtype=np.uint8)
        gradient_mask_inv = cv2.morphologyEx(mask_inv, cv2.MORPH_OPEN,
kernel)
        if mode is None:
            return gradient_mask_inv
        else:
            cv2.imwrite(os.path.join(self.dirname, "mask Result.png"),
gradient_mask_inv)
            cv2.imwrite(os.path.join(self.dirname, "threshold249.png"),
mask_inv)
```

Opérations logiques

En considérant une image en tant que tableau à deux dimensions, il est possible d'appliquer des opérations logiques impliquant les données de deux images distinctes. La fonction extract_cell() nous donne l'occasion d'appliquer ce type d'opération. On extrait les coordonnées et les dimensions d'une ROI pour une cellule donnée (en argument de la fonction). On crée ensuite un tableau aux dimensions de l'image source, np.zeros() permettant de retourner un tableau dont les valeurs sont mises à zéros (codé en 8 bit, comme l'indique de le datatype: np.uint8) et qui correspond à une image “noir”. Ensuite nous allons dessiner l’intérieur de la bulle que l'on veut extraire sur le tableau qu'on vient de créer et qui deviendra notre future masque. Une seconde image noire sera créée np.zeros_like() (qui sera similaire a np.zeros mais en utilisant le même type de données que l'image source). Ce nouveau tableau sera modifié selon la logique suivante: sa région correspondant à la région où a été dessiné le contour (en blanc sur notre masque) va se voir attribuer la région de l'image en nuance de gris correspondant à ce même masque. Enfin, on sélectionne la partie de l'image ne contenant que la bulle qui nous intéresse cela dans le but de procéder à une éventuelle comparaison entre cette sélection et le reste de l'image.

Opération logique: identify.py

```
    def extract_cell_gray(self, i, mode=None):
        """Extract one cell from self.img with #id = i argument
            Call contour() to get all contours
            Get position and dimension of query cell's rectangle w/
cv2.boundingRect()
            Creat an array filled w/ zeros (black) and the same shape of the
grayscaled image
            Fill-in the body of the query cell (w/ white) and draw it in
previous array
            Creat an array filled w/ zeros (black) and the same shape of the
grayscaled image
                out[contours_img == 255] = img_gray[contours_img == 255]
            We select the region of img_gray that corresponds to our mask in
contours_img (as cell filled-in w/ 255)
            This selected region will be transferred on our 'out' array in
```

```
same position
            We crop the cell's rectangle on a black background
            The purpose is to only use the texture of the query cell for
matching and avoid surrounding cells part
            If 'mode' is not 'None' extract_cell)() will write image's file
in current directory
        """
        img_gray = cv2.cvtColor(self.img, cv2.COLOR_BGR2GRAY)
        _, contours,_ = self.contour()
        cnt = contours[i]
        x,y,w,h = cv2.boundingRect(cnt)
        #Crop internal area of current cell
        contours_img = np.zeros(img_gray.shape, np.uint8)  #Return a new
array of given shape and type, filled with zeros.
        cv2.drawContours(contours_img, [cnt], 0, (255,255,255), -1)
        out = np.zeros_like(img_gray) #Same as np.zeros() but return an
array w/ same dtype as the given array
        out[contours_img == 255] = img_gray[contours_img == 255]
        cropped_cell = out[y:y+h,x:x+w]
        if mode is None:
            return cropped_cell
        else:
            cropped_cell_file = 'cropped_cell-'+str(i)+'.png'
            cv2.imwrite(os.path.join(self.dirname, cropped_cell_file),
cropped_cell)
```

[1] par rotation, étirement, changement d'échelle...
[2] En opposition à des valeurs scalaires comme c'est le cas pour les pixels d'images binaires ou en nuance de gris, qui possèdent une seule composante (codé par un entier généralement).
[3] appelée image 24 bits, puisque le codage de la couleur est réalisé sur 3 octets
[4] Une variante du BMP, le DIP contient un *header* avec des informations plus détaillées
[5] Les images BMP peuvent être en 2 couleurs (1 bit), 16 couleurs (4 bits), 256 couleurs (8 bits), 65 536 couleurs (16 bits) ou 16,8 millions de couleurs (24 bits)
[6] Les octets de poids faibles étant alors stockés aux adresses les plus petites
[7] Une image 24bits a la possibilité de représenter 16 777 216 couleurs différentes, bien plus que ce que nous pouvons discerner
[8] http://www.biostatisticien.eu/CCM/video/compimg.htm
[9] même si l'affichage reste possible, la creation d'une fenetre facilite sa manipulation
[10] http://vokvince.free.fr/IMAC/TS/morphologie.html
[11] http://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html

picTwin: de l'analyse à la réalisation

En décrivant les spécificités du casse-tête et de ses bulles (appelées *cell* dans le code), certains éléments de conception vont émerger. Il sera tout d'abord nécessaire de “nettoyer” notre image afin de ne conserver que les parties utiles. Ensuite, on aura besoin d'identifier les bulles et de les situer afin de les extraire et les manipuler dans une troisième phase qui est la comparaison à proprement dite. Pour plus de lisibilité, quatre modules seront créés: **cleanImage.py**, **identify.py** et **match.py**. Enfin, on ajoutera un module d'interaction et d'affichage défini dans **display.py**. Le point d'entrée du programme sera **pictwin.py**.

Nettoyer

C'est une partie faisant intervenir l'usager. Il est important que le processus soit le plus naturel et direct possible. Nous verrons que des changements seront apportés entre la version de test et celle d'intégration pour répondre à cette simplification.
Trois bibliothèques vont nous être nécessaires: celle d'openCv pour la manipulation de fichiers d'image, leur traitement et l'interaction entre les entrées standards clavier/souris et des fonctions qu'on aura définies, ainsi que les bibliothèques *os* et *shutil* pour la manipulation de fichiers et dossiers système.

Test Module #1 : *clean.py*

```
import cv2
import os
import shutil

GREEN = [0,255,0]
HIDE = [249,250,250]
ix,iy = -1,-1
picking = False
drawing = False

def getcolor_mouse(event, x, y, flags, param):
    global img, ix, iy, picking, HIDE
    #Pick BGR values of the pixel underneath mouse cursor
    if event == cv2.EVENT_LBUTTONDOWN and picking == True:
        ix, iy = x, y
    elif event == cv2.EVENT_LBUTTONUP and picking == True:
        pix = img[ix, iy]
        HIDE[0], HIDE[1], HIDE[2] = int(pix[0]), int(pix[1]), int(pix[2])
        print (HIDE)
        picking = False

def hide_mouse(event, x, y, flags, param):
    global img, img2, drawing, ix, iy, GREEN, HIDE
    # Draw rectangle
    if event == cv2.EVENT_LBUTTONDOWN:
```

```
        drawing = True
        ix,iy = x,y     # stores mouse position in global variables ix,iy
    elif event == cv2.EVENT_MOUSEMOVE:
        if drawing == True:
            img = img2.copy()       # refresh img to draw on clean image
            cv2.rectangle(img, (ix,iy), (x,y), GREEN, -1)
    elif event == cv2.EVENT_LBUTTONUP:
        drawing = False
        cv2.rectangle(img, (ix,iy), (x,y), HIDE, -1)
        img2 = img.copy()    # img2 will keep last drawn rectangle
```

Les **lignes 3** à **9** déclarent et initialisent des variables: **GREEN** et **HIDE**[12)] sont des listes d'entiers qui facilitent la manipulation de couleurs dans certaines méthodes de dessin d'openCv. **ix** et **iy** seront utilisées pour les coordonnées du curseur, tandis que les booléens **picking** et **drawing** serviront à déterminer la cohérence entre la situation dans un processus (choix de couleur ou sélection d'une aire de dessin) et les événements relatifs aux touches et boutons des entrées souris/clavier.
Nous avons ensuite défini deux fonctions de rappel (ou *callback* dans la documentation) qui adoptent un comportement selon les types d'événements souris(**event**), la position du curseur(**x**,**y**) et les conditions à l'occurrence de l'événement (**flags** indiquant si les touches Ctrl ou Alt sont appuyer par exemple).

On remarque une répétition dans ce bloc de test concernant la sélection de couleur (lignes **56** à **74**) et le masquage des zones superflus du poster original (lignes **77** à **96**):

1. On crée un contenant pour afficher notre image, en spécifiant que sa taille sera modifiable,
2. On positionne la fenêtre,
3. On associe la fonction de rappel à la fenêtre nouvellement créée,
4. Une boucle infini est utilisée pour la prise en compte de l'entrée clavier. On pourra enclencher certains états des booléens déclarés en amont en nous permettant de contrôler les action réalisés au gré des événement de la souris ou même sauvegarder notre image correctement masquée pour une utilisation ultérieur.

Test Module#1 suite: *clean.py*

```
#For test purpose
if __name__ == '__main__':
    #Read original file:
    #options[ cv2.IMREAD_COLOR=1, cv2.IMREAD_GRAYSCALE=0,
cv2.IMREAD_UNCHANGED=-1 ]
    img = cv2.imread('twinIt.jpg',1)
    im2 = img.copy()
    #Create working directory
    dirname = 'results'
    if os.path.exists(dirname):
        shutil.rmtree(dirname, ignore_errors=True)
    os.mkdir(dirname)
    text = ("""
                -Press 'c' for picking mode
                -Left mouse click to pick color
                -Press 'q' for next step: HIDE
            """)
    #Resizeable input windows
```

```
    cv2.namedWindow('PICK', cv2.WINDOW_KEEPRATIO)
    cv2.moveWindow('PICK', 500, 50)
    #Pick pixel color values in the background
    cv2.setMouseCallback('PICK', getcolor_mouse)
    while(1):
        cv2.imshow('PICK', img)
        k = 0xFF & cv2.waitKey(1)
        if k == 27:         # esc to exit
            break
        elif k == ord('c'): # pick color
            picking = True
        elif k == ord('r'): # reset all flags
            picking = False
            HIDE = [249, 250, 250]
            print(" Values have been reset \n")
        elif k == ord('q'): # next step
            print("Draw rectangles to hide useless area")
            break
    cv2.destroyWindow('PICK')
    #Resizeable input windows
    cv2.namedWindow('HIDE', cv2.WINDOW_KEEPRATIO)
    cv2.moveWindow('HIDE', 500, 50)
    #Draw rectangles to hide useless area + Save working image
    cv2.setMouseCallback('HIDE', hide_mouse)
    while(1):
        cv2.imshow('HIDE', img)
        k = 0xFF & cv2.waitKey(1)
        # key bindings
        if k == 27:         # esc to exit
            break
        elif k == ord('r'): # reset all flags
            drawing = False
            img = cv2.imread('twinIt.jpg', 1)
            img2 = img.copy()
            print(" Values have been reset \n")
        elif k == ord('s'): # save image
            cv2.imwrite(os.path.join(dirname, 'pictwin.png'), img)
            print("Saved in {}/pictwin.png".format(dirname))
            break
    cv2.destroyWindow('HIDE')
```

Quelques mots sur la **ligne 38**:

```
if __name__ == '__main__':
```

A la lecture d'un fichier source l'interpréteur python prend en compte l'ensemble du code lu. Ceci peut être problématique si notre module est importé par un module tiers afin d'en utiliser les fonctions définies et non pas le code qui a servi aux tests de validation des fonctions.

> Dans notre exemple, la lecture d'image (img = cv2.imread('twinIt.jpg',1)) et la création du dossier de travail (os.mkdir(dirname)) seront la responsabilité du script principal.
> Or chaque module possède un attribut **__name__** qui aura pour valeur la chaine de caractères **"__main__"** si celui-ci est exécuté de manière indépendante (*standalone*). Mais à l'inverse, si le fichier est importé par un autre module, alors l'attribut **__name__** sera attaché au nom du module courant.

Voyons maintenant à quoi ressemble ce premier module qui sera intégré au programme principal où l'on va minimiser la redondance de codes[13)] et encapsuler nos fonctions. Nous simplifierons par ailleurs l'utilisation, la sélection de couleurs se faisant avec le clic droit, tandis que le masquage se fera avec le clic gauche. Nous garderons deux touches claviers, pour réinitialiser les valeurs par défaut en cas d'erreur de manipulation et pour sauvegarder l'image dans un fichier. Nous ajouterons enfin une fonction retournant l'image afin de pouvoir la manipuler ultérieurement.
Notons aussi que les variables ne seront plus déclarées en global. En les déclarant au sein de la méthode **__init__**, elles vont dépendre d'une instance donnée et seront accessibles aux fonctions de la classe.

Module#1 intégration: *cleanImage.py*

```
import cv2
import os

class Cleaning(object):
    def __init__(self, poster, dirname):
        #Resizeable input windows
        cv2.namedWindow('CLEAN', cv2.WINDOW_NORMAL)
        cv2.resizeWindow('CLEAN', poster.shape[0]/2, poster.shape[1]/2)
        cv2.moveWindow('CLEAN', 0, 0)
        #Instance variables
        self.img = poster
        self.save = poster
        self.img2= poster.copy()
        self.dirname = dirname
        self.GREEN = [0, 255, 0]
        self.HIDE = [249, 250, 250]
        self.ix, self.iy = -1, -1
        self.drawing = False

    def getcolor_hide_mouse(self, event, x, y, flags, param):
        #Pick BGR values of the pixel underneath mouse cursor
        if event == cv2.EVENT_RBUTTONDOWN:
            self.ix, self.iy = x, y
            print (self.ix, self.iy)
        elif event == cv2.EVENT_RBUTTONUP:
            pix = self.img[self.ix, self.iy]
            self.HIDE[0], self.HIDE[1], self.HIDE[2] = int(pix[0]),
int(pix[1]), int(pix[2])
            print (self.HIDE)
        # Draw rectangle
        elif event == cv2.EVENT_LBUTTONDOWN:
```

```
            self.drawing = True
            self.ix, self.iy = x, y     # stores mouse position in global
variables ix,iy
        elif event == cv2.EVENT_MOUSEMOVE:
            if self.drawing == True:
                self.img = self.img2.copy()       # refresh img to draw on
clean image
                cv2.rectangle(self.img, (self.ix,self.iy), (x,y),
self.GREEN, -1)
        elif event == cv2.EVENT_LBUTTONUP:
            self.drawing = False
            cv2.rectangle(self.img, (self.ix,self.iy), (x,y), self.HIDE, -1)
            self.img2 = self.img.copy()    # img2 will keep last drawn
rectangle

    def clean(self):
        cv2.setMouseCallback('CLEAN', self.getcolor_hide_mouse)
        while True:
            cv2.imshow('CLEAN', self.img)
            k = 0xFF & cv2.waitKey(1)
            # key bindings
            if k == 27:         # esc to exit
                break
            elif k == ord('r'): # reset all flags
                self.HIDE = [250, 250, 250]
                self.drawing = False
                self.img = self.save
                self.img2 = self.img.copy()
                print(" Values have been reset \n")
            elif k == ord('s'): # save image
                cv2.imwrite(os.path.join(self.dirname, 'pictwin.png'),
self.img)
                print("Saved in {}/pictwin.png".format(self.dirname))
                break
        cv2.destroyWindow('CLEAN')
    def get_image(self):
        return self.img
```

Intégration

Au niveau du script principal, l'intégration des fonctions de la classe **Clean** se fait aprés avoir créer un dossier de travail (lignes **9**-**13**) et avoir défini une variable qui contient notre image (ligne **16**). On pourra alors creer un objet *start*, instance de Cleaning, et appeler la fonction **clean()**

Instanciation d'un objet de la classe Cleaning: *pictwin.py*

```
# built-in modules
import cv2
import os
import shutil
```

```
import cleanImage
import identify
import display

if __name__ == '__main__':

    #Create working directory
    dirname = 'results'
    if os.path.exists(dirname):
        shutil.rmtree(dirname, ignore_errors=True)
    os.mkdir(dirname)
    #Read original file:
    poster = cv2.imread('src/twinIt.jpg',1)

    print ('Poster dim. and channels: {}'.format(poster.shape))
    #Cleaning poster
    print (cleanImage.__doc__)

    start = cleanImage.Cleaning(poster, dirname)
    start.clean()
    img = start.get_image()
    #For test: source image with hidden areas
#    img = cv2.imread('src/pictwin.png',1)
```

Quelques mots sur la docstring à laquelle on accède avec

```
print (cleanImage.__doc__)
```

C'est une chaîne de caractères un peu particulière car elle n'est pas assignée, mais elle consiste en un bloc de texte borné par trois doubles *quotes* [“”“] et placée en première position dans un bloc de définition, elle permet la documentation de modules, classes et fonctions, en décrivant leurs utilisations. L'attribut **__doc__** permet d’accéder à cette documentation.

Identification

L'identification des bulles suit un processus que nous avons commencé à détailler dans la partie précédente (Transformation morphologique). En effet l'image doit subir un certain nombre de traitement avant de pouvoir se lancer dans la détection des objets qui la compose.

Détection d'objets

Une fois qu'on a obtenu une image monochrome bien nette, nous pourrons localiser les bulles grâce à la fonction cv2.findContours(). Cette fonction retourne notamment une liste de coordonnées d'objets

d'une certaine intensité. Cette liste dépendant de la méthode d'approximation choisie (le troisième argument de la fonction) et consiste en la successions de coordonnées pour chaque objet détecté. En plus de cette liste, la fonction retourne une image ainsi qu'une hiérarchisation des objets détectés. Celle-ci permettant d'indiquer les relations de parenté entre les objets[14)]:
Une fois les coordonnées obtenues, il sera facile d'en tirer des informations comme les dimensions du rectangle contenant l'objet (grâce à la méthode cv2.boundingRect()) et d'extraire les bulles garce à l'usage d'opérations logiques comme nous l'avons vu précédemment.

Détection d'objets: identify.py

```
    def contour(self):
        #Copy gradient_mask_inv into gradient_mask_inv2: findContours
function modifies the source image
        gradient_mask_inv2 = self.mask()
        #Find countour
        imCont, contours, hierarchy =
cv2.findContours(gradient_mask_inv2,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPL
E)
        #return values using a tuple
        return (imCont, contours, hierarchy)
```

Détection d'objets: identify.py

```
    def print_id(self, mode=None):
        img_gray = cv2.cvtColor(self.img, cv2.COLOR_BGR2GRAY)
        #To draw colored id on grayscal img
        img_gray_3chan = cv2.cvtColor(img_gray, cv2.COLOR_GRAY2BGR)
        image_cont, contours, hierarchy = self.contour()
        #Cell iteration
        for i in range(0, len(contours)):
            cnt = contours[i]
            (centre_cnt_x,centre_cnt_y),radius = cv2.minEnclosingCircle(cnt)
            #Draw labels on
            font = cv2.FONT_HERSHEY_SIMPLEX
            im_with_label = cv2.putText(img_gray_3chan, str(i),
(int(centre_cnt_x),int(centre_cnt_y)), font, 1, (72,33,242), 2)
        if mode is None:
            return im_with_label
        else:
            cv2.imwrite(os.path.join(self.dirname, "Cells ID.png"),
im_with_label)
```

NB: Une autre approche du problème de détection des bulles a été entreprise au départ grâce à la méthode cv2.SimpleBlobDetector_create(). Celle-ci permet une grande précision quant aux régalages de la détection et constitue un outils très puissant. Ce type de fonction qui est utilisé en imagerie médicale en permettant d'identifier des cellules, de les compter etc... (une introduction est disponible sur http://www.labbookpages.co.uk/software/imgProc/blobDetection.html). Le script blobDetect.py montre l'utilisation de cette méthode. Il a fallu faire des choix et l'usage de findContour() s'est avéré suffisant pour notre problématique.

Recherche de paires

Discrimination positive

On peut voir que plusieurs fonctions sont définies dans le module **identify.py**. Ces fonctions ont été implémentée afin d'effectuer certains tests durant la phase de recherche de paires. En effet, le seul fait d'extraire une bulle et de faire une recherche de paires (*matching*) sur l'ensemble de l'image allait toujours retourner au moins une correspondance: celle de la bulle en cours d'appairage. Il nous fallait éliminer cette redondance. C'est ainsi que s'explique les 3 fonctions: **hide_cell() hidden_rgb()** et **hidden_gray()**. Le problème étant que malgré l'obturation de la bulle extraite sur l'image de travail, la fonction de recherche de paires cv2.matchTemplate() retournait inlassablement une redondance dans les résultats: la zone obturée était marquée positivement (rectangle rouge). Ces zones marquées sont obtenues à l'aide d'une liste de coordonnées retournées par cv2.matchTemplate(). Il n'a pas été possible de comprendre la raison de ce comportement.
Le moyen alternatif pour éliminer ces résultats aberrants est de calculer la distance entre le centre de la bulle en cours d'appairage [15)] et le centre du rectangle de la liste de localisations de cv2.matchTemplate(). Le rectangle signalant une zone de similarité n'est dessiné que si cette distance est supérieur au rayon du cercle qui entoure la bulle en cours d'appairage. Cette alternative est définie dans la fonction find_me().

Recherche de paires: matchCells.py

```
    def find(self):
        process = identify.Identify(self.img,self.dirname)
        _,contours,_=process.contour()
        for i in range(0, len(contours)):
            print (i)
            cnt = contours[i]
            x,y,w,h = cv2.boundingRect(cnt)
            cropped_cell = process.extract_cell_rgb(i)
            img_hidden = process.hidden(i)
            #Match cropped cell template and masked image: Greyscale
            res =
cv2.matchTemplate(img_hidden,cropped_cell,cv2.TM_CCOEFF_NORMED)
            threshold = 0.50
            loc = np.where(res >= threshold)
            nbMatch = 0
            img_rgb = self.img.copy()
            #Unpacking Argument Lists of loc
            for pt in zip(*loc[::-1]):
                nbMatch = nbMatch + 1
                #Draw rect on matched area
                cv2.rectangle(img_rgb, (x,y),(x+w,y+h), self.BLUE, 5)
                cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), self.RED,
5)
                if (nbMatch > 10):
                    match_file_name = 'Match-'+str(i)+'.png'
                    cv2.imwrite(os.path.join(self.dirname, match_file_name),
img_rgb)
```

```
                    break
    def find_i(self, i, th):
        process = identify.Identify(self.img,self.dirname)
        _,contours,_=process.contour()
        print (i)
        cnt = contours[i]
        x,y,w,h = cv2.boundingRect(cnt)
        cropped_cell = process.extract_cell_rgb(i)
        img_hidden = process.hidden_rgb(i)
        #Match cropped cell template and masked image: Greyscale
        res =
cv2.matchTemplate(img_hidden,cropped_cell,cv2.TM_CCOEFF_NORMED)
        threshold = th
        loc = np.where(res >= threshold)
        nbMatch = 0
        img_rgb = self.img.copy()
        cv2.rectangle(img_rgb, (x,y),(x+w,y+h), self.BLUE, 6)
        #Unpacking Argument Lists of loc
        for pt in zip(*loc[::-1]):
            nbMatch = nbMatch + 1
            #Draw rect of matched area
            cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), self.RED, 3)
        print (nbMatch)
        return img_rgb
    def find_me(self, cell_id, th):
        def distance(xa,ya,xb,yb):
            return math.sqrt((xb-xa)**2+(yb-ya)**2)
        process = identify.Identify(self.img,self.dirname)
        _,contours,_=process.contour()
        cnt = contours[cell_id]
        x,y,w,h = cv2.boundingRect(cnt)
        (centre_cnt_x,centre_cnt_y),radius = cv2.minEnclosingCircle(cnt)
        cell = self.img[y:y+h,x:x+w]
        img_rgb = self.img.copy()

        #Match cropped cell template and masked image: Greyscale
        res = cv2.matchTemplate(self.img,cell,cv2.TM_CCOEFF_NORMED)
        threshold = th
        loc = np.where(res >= threshold)
        nbMatch = 0
        #Show query cell in blue
        cv2.rectangle(img_rgb, (x,y),(x+w,y+h), self.BLUE, 6)
        #Unpacking Argument Lists of loc
        for pt in zip(*loc[::-1]):
            nbMatch = nbMatch + 1
            #Get the center of current matching rectangle
            centerTx=((2*pt[0]+w)/2)
            centerTy=((2*pt[1]+h)/2)
            #Distance between both center
            calc_dist =
distance(centre_cnt_x,centre_cnt_y,centerTx,centerTy)
```

```
            #Only draw matching area
            if calc_dist > radius:
                #Draw rect of matched area
                cv2.rectangle(img_rgb, pt, (pt[0] + w, pt[1] + h), self.RED,
3)
        print ('{} matched areas'.format(nbMatch))
        return img_rgb
```

Template matching

La methode cv2.matchTemplate() a été le principal outil de travail pour détecter les zones de similarité. Cette méthode fonctionne assez simplement en comparant un *template* à une image source, et ce, en faisant translater le *template* pixel à pixel à travers l'image source. A chaque localisation, on calcule une “distance” représentant le niveau de similarité entre le voisinage d'un pixel du *template* et la zone en cours d'évaluation de l'image source. Ce qui permet d'obtenir une image en niveau de gris dont chaque pixel illustre le niveau de similarité avec le *template*, la *distance* est le fruit d'un calcul qui dépend du troisième argument de la fonction cv2.matchTemplate().

Par exemple, dans le cas où le troisième argument est **cv2.TM_SQDIFF**, le calcul suivant sera appliqué afin d'obtenir la *distance* de similarité:

$$R(x,y) = \sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2$$

Dans cette formule, **R** est la matrice contenant le résultat de similarité. Chaque élément de **R**, contiendra une valeur dépendant de la différence élevée au carré entre valeurs des pixels du *template* (**T(x',y')**) et valeurs des pixels de l'image source et de leur voisinage(**I(x+x',y+y')**). Pour ce type de méthode de comparaison, une similarité parfaite donnera une valeur de 0.

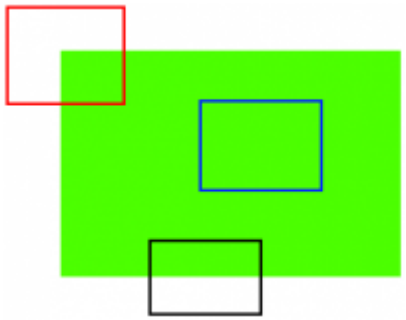
Mais nous utilisons une autre méthode de comparaison, la corrélation croisée qui permet de la même manière de mesurer la similitude entre deux signaux. Nous utilisons la version **normalisée** de cette corrélation croisée qui a pour effet d’atténuer les différences de luminosité entre *template* et image source. Dans notre cas, une parfaite similarité entre *template* et image source induira une valeur égale à 1, tandis que les zones parfaitement dissemblables tendrons vers -1.
On peut alors mettre en valeur les zones de l'image qui sont plus ou moins similaires à une bulle “requete” et cela en fonction d'un seuil, **threshold**:

```
loc = np.where(res >= threshold)
```

Ainsi **loc** sera une liste contenant les coordonnées des pixels au niveau de similarité supérieur au seuil minimum **threshold**. En utilisant, les dimensions du rectangle *template*, on va marquer la zone nouvellement trouvée à condition que cette zone ne soit pas dans le rayon de la bulle en cours d'appairage.

Feature detection

Une autre possibilité pour la recherche de paires aurait été d'utiliser les fonctions de détection de zones d'intérêts (*feature detection*). Cette méthode détermine des points d'intérêts significatifs (*key points*) sur les différentes images et les trie. Ainsi, les bulles fortement similaires sont détectées. Ces *key points* representent les points facilement identifiables. Dans l'exemple ci-dessous, on peut dire que le coin superieur gauche est plus facilement localisable et identifiable que la zone entourée en bleu:

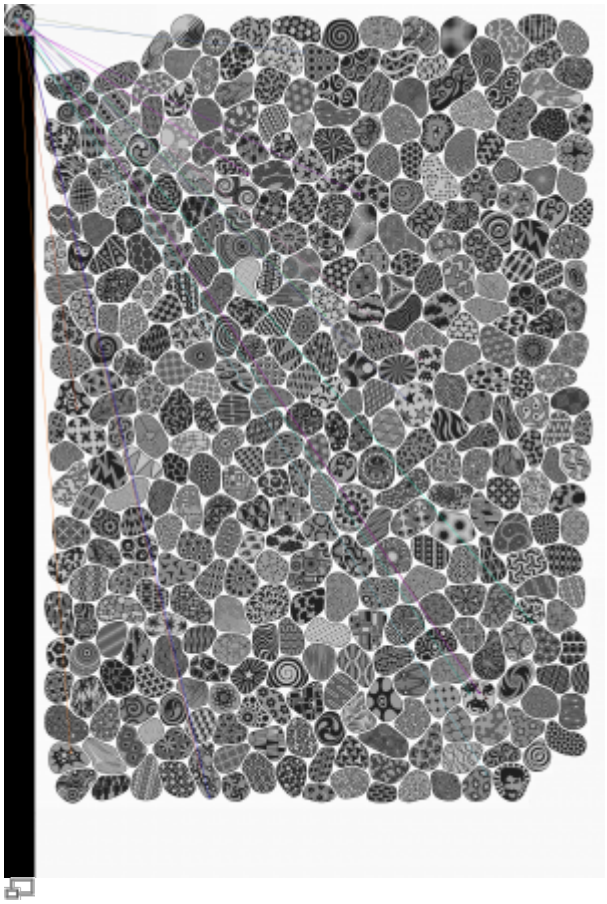


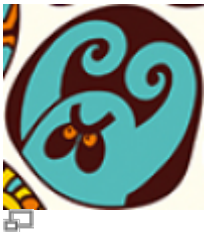
Ce principe a été testé en définissant la fonction **matching()** qu'on peut retrouver dans le dossier d'archive.

Recherche de paires: featureMatching.py

```
#fonction de comparaison
def matching(ROI,Poster,filename):
    img1 = ROI #Image requête
    img2 = Poster #Image globale
    #This is the detector we're going to use for the features.
    orb = cv2.ORB_create()
    #Here, we find the key points and their descriptors with the orb
detector.
    cv2.ocl.setUseOpenCL(False) #Pour corriger l'erreur lors de l'appel de
detectAndCompute
    kp1, des1 = orb.detectAndCompute(img1,None)
    kp2, des2 = orb.detectAndCompute(img2,None)
    #This is our BFMatcher object.
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    #Here we create matches of the descriptors, then we sort them based on
their distances.
    matches = bf.match(des1,des2)
    matches = sorted(matches, key = lambda x:x.distance)
    print matches
    result = cv2.drawMatches(img1, kp1, img2, kp2, matches[:100], None,
flags=2)
    plt.imshow(cv2.cvtColor(result, cv2.COLOR_BGR2RGB))
    plt.show()
    cv2.imwrite(filename, result)
```

Cette méthode va malheureusement retourner des résultats inexploitables. En effet, un certain nombre de textures sont trop similaires les unes aux autres, ce qui rend la détection de paires impossible. On peut voir cette problématique sur les illustrations ci-dessous:





Bulle

On remarque la disparité des résultats

Ici les résultats d'un test mettant en avant les avantages d'une telle méthode

Cette méthode ayant pour avantage de permettre la détection de zone de similarités sans souffrir des problèmes de rotations ou de changement d'échelle. Par ailleurs, cette méthode est très souvent utilisée dans la création d'image panoramique à partir de plusieurs clichés ou bien dans la détermination d'objets en mouvement dans une scène.
C'est l'un des regrets de ce projet, qui est de ne pas avoir pu exploiter cet outil et ni de trouver un moyen de contraindre les résultats retournés à des zones moins disparates.

Affichage

L'interface de ce projet et notamment la présentation graphique restent trés sommaires. Néanmoins, grâce à un outil simple et implémenté dans OpenCV, les *Trackbars*, il a été possible de rendre pictwin

un peu plus interactif. Une esquisse a été réalisée grâce à PyQt4 afin d'élaborer une interface proposant l'importation d'images, la création d'un menu, etc... Cependant, la liaison entre les fonctions des différents modules et les élements graphiques est manquante.
Quant au module **display.py**, il crée, dans un premier temps, des objets des classes Matching et Identify, définis précédement. La sélection d'un identifiant de bulle se fait grâce à la *trackbar* supérieure (ou aux touches 8 et 2 du pavé numérique), la séléction du seuil de similarité, se fait avec la *trackbar* inférieur (ou aux touches 4 et 6 du pavé numérique). Pour aider à la sélection de bulle, nous affichons en paralléle une fenêtre 'Cell label' qui indique les identifiants de chaque contour. On appelle alors la fonction **find_me()** en appuyant sur **m**.

Affichage: display.py

```
class Result(object):
    def __init__(self, image, dirname):
        self.img = image
        self.dirname = dirname
    def play(self):
        """play() function doc
            Display matching areas found for a given cell #id
            Use trackbars to setup values for cells #id and Accurateness
level
            NB: You can use NUMPAV keys also:
                8 and 2  for cell #id
                4 and 6  for Accurateness level
            Press 'm' to display matching area
        """
        match_getter = matchCells.Matching(self.img, self.dirname)
        contours_getter = identify.Identify(self.img, self.dirname)
        _,Contours,_ = contours_getter.contour()
        nb_Contours = len(Contours)-1
        def nothing(x):
            pass
        # Create a black image, a window
        disp = self.img
        #Display debugging
        cv2.namedWindow('picTwin', cv2.WINDOW_KEEPRATIO)
        cv2.moveWindow('picTwin', 0, 0)
        cv2.resizeWindow('picTwin', 800, 1200)
        #Identify Cells
        labels = contours_getter.print_id(mode=None)
        cv2.namedWindow('Cell labels', cv2.WINDOW_NORMAL)
        cv2.imshow('Cell labels', labels)
        # create trackbars for cell id selection
        cv2.createTrackbar('Cell #Id','picTwin',0,nb_Contours,nothing)
        # create trackbars for threshold selection
        cv2.createTrackbar('Precision','picTwin',0,20,nothing)
        cv2.imshow('picTwin',disp)
        while (1):
            cv2.imshow('picTwin',disp)
            k = 0xFF & cv2.waitKey(1)
            # key bindings
```

```
            if k == 27:         # esc to exit
                break
            elif k == ord('m'): # matching
                cellId = cv2.getTrackbarPos('Cell #Id','picTwin')
                step = cv2.getTrackbarPos('Precision','picTwin')
                th = step / float(20)
                disp = match_getter.find_me(cellId, th)
                print("Results for Cell #{} and similarity level= {}%
\n".format(cellId, th*100))
            elif k == ord('8'):
                pos = cv2.getTrackbarPos('Cell #Id','picTwin')
                pos = pos+1
                cv2.setTrackbarPos('Cell #Id','picTwin', pos)
            elif k == ord('2'):
                pos = cv2.getTrackbarPos('Cell #Id','picTwin')
                pos = pos-1
                cv2.setTrackbarPos('Cell #Id','picTwin', pos)
            elif k == ord('6'):
                pos = cv2.getTrackbarPos('Precision','picTwin')
                pos = pos+1
                cv2.setTrackbarPos('Precision','picTwin', pos)
            elif k == ord('4'):
                pos = cv2.getTrackbarPos('Precision','picTwin')
                pos = pos-1
                cv2.setTrackbarPos('Precision','picTwin', pos)
            elif k == ord('r'): # reset
                disp = self.img

        cv2.destroyAllWindows()
```

Afin de tester des résultats positifs, on fournit les identifiants de quelques bulles jumelles ainsi que du niveau de précision maximum retournant une paire:

Bulle source	**Bulle jumelle**	**Précision**
171	355	8
355	171	6
160	299	12
299	160	10
285	161	9
161	285	9

NB: Ces identifiants sont ceux obtenus pour une image parfaitement nettoyée et qui contient les 390 bulles.

[12] **HIDE** est dans un premier temps initialisée avec des valeurs BGR=[249,250,250]. Mais ces valeurs pourront être mises à jour à la manière d'une pipette à couleur par l'usager.
[13] L'initialisation de la fenêtre lorsqu'on instancie notre objet
[14] Plus de détails sur cette hiérarchisation et le second argument de la fonction cv2.findcontours() sont disponibles sur la documentation

http://docs.opencv.org/3.1.0/d9/d8b/tutorial_py_contours_hierarchy.html.
[15] grâce à cv2.boundingRect(cnt)

Conclusion

La tâche la plus ardue de tout projet est de savoir où et quand mettre le point final, tâche qui est d'autant plus difficile lorsqu'on n'atteint pas l'objectif de départ. Mais à l'heure du bilan, il faut aussi savoir reconnaitre la satisfaction d'avoir trouver des solutions pertinentes à une problématique pas si évidente au départ. Il est difficile de sauvegarder le raisonnement qui nous a fait préférer une solution à une autre. Le plus souvent quelques lignes du script ont été l'aboutissement de nombreuses heures de recherche et de lecture des documentations des différents outils.
Le but initial été de limiter l'interaction avec l'usager et d'obtenir un programme autonome qui retournerait les résultats, en mettant en valeurs les paires deux à deux par exemple. Mais la grande similarité entre certaines bulles, la rotation des motifs les uns par rapport aux autres invite à réfléchir à d'autre moyens de mise en évidences des doubles. De plus, ce projet à été l'occasion d'aborder le traitement d'images et d'en définir certains concepts. Nous avons pu notamment découvrir certaines spécificités de Python et utiliser de nombreuses méthodes proposées dans la bibliothèque OpenCV. Ainsi, la production d'une application qui reste cohérente permettant d'obtenir les zones à forte similarité a été réalisée.

Sources

Traitement d'image

- http://www.i3s.unice.fr/~mh/RR/2004/RR-04.05-D.LINGRAND.pdf
- http://www.labbookpages.co.uk/software/imgProc/blobDetection.html
- http://images.math.cnrs.fr/Le-traitement-numerique-des-images.html

OpenCv

- http://docs.opencv.org/3.1.0/
- http://docs.opencv.org/trunk/d9/d8b/tutorial_py_contours_hierarchy.html#gsc.tab=0
- http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html
- http://stackoverflow.com/questions/33729432/c-biological-cell-counting-with-opencv
- http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_gui/py_trackbar/py_trackbar.html
- http://lisa.ulb.ac.be/image/index.php/Programmation_en_Python#Analyse_d.27images_et_reconnaissance_de_formes
- https://www.learnopencv.com/blob-detection-using-opencv-python-c/
- http://docs.opencv.org/trunk/d7/d4d/tutorial_py_thresholding.html#gsc.tab=0
- http://answers.opencv.org/question/10186/how-to-save-rois-detected-in-an-image/
- https://en.wikipedia.org/wiki/Connected-component_labeling
- http://homepages.inf.ed.ac.uk/rbf/HIPR2/guidecon.htm
- http://machinelearningmastery.com/using-opencv-python-and-template-matching-to-play-where s-waldo/

Python

- http://lisa.ulb.ac.be/image/index.php/Programmation_en_Python
- https://deptinfo-ensip.univ-poitiers.fr/ENS/doku/doku.php/stu:python_gui:tuto_images
- https://docs.python.org/release/2.7/
- http://nbviewer.jupyter.org/github/gestaltrevision/python_for_visres/blob/master/Part1/Part1_Intro_to_Python.ipynb
- http://www.mon-club-elec.fr/pmwiki_mon_club_elec/pmwiki.php?n=MAIN.PYQTLABOpenCVFichiersLoadFileSelect
- http://pyqt.developpez.com/tutoriels/
- http://stackoverflow.com/questions/354883/how-do-you-return-multiple-values-in-python
- https://larlet.fr/david/biologeek/archives/20080511-bonnes-pratiques-et-astuces-python/