

Réalisation d'un panorama

LUDOVIC LAMARCHE
QUENTIN PERALES
ELIE POUSSOU

E.I.S.T.I
PAU

19 janvier 2014

Table des matières

1	Les six fonctions du deuxième livrable	3
1.1	Grayscale	3
1.2	Binnary	4
1.3	Histogramme	4
1.4	Dilate	5
1.5	Erode	6
1.6	Convolution [4]	7
2	Notre algorithme d'automatisation d'un panorama	9
2.1	Récupération des points clé	9
2.2	Algorithme de comparaison des points clés	15
2.2.1	Schématisation de notre algorithme	15
2.2.2	Algorithme	18
2.3	Fonction de collage des images	19
2.4	Notre algorithme principal	21
2.5	Comment améliorer cet algorithme ?	23

Introduction

Le projet se conclue par ce dernier livrable en fin de semestre. L'objectif est donc de créer un programme qui assemble des images ou des photos pour réaliser un panorama. Pour cela, il a fallu mettre en oeuvre toutes nos connaissances accumulées au cours des deux premiers mois, ainsi que nos recherches pour remplir l'objectif fixé par ce projet.

Dans un premier temps, nous allons présenter les fonctions réalisées pour le second livrable car elles ont été très utiles pour la réalisation de notre algorithme codé durant cette dernière partie.

Dans un second temps, nous allons présenter les étapes ainsi que les fonctions principales de notre algorithme qui relie les images entre elles. Puis, nous expliquerons les améliorations que nous aurions pu apporter au programme.

Chapitre 1

Les six fonctions du deuxième livrable

1.1 Grayscale

La fonction Grayscale permet de transformer une image couleur en teintes de gris.

Dans une image en couleur, chaque pixel est codé par 3 composantes (Rouge, Vert et Bleu).

En grayscale, chaque pixel sera codé par une seule composante.

La formule utilisée est la suivante :

$$pixelGrayscale = 0,299 * pixelRouge + 0,587 * pixelvert + 0,114 * pixelbleu$$

On parcourt donc l'image couleur, et on applique cette formule à chaque pixel, c'est à dire à ces trois composantes.

Cette fonction effectue donc une moyenne pondérée des pixels rouges, verts et bleus.

L'image obtenue sera donc 3 fois plus petite, et en noir et blanc.

1.2 Binnary

Chaque pixel d'une image binaire ne peut avoir pour valeurs que 0 ou 1. La fonction binary nécessite un seuil et une image en teintes de gris, le seuil étant choisi par l'utilisateur.

On parcourt l'image en teintes de gris, et pour chaque pixel, si son intensité est supérieure au seuil, on lui alloue la valeur 0 qui correspond à la couleur blanche. Si l'intensité du pixel est supérieure au seuil, on lui alloue la valeur 1 qui correspond à la couleur noire.

Voici l'algorithme utilisé :

```
1   POUR TOUT les pixels de l'image
2       SI le pixel >seuil
3           pixel=1
4       FINSI
5   FINPOUR
```

1.3 Histogramme

L'histogramme d'une image permet d'illustrer la répartition des teintes d'une image.

Le but est de compter pour chaque intensité le nombre de pixels qui sont de cette intensité.

D'abord on crée un tableau de taille égale à la teinte maximale de l'image dont on veut obtenir l'histogramme.

Ensuite, on parcourt l'image et pour chaque pixel, on rajoute 1 dans la case du tableau qui correspond à l'intensité de ce pixel.

On peut résumer tout ceci grâce à l'algorithme suivant :

```

1 tab : tableau d'entiers de taille égale à la teinte maximale de
      l'image

3 POUR chaque pixel de l'image
    tab [teinte_pixel]++
5 FINPOUR
```

On peut choisir de tracer cet histogramme mais le tableau suffit.

1.4 Dilate

Pour la dilatation nous avons utilisé un algorithme qui utilise des "objects pixel" tiré du livre Machine Vision[3]. Un objet pixel est un groupe de neuf pixels formant un carré de taille 3x3 et qui respecte certaines conditions. Pour simplifier l'algorithme on nomme chaque case de 0 à 8.

3	2	1
4	8	0
5	6	7

Si on utilise les conventions de logique combinatoire, c'est à dire que le signe . signifie ET et le signe + signifie OU, les conditions d'un objet pixel se résume comme ceci :

$$\begin{aligned}
 & 8.[((1 + 2 + 3).5 + 6 + 7).\bar{4}.\bar{0}] \\
 & +[(1 + 0 + 7).(3 + 4 + 5).\bar{2}.\bar{6}] \\
 & +[3.(5 + 6 + 7 + 0 + 1).\bar{2}.\bar{4}] \\
 & +[1.(3 + 4 + 5 + 6 + 7).\bar{2}.\bar{0}] \\
 & +[7.(1 + 2 + 3 + 4 + 5).\bar{0}.\bar{6}] \\
 & +[5.(7 + 1 + 2 + 3).\bar{4}.\bar{6}]
 \end{aligned}$$

Finallement, il suffit d'appliquer l'algorithme suivant :

```
1 POUR TOUT les pixels de l'image
    SI le pixel est un objet pixel ALORS
        copier le groupe de pixels dans l'image de destination
    FINSI
5 FINPOUR
```

Étant donné que cette méthode enlève beaucoup de pixels, il n'est pas nécessaire de faire une érosion en suivant. Cependant nous avons utilisé un autre algorithme moins restrictif pour l'érosion.

1.5 Erode

Pour l'érosion nous utilisons un masque. Nous testons un groupe de pixel et nous assignons la valeur 1 si tout les pixels dans le masque sont à 1. Sinon on retourne un 0. Nous avons pris directement l'algorithme d'opération pixel[2] :

```
1 destination : matrice de l'image érodée
    source : matrice de l'image à éroder
3 x,y,k1,k2 : variables d'incrémentation

5 POUR CHAQUE pixel de l'image FAIRE //x et y étant la position
    POUR k1 allant de -1 à 1 FAIRE
        POUR k2 allant de -1 à 1 FAIRE
            destination[x][y] = destination[x][y] ET source[x+k1][y+k2]
9     FINPOUR
    FINPOUR
11 FIN POUR
```

1.6 Convolution [4]

La convolution permet d'appliquer un filtre de convolution à une image en échelle de gris. Le but est d'affecter pour chaque pixel de l'image des coefficients aux pixels autour de celui-ci.

La taille du filtre est variable, cependant, elle est toujours paire. Les filtres les plus utilisés sont les filtres de taille 3x3, mais parfois, les filtres de tailles supérieures sont requis. Les filtres sont récupérés en début de fonction, une erreur est générée si le filtre n'est pas correct, c'est à dire si le fichier texte comporte des caractères autres que des chiffres ou nombres ou s'il n'y a pas assez de coefficients.

Afin d'expliquer l'algorithme qui mène à la convolution, nous allons utiliser un filtre 3x3 sur une image. Prenons le filtre F suivant :

$$\begin{matrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$$

F est appliqué à un pixel p de coordonnées (x, y). La valeur du pixel après la convolution sera alors

$$\begin{aligned} nouvelleValeur = & 0 * p(x - 1, y - 1) + 1 * p(x, y - 1) + 0 * p(x + 1, y - 1) \\ & + 1 * p(x - 1, y) + 1 * p(x, y) + 1 * p(x + 1, y) \\ & + 0 * p(x - 1, y + 1) + 1 * p(x, y + 1) + 0 * p(x + 1, y + 1) \end{aligned}$$

De plus, on crée un décalage pour éviter d'assigner des coefficients à des pixels hors image. Pour un filtre 3x3, le décalage est de 1 ; pour 5x5, il est de 3 ; pour 7x7, il est de 5. On peut donc dire que

$$decalage = \frac{taille - 1}{2} \quad (1.1)$$

Voici un exemple de convolution que l'on peut obtenir :

$$\begin{array}{ccccccccc} 35 & 40 & 41 & 45 & 50 & & 35 & 40 & 41 & 45 & 50 \\ 40 & 40 & 42 & 46 & 52 & 0 & 1 & 0 & 35 & 40 & 41 & 45 & 50 \\ 42 & 46 & 50 & 55 & 55 & X & 0 & 0 & 0 & 40 & 40 & 42 & 46 & 52 \\ 48 & 52 & 56 & 58 & 60 & 0 & 0 & 0 & 42 & 46 & 50 & 55 & 55 \\ 56 & 60 & 65 & 70 & 75 & & & & 56 & 60 & 65 & 70 & 75 \end{array}$$

Après avoir vu certaines fonctions très utiles dans le traitement d'image, nous allons maintenant voir comment les enchaîner pour les rendre utiles lors de la création du panorama.

Chapitre 2

Notre algorithme d'automatisation d'un panorama

Dans cette partie, nous allons expliquer plusieurs fonctions principales dans notre algorithme.

Tout d'abord, la récupération des points clés. Ensuite, la comparaison de ces points clés entre les images. Puis, la fonction de collage qui permet d'assembler deux images connaissant le décalage. Enfin, expliquer l'enchaînement de ces fonctions pour aboutir à notre programme.

Pour finir cette partie, nous expliquerons les améliorations que nous pourrions apporter à notre code.

2.1 Récupération des points clé

La récupération des points clés est une étapes cruciale dans la réalisation d'un panorama car ils permettent de savoir ce que l'image a de particulier, et de connaître les formes qu'on pourrait comparer avec une autre image.

Pour cela, nous avons mis en oeuvre plusieurs algorithmes que nous avons trouvé lors de nos recherches. Trois algorithmes majeurs ont été trouvés.

Tout d'abord, la transformée de Hough devait nous permettre de récupérer les points issus de rencontre de droites ou de cercles. Cependant, lors de la comparaison de formes compliquées, les points trouvés n'étaient pas assez

précis et suffisants pour permettre une comparaison efficace.

Puis, nous avons codé le décodeur de Harris qui s'appuie sur des calculs mathématiques, notamment les matrices pour récupérer des points clés. Ce détecteur n'a pas toujours données des points pouvant être utilisés.

Ensuite, nous avons commencé à coder la méthode SIFT, une méthode très efficace, mais très longue.

Entre temps, nous avons trouvé un autre moyen pour trouver des points clé, qui utilisait les fonctions expliquées dans la partie précédente.

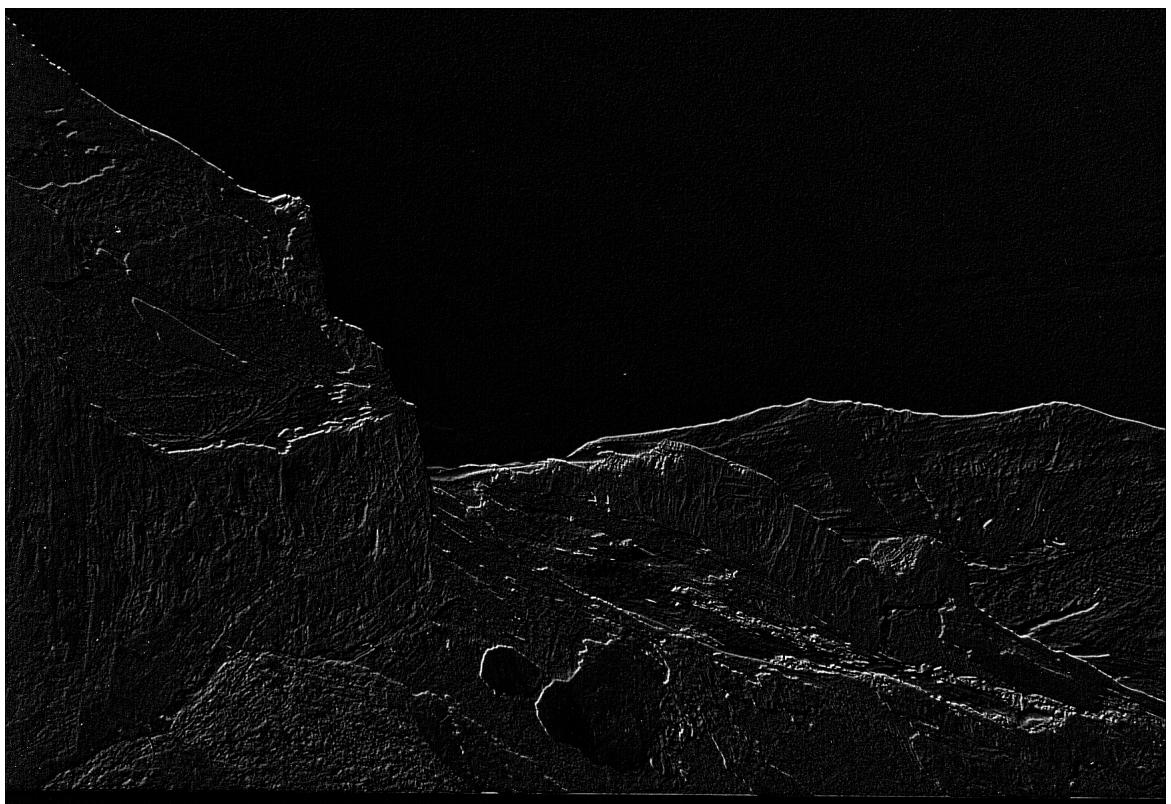
Raisonnons à partir de cette image :



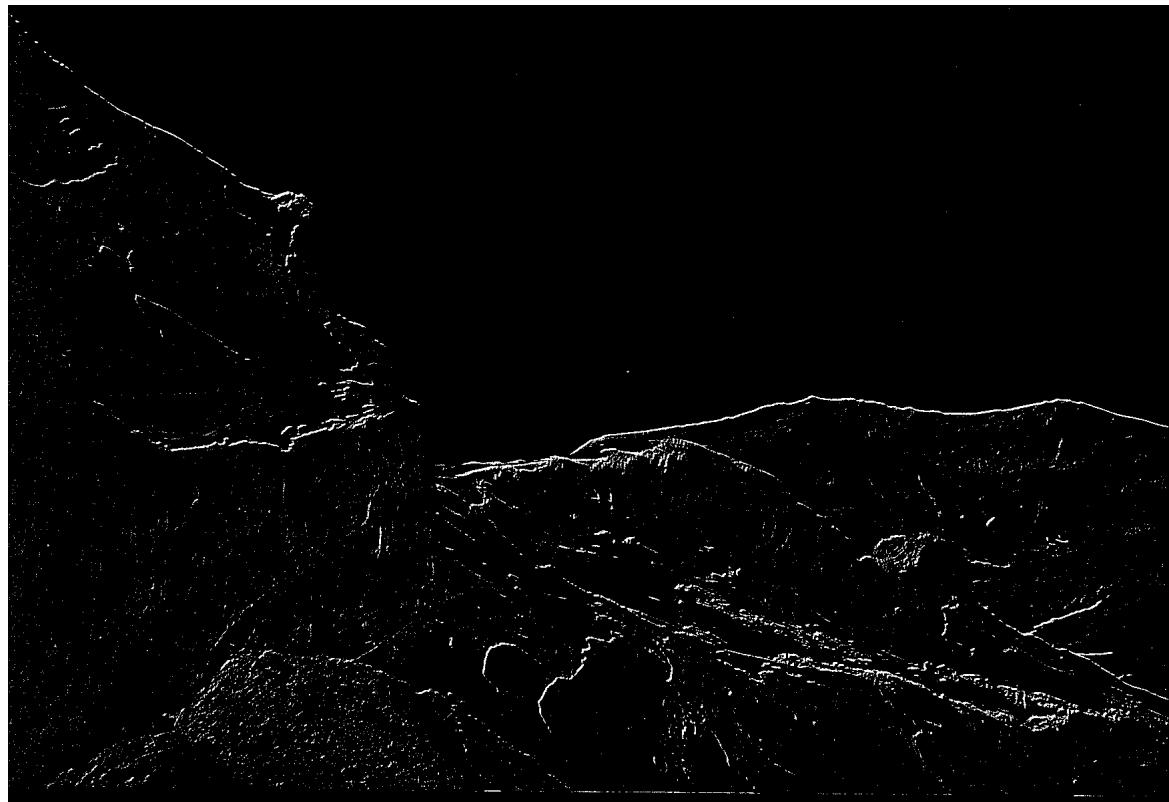
Pour cela, à partir d'une image en couleur, il faut la passer en greyscale puis effectuer une convolution avec un filtre de Prewitt.

Cela permet de détecter les contours tout en limitant l'effet du bruit sur la photo.

$$\begin{array}{ccc} 1 & 1 & 1 \\ \text{filtre de Prewitt : } & 0 & 0 & 0 \\ & -1 & -1 & -1 \end{array}$$



Ensuite, on converti cette image en binaire avec un seuil de 100.



Enfin, on opère à une dilatation puis une érosion cinq fois de suite pour enlever les points esseulés.



Pour finir cette détection, on parcourt l'image pour voir si un point blanc possède quinze voisins dans un carré de 15x15 pixels autour de lui. Cela peut s'apparenter à une forme de dilatation. Cette dernière opération est très utile lors de l'algorithme de comparaison des points clés expliqué ci-dessous.



2.2 Algorithme de comparaison des points clés

Pour trouver les points clés, nous avons d'abord fait beaucoup de recherches sur internet. Une des méthodes citées était le ZNCC, un algorithme qui donne une note entre deux points. Il suffisait donc avec cette méthode de comparer cette note. Cependant nous l'avons testé et il y avait trop de similitudes entre les points. La deuxième méthode trouvée était de comparer les valeurs de Harris. Or harris ne nous satisfaisait pas pour la recherche des points clés donc nous l'avons abandonné. Finalement nous avons arrêté nos recherches sur internet pour réfléchir de nous même et à concevoir un algorithme simple pour comparer nos points clés.

2.2.1 Schématisation de notre algorithme

Notre point de départ était de comparer des vecteurs dans chaque image. Nous avions d'abord essayé de comparer un seul vecteur sur des images coupées à l'ordinateur et le résultat était très satisfaisant. Par contre quand nous avons essayé cette méthode sur des photos cela n'a pas été concluant.

Recherches

Dans l'idée de comparer des vecteurs, nous avons d'abord essayé d'assembler deux photos à l'aide du logiciel Gimp. Voici le rendu :



De loin, le résultat n'est pas trop mal mais lorsqu'on se rapproche, on distingue clairement que si on aligne une partie de l'image, l'autre partie est complètement décalée. Ce phénomène est dû à la distorsion de l'appareil photo. Nous avons donc recherché un moyen de corriger cette distorsion. Pour cela nous avons utilisé la correction de Lens[1].

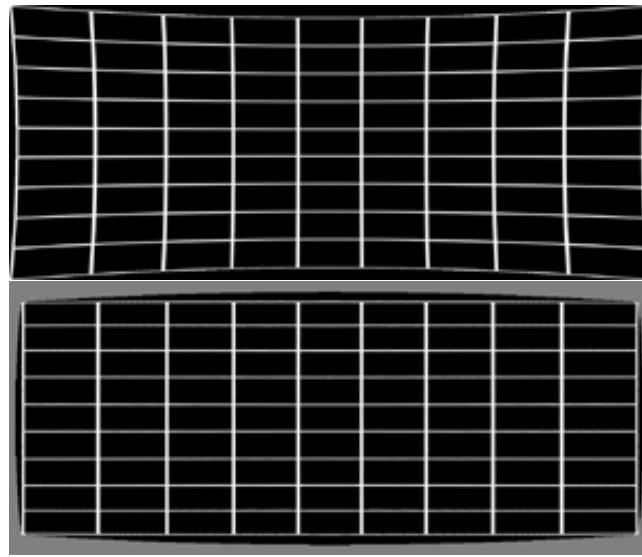


FIGURE 2.1 – Correction de Lens[1]

Algorithme de Lens :

```

1 Pre-conditions : coordonnées x et y, largeur et hauteur de l',
      image
Post-condition : retourne les nouvelles coordonnées
3 Fonction LensCorrection
    px = (2.0*x - largeur) / largeur;
5    py = (2.0*y - hauteur) / hauteur;
    r = px^2 + py^2;
7    p2x = px * (1.0 - 0.02 * r);
    p2y = py * (1.0 - 0.08 * r);
9    x = (p2x + 1)*largeur/2;
    y = (p2y+1)*hauteur/2;
11   FinFonction

```

Nous avons testé cet algorithme sur plusieurs tirages et il corrigeait très bien cette distorsion. Pour vérifier cela, nous avons de nouveau coller nos images après la correction avec Gimp, et les images s'assemblaient parfaitement.

Sélection des vecteurs

«««< HEAD Une fois que les images étaient prêtes, nous avons pu réfléchir à comment comparer les vecteurs. Nous avons choisi de sélectionner 20 vecteurs de même origine dans une fenêtre de 30px sur 30px. Nous avons donc parcouru la première image et pour chaque point clé nous avons généré un tableau de 20 points proche de ce point clé. Avec ce tableau nous avons pu calculer nos 20 vecteurs. ===== Une fois que les images étaient prêtes, nous avons pu réfléchir à comment comparer les vecteurs. Nous avons choisi de sélectionner 20 vecteurs de même origine dans une fenêtre de 30px sur 30px. Nous avons donc parcouru la première image et pour chaque point clé nous avons générer un tableau de 20 points proche de ce point clé. Avec ce tableau nous avons pu calculer nos 20 vecteurs. »»»>
442d943ead09ea0abb5cf58867639ea9d3e8211b

Recherche des correspondances

Pour trouver des correspondances, il faut parcourir la deuxième image, et vérifier pour chaque point clé si les vecteurs calculés précédemment tombent sur un autre point clé ou non. Pour parcourir les points clés, nous avons choisi d'utiliser des listes pour réduire la complexité du parcourt. Cependant pour vérifier que les coordonnées d'un vecteur tombe sur un point clé, nous avons utilisé une matrice pour accéder directement aux coordonnées données plutôt que de parcourir la liste jusqu'à trouver les coordonnées.

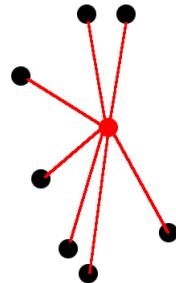


FIGURE 2.2 – Exemple de model de vecteurs

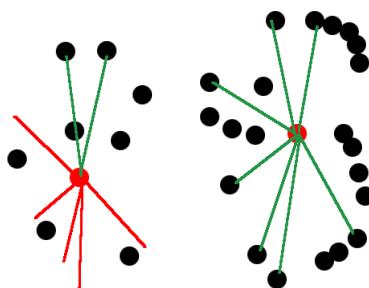


FIGURE 2.3 – Schéma de la recherche des correspondances

2.2.2 Algorithme

```

Pre-conditions :
2   Liste point clé image1
    Liste point clé image2
4   Matrice image 2

6 FONCTION comparer
  Pour chaque point clé de la liste 1
    tableau = Générer un tableau de 20 points proches
    Pour chaque point de la liste 2
      Pour chaque élément de tableau tant que TROUVE
        vecteur = tableau[i]-tableau[0]
        TROUVE = !matriceImage2[list2+vecteur]
        // si le point de la matrice est à 0, c'est un point clé
      FinPour

```

```

16   Si TROUVE
17     AjoutListe( CalculDécalage( tableau[0] , liste2 ) )
18   FinPour
19   FinPour
20   On retourne le décalage le plus présent
21 FinFonction

```

2.3 Fonction de collage des images

Nous voulons coller les deux images A et B. Ces deux images ont des points clefs en commun. Grâce à ces points clefs en commun, on peut calculer le décalage de la première image par rapport à l'autre.
C'est ce décalage que va utiliser l'algorithme de collage.

Il y a grossièrement deux cas de figure, soit les décalages en abscisse et en ordonnées sont positifs, soit le décalage en abscisse est positif et celui en ordonnée négatif.

Pour expliquer cet algorithme, nous allons nous concentrer sur le premier cas.

Il faut d'abord calculer la taille de l'image finale.

Calcul de la taille de l'image finale dans le cas où les décalages en abscisse et en ordonnée sont positifs :

```

FONCTION taille
2 {
3   largeur=max(largeurImage1 , decalage.x+largeurImage2);
4   hauteur=max(hauteurImage1 , decalage.y+hImage2);
5   On retourne la taille de l'image finale en hauteur et en largeur.
6 }

```

Après cette étape, on peut passer au collage à proprement parler.
 On obtient donc un plan dans lequel il faut faire rentrer ces deux images.
 On initialise donc d'abord l'image de la taille obtenue avec des pixels noirs.
 On écrit ensuite la première image à partir du point A(0,0) qui est l'origine.
 Ensuite on écrit la deuxième image à partir du point B(decalage.x, decalage.y).

```

FONCTION fusion
2 {
    matriceImageFinale=initMatrice (0 , largeur , hauteur );
4   POUR ( i=0; i<largeurImage1 ; i++)
    {
6     POUR ( j=0;j<hauteurImage1 ; j++)
     {
8       matriceImageFinale=image1 . teinte [ j ] [ i ];
      }
10    }
12   POUR ( i=decalage . x ; i<decalage . x+largeurImage2 ; i++)
14     {
16       POUR ( j=decalage . y ,j<hauteurImage2 , j++)
18         {
20           matriceImageFinale=image2 . teinte [ j-decalage . y
] [ i-decalage . x ];
           }
         }
       }
     }
   }
}
On retourne la matriceImageFinale
}

```

Le cas où le décalage en ordonnée est négatif, la méthode est équivalente sauf que l'on colle l'image 1 à partir du point A (0, decalage.y) et on colle l'image 2 à partir du point B(decalage.x,0).

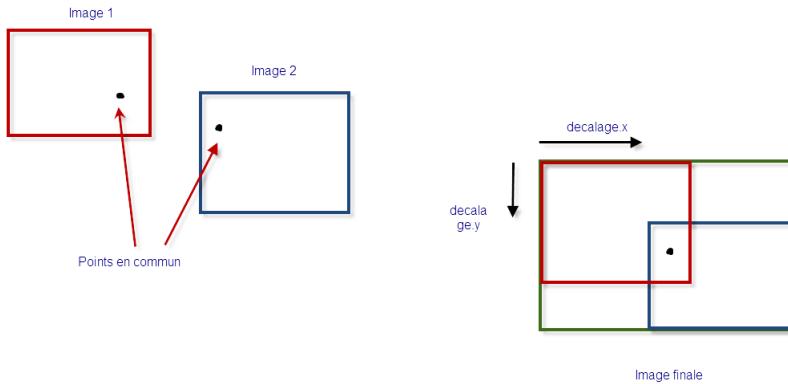


FIGURE 2.4 – Collage d’images dans les cas où les décalages en abscisse et en ordonnée sont positifs

2.4 Notre algorithme principal

L’assemblage d’images se déroule suivant plusieurs étapes bien distinctes.

Dans un premier temps, on charge les images en paramètre dans un tableau d’images.

```
1 tableauImagesCouleur = creationTableauImageCouleur(input ,  
          nombreInput , bool_erreur);
```

Ensuite, on transforme ces images couleur en noir et blanc puis on les découpe en quatre. On laisse apparaître les points clés dans le tier gauche de l’image, le tier droit, le tier haut et le tier bas. Le but est de limiter la comparaison du haut de l’image 1 et du bas de l’image 2 ou de la droite de l’image 1 et de la droite de l’image 2 et inversement pour limiter le temps du programme tout en conservant l’efficacité de l’algorithme de comparaison.

```
1 tableauImagesTemporaire = creationTableauImageTemporaire(  
          tableauImagesCouleur , nombreInput , bool_erreur);  
tabCoupes = creerTableauCoupe(tableauImagesTemporaire ,  
          nombreInput , bool_erreur);
```

Puis on récupère un tableau de décalages qui constituent tous les décalages possibles entre les images. On compare toutes les coupes entre-elles. On récupère aussi la fréquence des décalages s'ils sont trouvés plusieurs fois. Un décalage est stocké par image coupée. Pour l'utiliser, il faut créer auparavant un tableau de la taille du nombre de décalages qui indique s'il faut calculer un décalage ou pas. Cela permet de ne pas recalculer un décalage si cela a déjà été fait. Par exemple, si on calcule un décalage entre la partie droite de l'image 1 avec la partie gauche de l'image 2 sachant qu'on calcule tous les décalage à partir de l'image 1, alors lorsqu'on calcule les décalages de l'image 2 par rapport aux autres images, on ne recalculera pas le décalage entre l'image 2 et l'image 1.

```

1 decalageAPasCalculer = genererTableauDecalageAPasCalculer(
    nombreInput);
2 decalages = calculerTousLesDecalage(tabCoupes,
    decalageAPasCalculer, nombreInput);

```

Une fois ce travail effectué, on peut libérer le tableau d'images en noir et blanc.

```
libererTableauImages2(tableauImagesTemporaire, nombreInput);
```

Enfin, on colle les images en fonction des décalages les plus fréquents et on récupère l'image la plus grande.

```

1 imageFin = collerToutesLesImages(decalages, tableauImagesCouleur,
    nombreInput);

```

Dans la fonction principale, on récupère cette image et on la sauvegarde dans le fichier demandé.

```

1 panorama = traitementPanorama(tableauImagesCouleur, nombreInput,
    bool_erreur);
save(panorama, output, bool_erreur);

```

2.5 Comment améliorer cet algorithme ?

La première chose à améliorer est l'algorithme pour coller toutes les images. Il n'est pas tout à fait au point et ne prend pas en compte tout les cas possibles.

Une autre amélioration serait de réduire au maximum la sélection des points clés pour en avoir moins et gagner du temps lors de la comparaison.

De plus il serait intéressant d'accélérer le temps de traitement des images.

Et enfin il faudrait prendre en compte plus de cas, notre programme prend en charge uniquement les photos dont la correspondance se trouve sur le tiers extérieur de l'image.

Conclusion

Pour conclure, on peut dire que ce projet était très compliqué à gérer, notamment parce que le temps imparti était très important (un semestre) mais aussi parce que le traitement d'image est plus compliqué que ce que l'on peut penser à première vue. De plus c'est la première fois, que la classe est laissée en autonomie tout au long du projet, il a donc fallu apprendre un nouveau langage, tout en avançant notre projet.

Il a aussi fallu que tous les membres du groupe apprenne à se connaître pour que le fonctionnement du groupe et l'efficacité du travail soit optimal.

Après de nombreuses recherches, de nombreux tests et des tentatives plus ou moins fructueuse, nous avons réussi à remplir les objectifs du projet pas à pas et à rendre un travail fonctionnel même s'il reste des améliorations à apporter.

Bibliographie

- [1] Paul BOURKE. *Lens Correction and Distortion*. 2002. URL : <http://www.paulbourke.net/miscellaneous/lenscorrection/>.
- [2] OPERATIONPIXEL@FREE.FR. *Traitement d'images*. 2010. URL : http://operationpixel.free.fr/pointinteret_fast.php.
- [3] David VERNON. *Machine Vision*. 1991. Chap. 4. URL : <http://homepages.inf.ed.ac.uk/rbf/BOOKS/VERNON/Chap004.pdf>.
- [4] WIKIPÉDIA. *Kernel (Image processing)*. URL : [http://en.wikipedia.org/wiki/Kernel_\(image_processing\)](http://en.wikipedia.org/wiki/Kernel_(image_processing)).