

Méthodes rapides pour la recherche des plus proches
voisins SIFT : application à la recherche d'images
et
Contributions à la reconstruction 3D multi-vues

Adrien Auclair

8 septembre 2009

Résumé

Cette thèse adresse deux problèmes en vision par ordinateur. Dans une première partie, nous nous intéressons à l'indexation et à la recherche d'images par le contenu. Et dans une seconde partie, nous étudions la reconstruction 3D à partir d'images.

L'application motivant la première partie est celle de pouvoir identifier rapidement, dans une grande base, quelles images sont similaires à une image requête. Pour cela, nous utilisons dans nos travaux les méthodes utilisant des descripteurs locaux, notamment les descripteurs dit SIFT (Scale Invariant Features Transform). Parmi les étapes de ces algorithmes, nous avons concentrés nos travaux sur l'étape de recherche de correspondances, qui est un problème de recherche des plus proches voisins (k -NN ou r -NN). Nous étudions dans un premier temps les performances d'une recherche linéaire exacte ainsi qu'une implémentation de cet algorithme sur une architecture GPU. Ensuite, après un état de l'art des algorithmes r -NN approchés, nous introduisons un nouvel algorithme de hachage de points que nous nommons HASHDIM, basé sur une description ordinale des points étudiés. Cet algorithme offre des performances comparables aux hachages les plus performants, dits LSH (Locality Sensitive Hashing). Mais l'avantage majeur de HASHDIM est sa faible consommation mémoire. En utilisant le formalisme dit Bag-Of-Words, nous utilisons HASHDIM pour effectuer des recherches d'images similaires dans des bases contenant jusqu'à 510.000 images. Sur un benchmark d'images déformées, HASHDIM obtient des performances identiques à celles des meilleures méthodes de l'état de l'art mais son point fort est de ne nécessiter aucun apprentissage.

Dans la seconde partie de ces travaux, la motivation industrielle est d'obtenir une représentation 3D de véhicules filmés par une caméra (par exemple pour un péage). La difficulté tient au fort aspect réfléchissant des surfaces étudiées. Nous proposons une méthode en plusieurs étapes pour être robuste à cette particularité. En faisant l'hypothèse de véhicules en translation pure, nous déterminons le point de fuite du mouvement pour rendre plus efficace l'étape de suivi de points et enfin nous utilisons un algorithme de Structure-from-Motion pour générer un nuage de points 3D approchant la surface de l'objet. Cette approche permet d'obtenir une représentation simple des véhicules étudiés, mais ayant l'avantage d'être robuste aux reflets à la surface des voitures. Pour obtenir des objets 3D plus précis à partir d'images, nous étudions ensuite le problème plus générique de la reconstruction d'objets 3D mates à partir d'images. Après un état de l'art de ce problème classique, nous rapprochons nos travaux de l'approche de Pons & al. qui mesurent l'erreur d'une surface 3D temporaire en estimant la différence entre les projections de cette surface et les images d'entrées de l'algorithme. Nous avons proposé de modifier leur approche en n'utilisant non plus une erreur dense, mais des erreurs calculées comme correspondances SIFT entre les images de l'objet temporaire et les images d'entrée de l'algorithme. Sur le benchmark public de Seitz & al., notre algorithme obtient des résultats qui offrent un très bon compromis entre qualité de la surface obtenue et temps de calcul.

Remerciements

Mes premiers remerciements vont à mes directeurs de thèse, Nicole Vincent et Laurent Cohen, pour leurs conseils, leur suivi et la liberté qu'ils m'ont laissé dans l'orientation de mes travaux de thèse. Je remercie tout particulièrement Nicole de m'avoir accueilli au sein de l'équipe SIP du CRIP5. Merci pour toutes ces relectures d'articles, répétitions de présentation, relectures de versions du manuscrit...

Je remercie aussi mes rapporteurs Matthieu Cord et Jean Ponce pour l'intérêt qu'ils ont porté à mes travaux, leurs remarques pertinentes, malgré la lecture estivale que je leur ai imposé. Merci également aux autres membres du jury, Radu Horaud et Frederic Jurie d'avoir acceptés de juger mes travaux.

Pour remonter un peu plus loin, merci à Dominique Attali, du laboratoire LIS à Grenoble pour m'avoir montré les joies de la recherche et comment on pouvait se passionner pour chaque problème. Merci aussi à Olivier Jamet du LAREG pour avoir encadré mon stage de Master.

Un mot aussi pour remercier Philippe Mouttou qui a su réunir les conditions pour que je puisse démarrer ma thèse. Merci à Nicolas Verbeke, avec qui j'ai partagé les creux de vague du milieu de thèse.

Une pensée particulière pour les autres doctorants du SIP, pour leur bonne humeur quotidienne. Merci à Nicolas Lauménie pour son dynamisme communicatif, à Florence, à Georges...

Enfin, un dernier mot pour remercier Kanitha, pour son soutien de tous les jours et pour m'avoir encouragé dans mes choix.

Table des matières

1	Introduction	2
1.1	Vision par ordinateur	3
1.2	Partie 1 : Méthodes rapides pour la recherche des plus proches voisins : application à la recherche d'images	4
1.3	Partie 2 : Contributions à la reconstruction 3D à partir d'images	10
I	Méthodes rapides pour la recherche des plus proches voisins SIFT : application à la recherche d'images similaires	14
2	Algorithmes de recherche d'images similaires par descripteurs locaux et recherche linéaire des plus proches voisins	15
2.1	Problématique de la recherche de descripteurs locaux	17
2.2	Le descripteur local SIFT	19
2.3	Analyse de données sur des descripteurs SIFT	29
2.4	Algorithmes utilisés pour les applications visées	31
2.5	Choix de la taille des voisinages recherchés	39
2.6	Recherche linéaire	46
2.7	Protocole d'évaluation pour la recherche d'images	59
2.8	Modification de l'étape de sélection des régions d'intérêt	61
2.9	Résultats avec la recherche linéaire	65
2.10	Conclusion	66
3	Etat de l'art de la recherche des plus proches voisins	68
3.1	Introduction	69
3.2	Méthodes exactes à base d'arbres	69
3.3	Autres méthodes exactes	73
3.4	Méthodes approchées à base d'arbres	73
3.5	Méthodes approchées à base de hachage	74
3.6	Autres méthodes approchées	80
3.7	Diminution de la dimension de l'espace	82
3.8	Taille des bases de données dans la littérature	83
3.9	Méthodes d'évaluation de la recherche r-NN approchée	83
3.10	Conclusion	86

4	Méthode de hachage proposée	87
4.1	Paramétrage du hachage LSH1	88
4.2	LSH-Opti : une modification de l'algorithme LSH1	88
4.3	HASHDIM : un algorithme r-NN approché utilisant des fonctions de hachage basées sur un ordre des dimensions	94
4.4	Evaluation	100
4.5	Recherche d'images similaires	104
4.6	Tests sur une grande base	106
4.7	Résultats pour l'application d'analyse de linéaires	110
4.8	Conclusion	116
4.9	Appendice	118
5	Utilisation du hachage HASHDIM dans une recherche Bag-Of-Features	122
5.1	Approche Bag-Of-Features	124
5.2	Lien entre vocabulaire visuel et algorithme de recherche des plus proches voisins	125
5.3	Utilisation de notre hachage HASHDIM dans une recherche BOF	126
5.4	Taille du vocabulaire	128
5.5	Choix de la fonction de coût	128
5.6	Impact des paramètres (n, k)	130
5.7	Comparaison avec des vocabulaires K-means	131
5.8	Test sur la base de 510.000 images	134
5.9	Tests sur la base Nister	136
5.10	Conclusion	139
6	Conclusion et Perspectives	140
II	Contributions à la reconstruction 3D multi-vues	144
7	Reconstruction 3D : Introduction	145
7.1	Reconstruction 3D multi-vues	146
7.2	Histoire de la formation des images	146
7.3	Histoire de la reconstruction 3D	147
7.4	Problématique	149
8	Etat de l'art de la reconstruction 3D	151
8.1	Systèmes actifs	152
8.2	Introduction aux systèmes passifs	154
8.3	Méthodes par approximation de nuages de points	162
8.4	Méthodes par analyse de l'espace 3D	164
8.5	Méthodes de reconstruction cohérente avec les images	166
8.6	Autres approches par critère de photo-cohérence local	176
8.7	Conclusion	177

9	Méthode proposée pour la reconstruction d’une voiture	179
9.1	Motivations	180
9.2	Analyse du problème	180
9.3	Algorithme de “Structure From Motion” pour une translation	181
9.4	Reconstruction d’une surface à partir du nuage de points 3D	188
9.5	Evaluation	192
9.6	Perspectives	193
9.7	Conclusion	202
10	Reconstruction 3D par déformation de surface guidée par des correspondances SIFT	203
10.1	Notations	205
10.2	Mesure de photo-cohérence utilisée	206
10.3	Modèle déformable utilisé	208
10.4	Force calculée à partir de correspondances SIFT	210
10.5	Ajout d’un terme plus local	212
10.6	Détails concernant l’implémentation	213
10.7	Résultats	215
10.8	Analyse qualitative	215
10.9	Conclusion	218
11	Conclusion et Perspectives	222
	Bibliography	225

Chapitre 1

Introduction

Sommaire

1.1	Vision par ordinateur	3
1.2	Partie 1 : Méthodes rapides pour la recherche des plus proches voisins : application à la recherche d'images	4
1.3	Partie 2 : Contributions à la reconstruction 3D à partir d'images	10

1.1 Vision par ordinateur

Cette thèse s'inscrit dans le domaine de la vision par ordinateur. Ce domaine regroupe les systèmes informatiques qui extraient de l'information à partir d'images. Ces images peuvent prendre plusieurs formes : une image unique, plusieurs images ou encore une vidéo. Elles peuvent par exemple provenir d'un simple appareil photo, d'une caméra, d'un système d'imagerie par résonance magnétique.

L'information recherchée peut être très variable en fonction du problème posé. On peut chercher des intrusions dans un site à partir d'une caméra de vidéosurveillance, on peut vouloir automatiser la lecture d'un document manuscrit, reconnaître une empreinte digitale, détecter une anomalie sur une image médicale... Cette liste est évidemment loin d'être exhaustive.

La finalité de la vision par ordinateur est souvent d'imiter les tâches effectuées par la vision humaine. En terme de qualité, notre système de vision couplé à notre cerveau est un outil exceptionnel. Nous sommes tellement habitués à ce système qu'il est rare de le réaliser. Alors que la donnée en entrée ne consiste qu'en un signal lumineux reçu par nos yeux, nous sommes capables d'avoir une représentation tridimensionnelle de la scène, de catégoriser les objets, etc. Par exemple, ce système est capable de passer d'un signal lumineux à une interprétation du type : "je vois une voiture de sport, située à une centaine de mètres, qui roule doucement sur une route de campagne".

Pour un ordinateur, il est impossible d'arriver à de tels résultats, avec une telle généralité. Toutefois, avec des objectifs moins ambitieux, on peut aboutir à des résultats très utiles. On peut par exemple détecter les voitures sur une image. On peut essayer de catégoriser des images de véhicules entre plusieurs classes : voitures/camion. On peut, à partir de plusieurs images, étudier les changements d'intensité des pixels pour détecter un objet en mouvement, calculer des informations sur la géométrie de la scène, et en déduire la vitesse d'un objet. Enfin, on peut tenter de segmenter une image en plusieurs régions et assigner une classe à chaque région : prairie/route/ciel. On est bien loin du résultat de l'interprétation humaine, mais cela permet toutefois d'envisager certaines applications, avec les avantages de l'informatique : automatisation, gestion de grandes masses d'informations.

Dans les paragraphes suivants, nous introduisons les deux parties de nos travaux. La première concerne les méthodes rapides de recherche des plus proches voisins, pour des descripteurs locaux d'images. La seconde s'intéresse à la reconstruction 3D à partir d'images.

1.2 Partie 1 : Méthodes rapides pour la recherche des plus proches voisins : application à la recherche d'images

1.2.1 Motivations

Nos recherches sur ce premier thème sont motivées par deux applications industrielles qui font toutes deux appel à une recherche de similarités entre images. Toutefois, avant de présenter ces deux applications, nous définissons une hiérarchie de difficulté dans la recherche d'images similaires afin de situer nos travaux.

L'exercice le plus simple de recherche d'images est celui que nous appellerons la recherche d'images similaires (illustré sur la figure 1.1). Par ce terme, nous signifions qu'un être humain va percevoir ces images comme identiques. C'est à dire que la différence entre des images similaires peut venir d'un ajout de bruit, d'effets d'algorithmes de compression, de légers changements globaux des intensités des pixels... Dans notre définition d'images similaires, nous rajoutons des images d'un objet qui peuvent avoir été prises selon un point de vue très proche. La difficulté supérieure concerne la recherche d'images de même scène (illustré sur la figure 1.2). Deux images sont dites de même scène si elles sont une représentation d'une unique scène 3D (ou d'un même objet). Les points de vue entre les images peuvent être complètement différents, ou il peut y avoir des occlusions partielles entre les images. A noter que les transformations des images similaires sont incluses dans celles des images de même scène. Enfin, le dernier niveau de difficulté est celui de la recherche d'images de même catégorie (illustré sur la figure 1.3). Deux images sont dites de la même catégorie si elles représentent le même concept. C'est à dire qu'il s'agit de classifier les images. Les classes peuvent par exemple être : vélo/voiture/rue/immeuble... Dans nos travaux, nous nous sommes principalement intéressés à la recherche d'images similaires. Toutefois, à des fins de comparaison, nous avons aussi appliqué nos algorithmes à des jeux de tests concernant la recherche d'images de même scène, mais nous n'avons pas étudié le problème de la catégorisation d'images.

Application 1 : Recherche d'images similaires

Dans cette première application, on dispose d'une base d'images et on veut savoir si une image requête appartient à cette base. On souhaite retrouver l'image même si elle a subi des déformations du type changement de taille, rotation, étirement, changements d'illumination, recadrage... Il s'agit donc d'une application appartenant à la catégorie recherche d'images similaires. Le tableau 1.4 montre une image et des transformations de cette image que nous avons utilisées dans nos tests. Si l'on utilise l'une des déformations comme image requête, l'application doit retrouver dans une large base l'image originale. Un tel système présente un intérêt par exemple pour vérifier les droits d'auteurs dans des bases d'images.

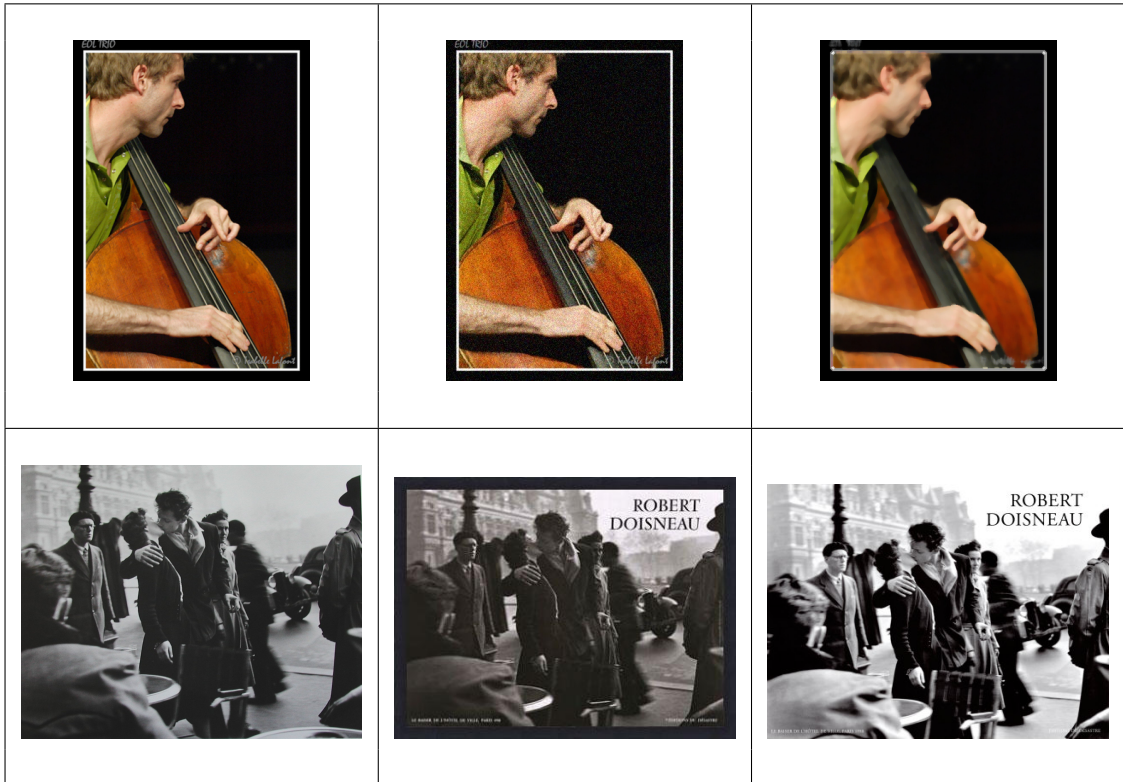


TABLE 1.1 – Chaque ligne illustre un exemple d’images similaires.

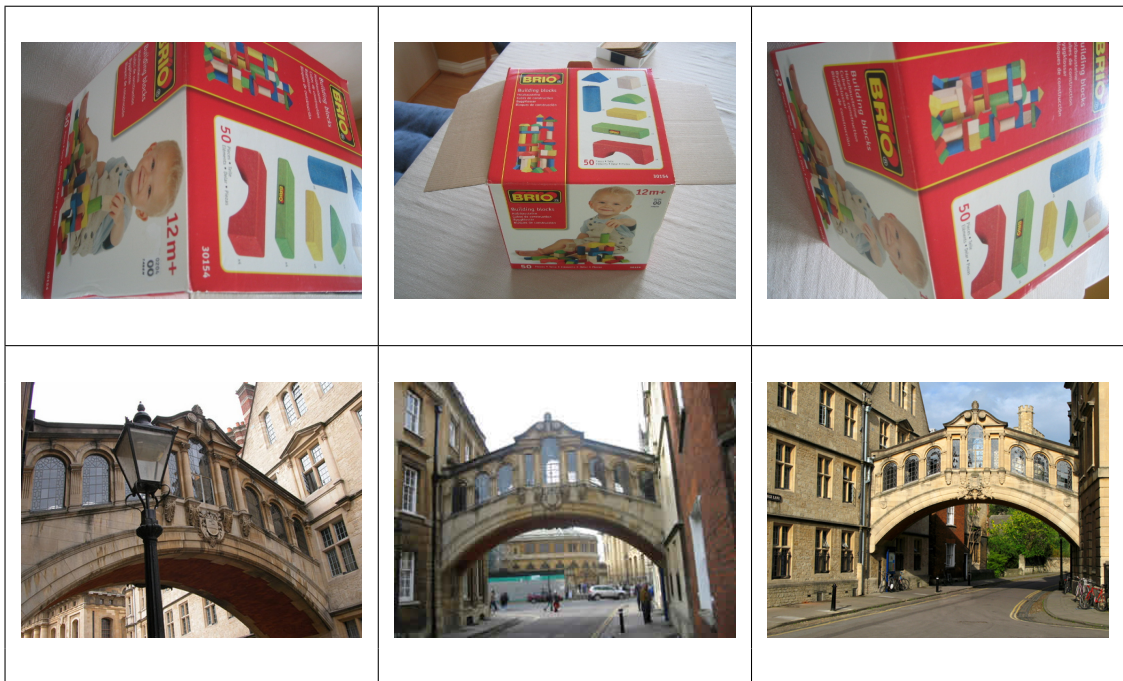


TABLE 1.2 – Chaque ligne illustre un exemple d’images de même scène.


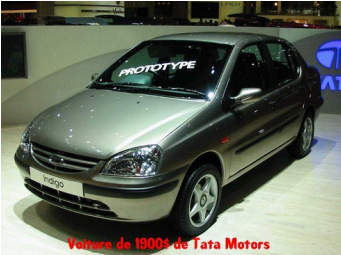




		
		

TABLE 1.3 – Chaque ligne illustre un exemple d’images appartenant à la même catégorie.







 <p data-bbox="459 898 665 931">Image originale</p>	 <p data-bbox="810 898 1179 931">Diminution du contraste x3</p>
 <p data-bbox="363 1223 761 1256">Augmentation du contraste x3</p>	 <p data-bbox="786 1223 1195 1256">Suppression de 90 % de l'image</p>
 <p data-bbox="405 1547 719 1581">Transformation Emboss</p>	 <p data-bbox="802 1547 1182 1581">Application filtre median 3x3</p>

TABLE 1.4 – Exemples des transformations d'images utilisées.



FIGURE 1.1 – Exemple de photo d'un linéaire de supermarché.



FIGURE 1.2 – Quelques images d'une base de produits de supermarchés.

Application 2 : Analyse automatique de linéaires de supermarché

Pour cette seconde application, on dispose d'une photo d'un linéaire de supermarché (figure 1.1) ainsi que d'une base d'images de produits (figure 1.2). On souhaite retrouver les positions des produits dans l'image initiale (un exemple de résultat est montré sur la figure 1.3). A partir des positions des produits sur l'image, on peut en déduire le planogramme du linéaire, c'est à dire une carte des produits sur le linéaire. Ce planogramme est un outil très utilisé dans le domaine du merchandising.

1.2.2 Problématique

Dans les deux applications présentées, pour atteindre le but de détecter des images similaires (et non des images de même scène ou de même catégorie), la difficulté centrale est de détecter des similarités entre des portions d'images. Ce problème peut se diviser en deux étapes. Premièrement, des zones d'intérêt sont détectées dans chaque image et un descripteur est calculé pour chacune de



FIGURE 1.3 – Exemple de résultat d’une analyse d’une photo de linéaire. Les images de la base de produits détectés sur la photo à analyser sont copiés à l’emplacement où ils sont trouvés. Sur cette image, on a aussi tracé un rectangle autour des zones de même produit.

ces zones. Deuxièmement, on cherche des descripteurs proches entre ceux d’une image requête et ceux de la base d’images.

Concernant la première étape, l’algorithme SIFT de D. Lowe [Low04] est très efficace pour cette tâche. Il combine un détecteur de régions d’intérêt ainsi qu’un algorithme pour décrire ces régions. Dans le comparatif de Mikolajczyk et al. [MS05], il obtient les meilleurs résultats. Nous avons donc fait le choix d’utiliser ce descripteur dans nos travaux.

Concernant la seconde étape, la recherche rapide de voisins est difficile car ces descripteurs SIFT sont des points dans un espace à 128 dimensions et les nuages de points peuvent être très grands (chaque image générant en moyenne 1000 points).

Dans nos travaux, nous nous concentrons principalement sur cette seconde étape. Les méthodes étudiées font partie de la classe d’algorithmes appelée “recherche des plus proches voisins dans un espace de grande dimension”.

D’autre part, nous nous intéressons également à la recherche d’images similaires (et pour quelques tests, à la recherche d’images de même scène) par la méthode dite de Bag-Of-Features introduite dans Sivic et al. [SZ03]. Cette partie de nos travaux intègre parfaitement le point précédent car comme l’ont montré Jegou et al. dans [JDS08], il est possible d’utiliser n’importe quel algorithme de plus proches voisins comme étape centrale de cette méthode de recherche d’images.

1.2.3 Plan

Dans le chapitre 2, nous introduisons les algorithmes de haut niveau que nous utilisons pour les applications visées. Nous y détaillons aussi la méthode linéaire pour la recherche des plus proches voisins. Nous présentons aussi des variantes de cet algorithme ainsi qu’une implémentation sur processeur gra-

phique (GPU). Ce chapitre permet de mettre en place toutes les briques de la recherche d'images similaires par descripteurs locaux. Dans un second chapitre, dans le but d'accélérer fortement la recherche d'images similaires, nous faisons un état de l'art des méthodes approchées de recherche des plus proches voisins. Ensuite, dans le chapitre 4, nous introduisons un nouvel algorithme de recherche des plus proches voisins basé sur des fonctions de hachage originales (nous nommons cet algorithme HASHDIM) et évaluons ses performances. Enfin, dans le dernier chapitre de cette partie, nous intégrons HASHDIM dans une recherche d'images par Bag-Of-Features et comparons les résultats obtenus à ceux d'autres algorithmes.

1.3 Partie 2 : Contributions à la reconstruction 3D à partir d'images

Dans une seconde partie de nos travaux, nous souhaitons obtenir une représentation de la géométrie 3D d'un objet à partir de plusieurs images de celui-ci. Cette étape appelée "reconstruction 3D" est très importante en vision par ordinateur. En effet, connaître la géométrie 3D des objets d'une scène peut permettre des interprétations qui sont impossibles en utilisant uniquement des informations 2D. On peut aussi utiliser la représentation 3D obtenue pour faire du rendu graphique. Par exemple, une représentation 3D d'un présentateur de télévision permet ensuite de le faire apparaître dans une scène virtuelle. La reconstruction 3D à partir d'images peut aussi être utile pour la conservation du patrimoine. La figure 1.4 montre des images d'un temple et la figure 1.5 montre un rendu de la surface 3D que l'on souhaite obtenir.

La continuité de ces travaux avec ceux de la première partie vient du fait que l'étape sous-jacente à la reconstruction 3D est toujours de trouver des similarités, cette fois locales, entre plusieurs images (on parle alors de correspondances). En stéréo-vision, on cherchera par exemple des similarités entre deux images. A partir de ces correspondances, on peut déduire une information de profondeur de la scène par rapport au système de vision. Si l'on utilise une séquence vidéo, on cherchera plutôt des similarités entre les images successives.

1.3.1 Motivations

L'application qui a motivé ces travaux est la reconstruction 3D de voitures à partir de séquences vidéos. L'intérêt est de connaître le niveau de qualité que l'on peut obtenir et quelles interprétations sont possibles à partir des informations 3D. Est-ce utile pour distinguer le modèle du véhicule ? Peut-on s'en servir pour estimer le gabarit à un péage autoroutier ?

La figure 1.6 montre des images extraites d'une séquence vidéo filmée à partir d'une caméra statique au bord d'une route. La figure 1.7 montre une image extraite d'une autre séquence vidéo ainsi qu'une reconstruction 3D.



FIGURE 1.4 – Images d'un objet que l'on souhaite reconstruire en 3D.

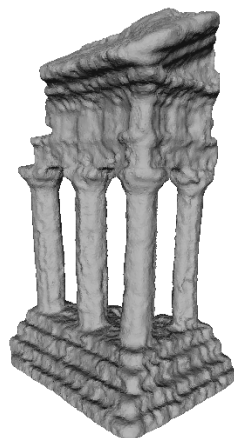


FIGURE 1.5 – Exemple d'une reconstruction 3D obtenue.



FIGURE 1.6 – Quelques images d’un film qui sert de donnée d’entrée à notre algorithme.



FIGURE 1.7 – (a) Une image extraite d’une séquence vidéo et (b) la reconstruction 3D obtenue par extrusion de polynômes.

1.3.2 Problématique

Au cours de la décennie passée, des méthodes efficaces de reconstruction 3D ont été proposées. Le livre “Multiple View Geometry in Computer Vision” [HZ04] est un ouvrage classique qui détaille ces méthodes. Toutefois, une des difficultés du sujet est la complexité des surfaces des voitures. On peut par exemple trouver des surfaces transparentes, des reflets, de larges zones peu texturées. Les méthodes classiques sont alors plus délicates à mettre en oeuvre.

Dans nos travaux, nous cherchons la façon d’utiliser ces algorithmes classiques pour résoudre le problème de la reconstruction 3D de voitures. Nous étudierons la possibilité d’utiliser des connaissances sur l’objet reconstruit (i.e. un véhicule) pour rendre robustes les étapes de reconstruction.

Nous étudions aussi si les descripteurs SIFT peuvent permettre d’améliorer les résultats d’une reconstruction 3D, sans se restreindre au cas des voitures. Pour cette seconde partie, nous comparons les résultats obtenus à ceux du benchmark de Seitz et al. [SCD⁺06].

1.3.3 Plan

Dans un premier chapitre, nous présentons un état de l’art de la reconstruction 3D. Ensuite, dans le chapitre 9, nous proposons une méthode pour utiliser des algorithmes classiques pour reconstruire des voitures en 3D. Cette méthode exploite des connaissances sur l’objet reconstruit ainsi que sur son mouvement. Enfin, dans un dernier chapitre, nous développons une méthode originale de reconstruction 3D qui utilise les descripteurs SIFT afin d’obtenir

une reconstruction 3D précise d'un objet à partir d'images.

Première partie

Méthodes rapides pour la recherche des plus proches voisins SIFT : application à la recherche d'images similaires

Chapitre 2

Algorithmes de recherche d'images similaires par descripteurs locaux et recherche linéaire des plus proches voisins

Dans ce chapitre, nous présentons d'abord le descripteur local SIFT utilisé dans nos travaux. Ensuite, nous détaillons les algorithmes que nous avons mis en place pour rechercher des images similaires dans une base d'images et pour construire un planogramme à partir d'une image de supermarché. Dans ces deux algorithmes, l'étape centrale est la recherche de descripteurs locaux proches. Ensuite, nous étudions les performances de l'algorithme le plus simple pour cette étape : la recherche linéaire des plus proches voisins. Après avoir détaillé un protocole de test pour la recherche d'images similaires, nous proposons une méthode pour diminuer le nombre de descripteurs SIFT par image. Enfin, pour servir de références pour les chapitres suivants, nous mesurons les performances de la recherche d'images similaires en utilisant un algorithme linéaire pour la recherche des plus proches voisins.

Dans cette partie, notre première contribution est une méthode pour sélectionner les descripteurs parmi ceux retournés par l'algorithme SIFT classique (partie 2.8). Nous introduisons aussi un algorithme pour obtenir un planogramme à partir d'une image de linéaire de supermarché en partie 2.4.2 (Auclair et al. [ACV07a]). La troisième contribution est l'utilisation de distances partielles pour la recherche linéaire de plus proches voisins SIFT (partie 2.6.3). Enfin, une contribution est la comparaison des performances de la recherche linéaire des plus proches voisins sur CPU et GPU (processeur de la carte graphique), en partie 2.6.4.

Sommaire

2.1	Problématique de la recherche de descripteurs locaux	17
2.2	Le descripteur local SIFT	19

2.3	Analyse de données sur des descripteurs SIFT . . .	29
2.4	Algorithmes utilisés pour les applications visées . .	31
2.5	Choix de la taille des voisinages recherchés	39
2.6	Recherche linéaire	46
2.7	Protocole d'évaluation pour la recherche d'images	59
2.8	Modification de l'étape de sélection des régions d'intérêt	61
2.9	Résultats avec la recherche linéaire	65
2.10	Conclusion	66

2.1 Problématique de la recherche de descripteurs locaux

La problématique de cette partie est liée à la recherche de similarités entre images. Pour trouver des similarités, il n'est pas efficace de comparer directement les valeurs des pixels entre des images. Cette méthode ne serait par exemple pas robuste à des déformations comme celles liées à des changements d'illumination, à l'échelle, aux rotations, aux changements de points de vue. Il est toutefois possible d'essayer de trouver des corrélations entre des fenêtres de pixels. Les méthodes récentes, plus efficaces, consistent à décrire chaque image avec un ou plusieurs descripteurs. L'objectif est alors de construire un descripteur à partir des valeurs des pixels, qui sera invariant selon les critères souhaités (rotation, échelle...). Ces descripteurs sont alors des points dans un espace de grande dimension. Pour trouver des similarités entre images, on se limite à trouver des descripteurs proches selon une distance choisie.

2.1.1 Plusieurs algorithmes de recherche d'images mais un même besoin pour des algorithmes de plus proches voisins efficaces

Quel que soit le type d'image et le type de descripteur utilisé, l'algorithme de recherche des plus proches voisins est un composant central. Ceci est vrai pour différents algorithmes de recherche d'images similaires. Afin d'illustrer ce propos, nous donnons ici les grandes lignes de trois algorithmes (les deux derniers sont détaillés dans les parties suivantes) :

- Recherche par histogramme couleur

Chaque image de la base est décrite par un histogramme couleur. Pour retrouver les images similaires à une image requête, son histogramme couleur est calculé et on cherche dans la base les histogrammes proches selon une certaine distance. Cette dernière étape nécessite un algorithme de recherche des plus proches voisins.

- Méthode de D. Lowe dans [\[Low04\]](#)

Les étapes de cet algorithme sont :

1. Calcul des descripteurs locaux SIFT des images de la base.
2. Calcul des descripteurs locaux SIFT de l'image requête.
3. Pour chaque descripteur de l'image requête, recherche de ses voisins dans les descripteurs de la base d'images.
4. Analyse des correspondances de points trouvées pour décider si l'image requête est similaire à des images de la base. Cette analyse est faite par recherche d'une transformation affine entre l'image requête et les images de la base (la transformation affine est cherchée à partir des correspondances de points trouvées).

Dans cet algorithme, la recherche des plus proches voisins est utilisée à l'étape 3. Par la suite, on appellera cet algorithme SIFT+NN+AFF (pour "descripteurs SIFT + Nearest Neighbors + vérification Affine").

- Méthode Bag-Of-Features de Sivic et al. [SZ03] Pour chaque image de la base, ses descripteurs locaux sont calculés. Pour chaque descripteur, un “mot” d’un vocabulaire lui est assigné. Cette assignation est faite par un algorithme de recherche des plus proches voisins. Puis chaque image est décrite par un vecteur de fréquence construit avec ses mots. La recherche d’images similaires revient ensuite à une recherche de vecteurs de fréquences proches. Cette explication succincte (l’algorithme est beaucoup plus détaillé dans le chapitre 5) est uniquement présentée ici pour montrer le besoin d’un algorithme de plus proches voisins.

Ces trois exemples illustrent bien l’importance de l’étape de recherche des plus proches voisins dans la recherche d’images similaires. Dans nos travaux, nous nous sommes restreints aux algorithmes de plus proches voisins pour descripteurs SIFT. Cette restriction nous a conduit à concentrer nos travaux sur la recherche d’images similaires utilisant uniquement les algorithmes SIFT+NN+AFF et Bag-Of-Features. Toutefois, ces deux algorithmes sont très performants et permettent de couvrir une vaste gamme d’applications. Ils s’appliquent tous deux à la recherche d’images similaires (notre première application présentée en 1.2.1). Pour l’application d’analyse d’images de linéaires de supermarché (voir 1.2.1), seul l’algorithme SIFT+NN+AFF est applicable (car nous ne cherchons pas une seule mais plusieurs transformations entre l’image de linéaire et les images des produits).

Dans ce chapitre et le suivant, nous avons utilisé cette recherche des plus proches voisins en tant qu’étape de l’algorithme SIFT+NN+AFF. C’est la première méthode de recherche d’images que nous avons testée. Ensuite, dans le chapitre 5, nous utilisons différents algorithmes de recherche des plus proches voisins dans une approche Bag-Of-Features.

2.1.2 Les difficultés du problème

Les deux grandes caractéristiques du problème de la recherche des plus proches voisins pour les descripteurs locaux SIFT sont la taille des nuages de points et le nombre de dimensions de l’espace dans lequel sont ces descripteurs. Du fait de ces deux caractéristiques, le temps nécessaire pour trouver les plus proches voisins d’un point peut être un frein pour certaines applications.

Imaginons un moteur de recherche d’images similaires qui travaille sur la base d’images du site Flickr. A l’heure à laquelle ces lignes sont écrites, la base contient un peu plus de deux milliards d’images. Si l’on considère qu’en moyenne, chaque image est décrite avec 1000 descripteurs, la quantité de points à gérer devient gigantesque ($2 \cdot 10^{12}$ points pour la base Flickr).

La seconde caractéristique de ce problème est le nombre de dimensions des espaces étudiés. Dans nos travaux, nous nous sommes concentrés sur les descripteurs locaux SIFT qui génèrent des points dans un espace à 128 dimensions. Les difficultés liées au grand nombre de dimensions de l’espace étudié sont classiquement regroupées sous le terme “malédiction de la dimension”.

Malédiction de la dimension Dans le domaine de la vision par ordinateur, les espaces utilisés sont souvent le plan ou un espace à 3 dimensions. Il

s'avère que les algorithmes d'indexation utilisés pour ces espaces deviennent inefficaces lorsque le nombre de dimensions augmente (e.g. à partir d'une dimension égale à 10). Cette perte d'efficacité des algorithmes d'indexation, liée à la grande dimension des espaces est une composante de cette "malédiction de la dimension".

Pour mieux appréhender les difficultés possibles liées à des espaces de grande dimension, nous en exposons ici quelques particularités (d'autres exemples sont illustrés dans [BBK01] ou [Kop00]) :

- Pour se représenter des propriétés dans un espace de grande dimension, nous avons souvent tendance à imaginer un espace 3D et à extrapoler les conclusions obtenues. Toutefois, cette approche peut être trompeuse. Par exemple, considérons un hypercube dans un espace à d dimensions : $[0, 1]^d$. On définit le centre du cube : $c = (0.5, \dots, 0.5)$. Le lemme "toute sphère qui touche ou intersecte toutes les frontières (de dimension $d - 1$) du cube de dimension d contient le centre c " est vrai en dimension 2 et 3. Toutefois, il ne l'est plus en dimension 16. On montre cela par un contre exemple. Considérons le point $p = (0.3, \dots, 0.3)$. La distance euclidienne entre ce point et le centre est de $\sqrt{16 \times 0.2^2} = 0.8$. On définit la sphère S centrée en p , de rayon 0.7. Cette sphère touche ou intersecte tous les bords du cube (surface de dimension 15) mais ne contient pas le point central, et donc ne vérifie pas le lemme.
- Dans [Kop00], l'auteur trace le rapport entre le volume d'une hypersphère de rayon 1 et le volume d'un hypercube de côté 2 pour des dimensions d'espace variables. On voit qu'à partir de 10 dimensions, le volume de l'hypersphère est négligeable par rapport au volume de l'hypercube. Cela a un impact direct sur la conception de structures de recherche pour des espaces de grande dimension.

Ces exemples montrent que le nombre de dimensions est un facteur essentiel du problème. Un algorithme de recherche des plus proches voisins performant, pour un espace à trois dimensions, ne le sera pas forcément pour un espace à 128 dimensions comme celui des SIFT.

Avant d'aborder en détail les algorithmes de recherche d'images similaires et de plus proches voisins, il nous a paru nécessaire de présenter l'algorithme SIFT et de justifier ce choix par rapport à d'autres descripteurs.

2.2 Le descripteur local SIFT

Les descripteurs locaux sont le résultat de deux étapes. Dans un premier temps, des points d'intérêt dans l'image sont détectés. Ensuite, pour chaque point d'intérêt, un descripteur qui décrit localement l'image autour du point est construit. Au final, on cherche à décrire une image avec des descriptions de certaines zones locales de cette image. L'objectif est de retrouver les mêmes zones décrites de façon similaire même si l'image subit des déformations.

La qualité d'un détecteur de points d'intérêt se mesure par sa répétabilité. C'est-à-dire que l'on doit idéalement retrouver les mêmes points d'intérêt après que l'image a subi une modification. Un descripteur sera d'autant meilleur qu'il

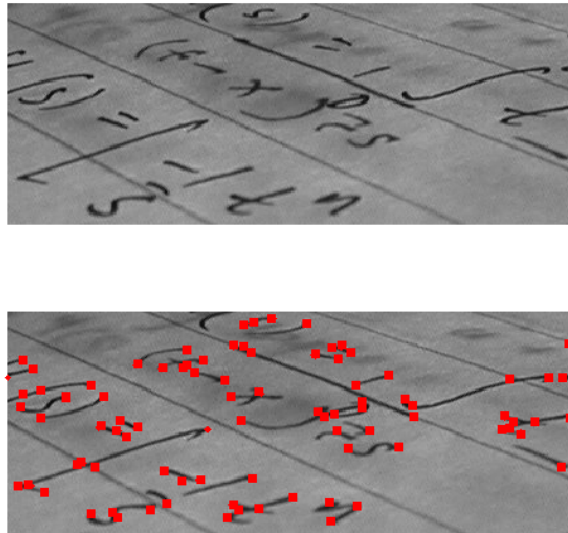


FIGURE 2.1 – Exemple de points d’intérêt détectés sur une image (ici des points de Harris [HS88]).

différencie les zones locales décrites. Il doit aussi être robuste par rapport aux changements possibles (illumination, angle de vue, échelle, rotation...) et aussi à de petites variations de la position du point d’intérêt. Par exemple, l’image 2.1 montre des points d’intérêt (ici des points de Harris [HS88]) obtenus sur une image. Si l’image est déformée ou si la photo est prise d’un autre point de vue, on retrouverait certainement quasiment les mêmes points.

Dans les paragraphes suivants, nous présentons rapidement les différents types de détecteurs de points d’intérêt et de descripteurs. Ensuite, l’algorithme SIFT (pour Scale Invariant Features Transform [Low04]) que nous avons choisi est présenté en détail (il regroupe un détecteur de régions d’intérêt ainsi qu’un descripteur).

2.2.1 Détecteurs de points d’intérêt

Un détecteur fournit les positions dans l’image où sont situés des points d’intérêt. Le descripteur sera alors calculé sur une région autour du point d’intérêt. Le détecteur le plus connu est le détecteur de Harris [HS88]. Ce détecteur, ainsi que tous les détecteurs suivants sont invariants aux rotations. Nous distinguons les détecteurs de régions d’intérêt qui fournissent en plus de la position du point d’intérêt, la région sur laquelle le descripteur doit être calculé. Cette région peut être de forme fixe, mais de taille variable. Cela permet par exemple de fournir une région d’intérêt invariante au facteur d’échelle (Mikolajczyk et al. [MS01, MS04], Lowe et al. [Low99],[Low04]). La forme de la région peut aussi varier pour que le descripteur soit invariant lors d’une transformation

affine de l'image (Harris Affine et Hessian Affine [MS02], MSER [MCMP02]). Une fois le détecteur appliqué, pour chaque région d'intérêt, son descripteur est calculé.

2.2.2 Descripteurs de régions d'intérêt

Le descripteur le plus simple est le vecteur concaténant les intensités des pixels d'une région d'intérêt. Mais cela s'avère très peu robuste à certaines transformations. Pour ne pas être sensible à un changement global d'intensité lumineuse, il est d'autre part nécessaire d'utiliser une mesure de corrélation normalisée (type NCC) et non une distance simple (e.g. L^1 ou L^2). Le calcul de similarité devient alors plus lent.

Des études comparatives de descripteurs locaux plus robustes sont présentes dans [GMDP00, MS02, MTS+05, SMB00]. Toutefois, dans le cadre de cette thèse, l'article le plus intéressant est [MS05] dans lequel l'auteur présente une évaluation comparative de plusieurs descripteurs locaux couplés à divers détecteurs de zones d'intérêt. L'intérêt de cet article est de comparer des couples détecteur-descripteur, et de conclure sur leur efficacité pour une application de recherche de similarités entre images.

La conclusion générale est que les descripteurs basés sur l'algorithme SIFT [Low04] sont les plus performants. Pour le détecteur de région d'intérêt, nous choisissons d'utiliser celui également présenté dans [Low04], qui couplé aux descripteurs de type SIFT obtient les meilleurs résultats. A noter que l'utilisation du détecteur de zone d'intérêt Hessian Affine de [MS02] aboutit à des performances similaires tout en étant invariant aux transformations affines. Toutefois, cela rajoute de la complexité dans la détection des zones d'intérêt, et l'invariance aux transformations affines n'est pas requise pour nos applications si l'on suppose que les photos sont prises de points de vue similaires. Pour cette raison, nous ne l'avons pas utilisé.

Notre but principal n'est pas de modifier ce descripteur mais d'optimiser la recherche des plus proches voisins pour des points qui sont des descripteurs SIFT. Pour cela, il paraît toutefois nécessaire d'étudier précisément la construction de ce détecteur et de ce descripteur, ainsi que les variantes existantes.

2.2.3 Description des SIFT

Dans les travaux de Lowe [Low04] où les SIFT ont été introduits, le terme SIFT désigne l'algorithme complet regroupant un détecteur de points d'intérêt et un algorithme de calcul de descripteur local. Dans notre thèse, le terme SIFT désignera aussi la chaîne complète aboutissant aux vecteurs descripteurs. Pour signifier uniquement l'une des deux étapes, nous parlerons alors de détecteur SIFT ou de descripteur SIFT. Dans les paragraphes suivants, nous décrivons successivement ces deux algorithmes.

Détecteur SIFT de régions d'intérêt

Le détecteur SIFT s'inspire des travaux de Lindeberg présentés dans [Lin98]. Dans cet article, l'auteur analyse en profondeur la détection de points d'intérêt

dans le scale-space (le scale-space est un outil permettant de décrire un signal à différents facteurs d'échelle). Pour une image représentée par un signal 2D f , on note L sa représentation dans le scale-space avec :

$$L(x, y, \sigma) = g(x, y, \sigma) * f(x, y) \quad (2.1)$$

où σ est le facteur d'échelle et g est une gaussienne par laquelle est convoluée l'image :

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x^2+y^2)}{2\sigma^2}} \quad (2.2)$$

C'est-à-dire qu'une représentation dans le scale-space correspond à l'ensemble des lissages par des gaussiennes de l'image étudiée.

Les travaux de Lindeberg montrent qu'à une échelle donnée σ , il est possible de détecter des blobs (terme désignant une région plus claire ou plus foncée que son voisinage) de taille $\sqrt{\sigma}$ en recherchant les extrema de l'opérateur laplacien :

$$\Delta L(x, y) = L_{xx} + L_{yy} \quad (2.3)$$

Ce détecteur ne permet toutefois pas de décider à quelle échelle une zone d'intérêt doit être détectée. Pour choisir automatiquement l'échelle, Lindeberg a proposé dans [Lin98] d'utiliser un opérateur laplacien normalisé selon le facteur d'échelle :

$$\Delta_{norm} L(x, y, \sigma) = \sigma(L_{xx} + L_{yy}) \quad (2.4)$$

Un point d'intérêt est alors détecté si ce point est un extremum de Δ_{norm} (extremum selon trois dimensions : les deux du plan image et la dimension du facteur d'échelle).

Approche par différence de gaussiennes En pratique, le calcul de l'opérateur Δ_{norm} est coûteux en temps. Les travaux de Lindeberg montrent qu'il est possible d'approcher cette fonction par une méthode plus rapide. L'auteur utilise pour cela le fait que la représentation dans le scale-space L satisfait l'équation de la chaleur :

$$\frac{\partial L}{\partial \sigma} = \frac{1}{2} \Delta L \quad (2.5)$$

Cela implique que l'opérateur laplacien peut être approché par une différence finie :

$$\Delta L(x, y, \sigma) = 2 \frac{\partial L}{\partial \sigma}(x, y, \sigma) \approx \frac{2}{(k-1)\sigma} (L(x, y, k\sigma) - L(x, y, \sigma)) \quad (2.6)$$

où σ et $k\sigma$ sont deux facteurs d'échelle proches. Et au final, l'opérateur laplacien normalisé est approché par une différence de gaussiennes :

$$\Delta_{norm} L(x, y, \sigma) \approx \frac{2}{(k-1)} (L(x, y, k\sigma) - L(x, y, \sigma)) \quad (2.7)$$

Dans la suite de ce document, cette différence de gaussienne est notée $D(x, y, \sigma)$:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \quad (2.8)$$

où k est un coefficient multiplicateur et $L(x, y, k\sigma)$ et $L(x, y, \sigma)$ sont deux images convoluées par deux facteurs d'échelle successifs. Cette image $D(x, y, \sigma)$ est souvent appelée Dog pour Difference Of Gaussian. Dans le détecteur SIFT, une pyramide des images $D(x, y, \sigma)$ est construite de cette manière, à différentes échelles (ces différences de gaussiennes approchent la représentation de l'opérateur Δ_{norm}). La détection de blobs se fait ensuite comme dans [Lin98] en cherchant les extrema selon le plan image et la dimension du facteur d'échelle. C'est-à-dire que pour un pixel d'une image $D(x, y, \sigma)$, il est comparé à ses 8 pixels voisins dans la même image ainsi qu'à ses 9 pixels voisins dans l'image d'échelle supérieure et aux 9 pixels voisins de l'image d'échelle inférieure (au final, 26 voisins sont comparés).

Amélioration de la position du point d'intérêt Enfin, l'auteur de [Low04] utilise une étape présentée dans [BL02] pour affiner la position des points détectés. L'expansion de Taylor à l'ordre 2 de D autour du point d'intérêt est utilisée (ici, la fonction D est décalée pour que le point d'intérêt soit à l'origine) :

$$D(x) = D(M) + \frac{\partial D^T}{\partial x} x + \frac{1}{2} x^T \frac{\partial^2 D}{\partial x^2} x \quad (2.9)$$

où x est le décalage par rapport au point d'intérêt M selon les 3 dimensions. La position de l'extremum $M + \hat{x}$ est alors obtenue en annulant la dérivée de cette fonction. On obtient alors :

$$\hat{x} = - \left(\frac{\partial^2 D}{\partial x^2} \right)^{-1} \frac{\partial D}{\partial x} \quad (2.10)$$

Contraste d'un point d'intérêt La valeur $D(\hat{x})$ est utilisée pour mesurer le contraste du point d'intérêt. Les points qui ont un contraste "trop" faible sont éliminés. Comme nous le verrons dans les chapitres suivants, il s'agit d'une étape importante car en augmentant le seuil sur ce contraste, le nombre de descripteurs par image diminue. Et cela influencera fortement les temps d'exécution des algorithmes de recherche d'images similaires.

Suppression des points sur les contours Le détecteur présenté répond aussi sur les contours et génère ainsi des points dont la position n'est pas stable. Pour éliminer les points positionnés sur les contours, l'algorithme SIFT utilise la matrice hessienne de D selon les dimensions spatiales, notée H , :

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \quad (2.11)$$

Les valeurs propres de cette matrice sont en effet proportionnelles aux courbures principales de D et permettent de savoir si le point est situé sur un contour ou pas. En pratique, ces valeurs propres ne sont pas calculées. Comme seul leur ratio est intéressant, l'auteur utilise uniquement la trace et le déterminant de H pour approcher ce ratio (d'une manière similaire au détecteur de Harris [HS88]).

Autre approche de détection de blob possible A noter qu'une approche similaire utilise le déterminant de la matrice hessienne de L au lieu d'utiliser l'opérateur laplacien normalisé. On cherche alors les extrema, selon les 3 dimensions (x, y, t) de la fonction :

$$\det HL(x, y, t) = t^2(L_{xx}L_{yy} - L_{xy}^2)(x, y, t) \quad (2.12)$$

où HL est la matrice hessienne de L . Cette fonction peut être calculée de manière approchée en utilisant des intégrales d'images. Cette méthode rapide a été utilisée comme détecteur de zones d'intérêt dans les descripteurs SURF [BTG06].

Calcul de l'orientation En plus de fournir la position dans le plan image et l'échelle de la zone d'intérêt, le détecteur SIFT fournit l'orientation du point. Cette orientation est obtenue comme l'orientation maximale de l'histogramme des orientations des gradients locaux autour du point d'intérêt. Si plusieurs orientations maximales sont détectées pour un même point d'intérêt, plusieurs descripteurs sont générés. En calculant le descripteur aligné selon cette orientation, l'invariance aux rotations de l'image est assurée.

Descripteur local type SIFT

Le descripteur est construit à partir de l'image d'échelle la plus proche de l'échelle du point d'intérêt. La fenêtre de la zone d'intérêt est divisée en $n \times n$ blocs. Dans chaque bloc, un histogramme de r valeurs est construit, les amplitudes des gradients locaux y sont enregistrées. En général, un histogramme de 8 orientations ($r = 8$) est utilisé. Pour éviter les effets de bords, une interpolation trilinéaire est utilisée pour propager un gradient local dans les cases voisines. Les gradients locaux sont aussi pondérés par une gaussienne qui diminue le poids des gradients les plus éloignés de la position du point d'intérêt. La figure 2.2 illustre la création d'un descripteur SIFT-32 avec une grille de 2×2 histogrammes, chacun ayant 8 orientations possibles.

Pour être robuste à un changement global de contraste, le descripteur est normalisé. Un changement de contraste multiplie les gradients par un coefficient constant. La normalisation du descripteur le rend donc robuste à cette transformation. Il faut noter que ce descripteur est robuste à un changement global de luminosité puisqu'il est calculé à partir de différences de pixel (pour calculer les gradients locaux). Pour être plus robuste aux changements d'illumination non linéaires (par exemple saturation des capteurs ou aux changements d'illuminations sur un objet 3D non plat), les coordonnées du descripteur normalisé

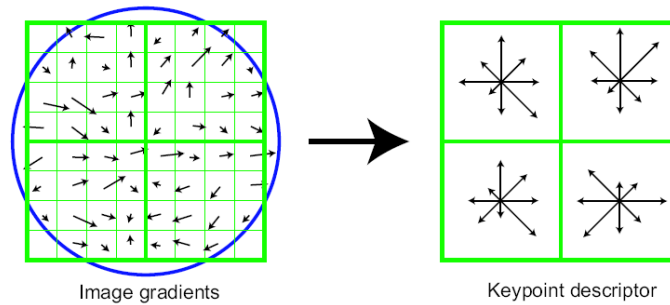


FIGURE 2.2 – Histogrammes des gradients locaux pour construire un vecteur SIFT-32 ($n = 2$ et $r = 8$).

sont bornées à 0.2 et le descripteur est ensuite normalisé une seconde fois. La valeur 0.2 a été déterminée expérimentalement dans [Low04].

En général, les SIFT sont calculés sur une grille 4x4 et des histogrammes de 8 orientations, et sont donc composés de 128 valeurs.

La figure 2.3 illustre des SIFT sur une image réelle et une image synthétique. La figure 2.4 illustre un par un quelques descripteurs SIFT. La première ligne de ce tableau permet de visualiser le descripteur détecté à l'extrémité d'une demi-ellipse. Nous pouvons voir que les histogrammes des cases de droite sont bien symétriques à ceux des histogrammes de gauche et la forme de l'ellipse s'y retrouve bien, du fait que la direction de la fenêtre est horizontale.

2.2.4 Variantes

De nombreux articles ont cherché à améliorer les résultats des SIFT en modifiant quelques points de l'algorithme. Les paragraphes suivants les présentent brièvement.

PCA-SIFT

Dans Ke et al. [KS04], l'auteur utilise le même algorithme que les SIFT pour détecter des points d'intérêt, leurs échelles et leurs orientations. Seul le vecteur descripteur n'est pas le même. Dans SIFT, la grille des gradients locaux est représentée par un descripteur à 128 dimensions. Dans les PCA-SIFT, les auteurs utilisent une réduction de dimensions par analyse en composantes principales (voir [Fod02]) pour réduire la grille de 39x39 gradients locaux à un vecteur de dimension beaucoup plus petite. L'analyse en composantes principales est effectuée sur une large base de grilles de gradients locaux issus d'images diverses. Le meilleur compromis qualité/performance est obtenu lorsque le vecteur initial de gradient ($39 \times 39 \times 2 = 3042$ dimensions) est réduit à un vecteur de 36 dimensions.

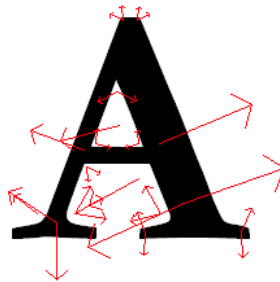
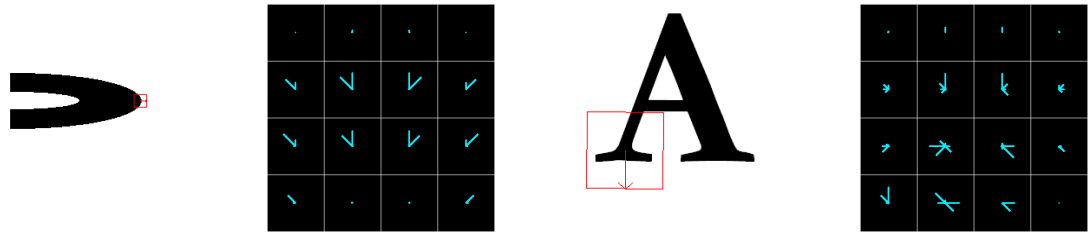
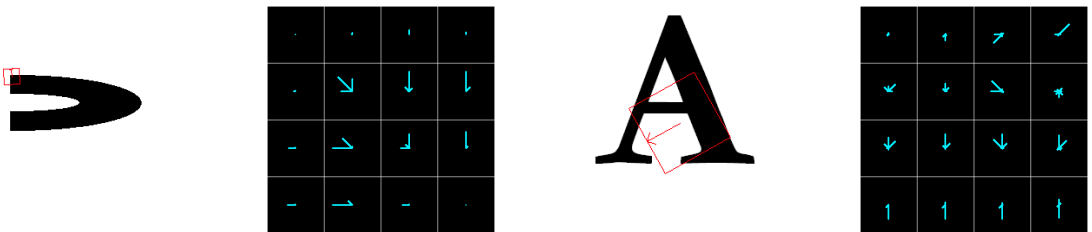


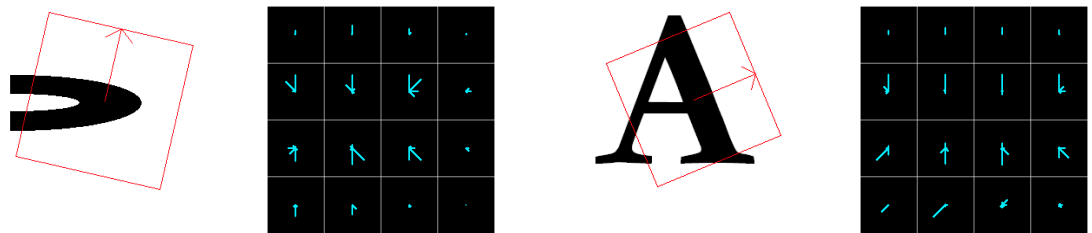
FIGURE 2.3 – Sur chaque image, des SIFT détectés sont représentés par des flèches. Chaque flèche est orientée selon l'orientation du descripteur et sa longueur est proportionnelle à la taille de la zone d'intérêt utilisée pour construire le descripteur.



Ici, le SIFT est orienté vers la droite



Ici, le SIFT est orienté vers le haut



(a)

(b)

(c)

(d)

FIGURE 2.4 – (a) et (c) : Zone d'intérêt d'un descripteur, (b) et (d) : grille 4x4 utilisée pour construire le descripteur. Chaque case contient un histogramme de 8 directions. La direction principale du SIFT correspond à la direction vers le haut de cette grille. C'est-à-dire que la région d'intérêt est tournée pour que la direction du SIFT soit vers le haut.

Gloh-SIFT

Dans [MS05], les auteurs présentent ce descripteur très proche du descripteur SIFT initial. La seule différence vient du fait que les gradients locaux sont représentés dans des histogrammes selon leurs coordonnées en log-polaire, avec 3 cases dans la direction radiale et 8 en angulaire. Cela fait au final 17 histogrammes (la case au milieu n'est pas divisée en 8 directions). Chaque histogramme contient 16 directions possibles. Au final, la dimension est réduite de 272 à 128 par une analyse en composantes principales.

SURF

Dans l'algorithme SURF [BTG06], les auteurs utilisent une méthode approchée (par intégrales d'images, introduites dans [VJ01]) pour calculer les dérivées secondes de la convolution de l'image par une gaussienne, et ensuite en déduire le déterminant de la matrice hessienne. Le descripteur est ensuite calculé. Pour cela, une région rectangulaire alignée sur la direction principale locale est divisée en 16 sous-régions. Dans chaque sous-région, les réponses aux ondelettes de Haar, horizontales et verticales, notées dx et dy sont calculées, en 25 points régulièrement échantillonnés. Pour chacune des 16 sous-régions, son vecteur descripteur est calculé : $v = (\sum dx, \sum |dx|, \sum dy, \sum |dy|)$. Un vecteur à 64 dimensions est ainsi obtenu. Les auteurs introduisent aussi les SURF-128 où sont sommées les valeurs positives d'ondelettes de Haar et les valeurs négatives dans deux résultats distincts (on double ainsi le nombre total de valeurs). Ce dernier descripteur est le plus performant dans leurs tests. Les auteurs enregistrent également pour chaque descripteur si le signe du laplacien (trace de la matrice hessienne) est positif ou négatif (i.e. s'il agit d'un blob foncé sur fond clair ou l'inverse). Cela permet ensuite d'accélérer la recherche de correspondances (seule la moitié des descripteurs est testée pour une requête). Cette idée n'existait pas dans l'algorithme initial des SIFT. A noter qu'une approche similaire d'approximation des SIFT a été utilisée dans [GGB06] sous le nom de Fast Approximated SIFT.

*Eff*²

Les auteurs de [LJA06] présentent quelques heuristiques pour améliorer le descripteur SIFT, qu'ils regroupent sous le nom *Eff*². Ils constatent que la majorité des descripteurs sont calculés à un facteur d'échelle faible (i.e. ce sont des détails dus aux hautes fréquences dans l'image). Ils expliquent cela par le fait qu'à un fort coefficient d'échelle, à cause du lissage gaussien, le contraste des images diminue, générant moins d'extrema au dessus du seuil minimal. Pour contrer cet effet, ils appliquent une correction gamma pour renforcer le contraste proportionnellement au facteur d'échelle. En plus de cela, ils diminuent le seuil de contraste nécessaire pour qu'un extremum soit accepté, proportionnellement au facteur d'échelle. Cela permet de détecter plus de points aux plus hauts niveaux d'échelle, c'est-à-dire correspondant aux basses fréquences de l'image. Pour construire le descripteur, ils n'utilisent plus le module du gradient comme dans SIFT mais la racine quatrième de la norme du gradient. De plus, la taille

de la fenêtre de pixels considérés est augmentée avec l'échelle du point d'intérêt étudié, mais de manière différente à celle de l'algorithme SIFT. Et ils utilisent, comme dans les SURF [BTG06], une distinction entre les régions claires et les régions foncées. Enfin, ils emploient un algorithme différent pour rejeter les points situés sur les lignes, en utilisant le descripteur construit. Ils se servent également de ce descripteur pour éliminer les régions qui sont des tâches sombres ou lumineuses, sans information. A noter que leur descripteur est de type SIFT-72 (3x3 cases avec des histogrammes de 8 directions). Ces descripteurs ont aussi été utilisés dans une application de recherche de vidéo similaires dans [DLÁ+07].

Autres variantes

Dans [MDS05], les auteurs couplent un second descripteur à un descripteur SIFT pour le rendre plus robuste (appelé "SIFT with global context"). Ils utilisent un descripteur de type shape context (voir [BMP02]), pour ajouter au descripteur final des informations à grande échelle. Le shape context est un descripteur qui enregistre un histogramme des distributions des points qui appartiennent à des contours. Ce descripteur shape context obtient des performances inférieures à celles du SIFT dans le comparatif de [MS05]. Dans [MDS05], le descripteur SIFT décrit localement la zone d'intérêt et le shape context décrit la zone à plus grande échelle. Cela permet d'être plus robuste notamment quand les images contiennent des répétitions de motifs.

2.2.5 Conclusion sur le descripteur choisi

Dans ces travaux, nous ne considérerons que des descripteurs issus de l'algorithme SIFT initial. Nous verrons tout de même qu'il est intéressant de modifier la méthode de sélection des zones d'intérêt selon le contraste afin de réduire la taille des nuages de points considérés. Pour certains tests, nous utiliserons aussi des descripteurs SIFT calculés sur d'autres détecteurs de zones d'intérêt (disponibles avec certaines bases d'images publiques). Dans la partie suivante, nous étudions les possibilités de modifier l'algorithme SIFT pour diminuer le nombre de descripteurs par image. Ensuite, nous étudions quelques statistiques des données SIFT qui pourront permettre la mise au point d'algorithmes de recherche des plus proches voisins qui soient adaptés aux données.

2.3 Analyse de données sur des descripteurs SIFT

Nous mesurons d'abord le descripteur moyen à partir de plusieurs bases SIFT. On note \bar{p} ce vecteur pour un nuage de points P (de taille n) :

$$\bar{p} = \frac{1}{n} \sum_{p_i \in P} p_i \quad (2.13)$$

La figure 2.5 montre \bar{p} pour deux nuages de points de test. Deux choses peuvent être observées. Premièrement, le vecteur moyen a des coordonnées très variables selon les dimensions. Et deuxièmement, cette variabilité est quasiment identique quelle que soit la base d'images (nous avons vérifié ce point sur

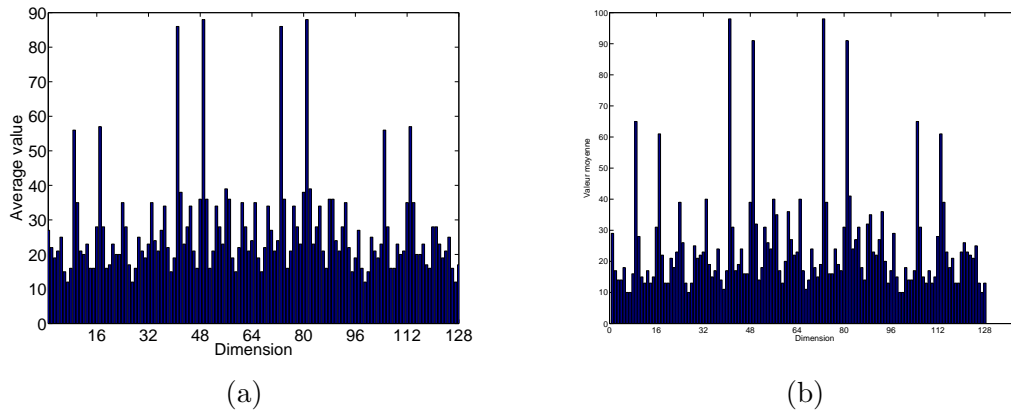


FIGURE 2.5 – Vecteur SIFT moyen sur une base (a) de 620 photos personnelles (826.10^3 descripteurs) (b) de 1000 photos de la base ALOI [GBS05] (192.10^3 descripteurs).

d'autres bases d'images). Nous en concluons que ce vecteur SIFT moyen est une conséquence de l'algorithme de construction des descripteurs. Nous constatons par exemple que quelle que soit la base d'images, le vecteur moyen contient quatre coordonnées qui sont distinctement plus importantes que les autres. Il s'agit des dimensions 41,73,81 et 49 (en énumérant les dimensions sur [1, 128] sur notre implémentation des SIFT).

Pour comprendre pourquoi on retrouve ces quatre dimensions, il faut remonter à l'étape de construction du descripteur. Nous avons vu dans la partie 2.2.3 que le descripteur SIFT est construit à partir d'une grille de 4×4 cases et que chaque case est un histogramme de 8 orientations ($4 \times 4 \times 8 = 128$). L'orientation du descripteur correspond à l'orientation majoritaire dans ces histogrammes. De plus, comme les normes des gradients locaux sont pondérées par une gaussienne centrée au point d'intérêt, les histogrammes les plus proches du centre (i.e. ceux correspondant aux 4 cases du centre) ont des valeurs plus fortes. Au final, cela signifie que pour un descripteur SIFT, il y aura généralement 4 valeurs plus fortes que les autres qui correspondent à la direction du descripteur dans les histogrammes des 4 cases centrales. Cela explique les 4 valeurs qui se détachent sur le descripteur moyen. Sur la figure 2.6, nous avons noté les index des dimensions utilisées dans notre implémentation. Pour une case de la grille 4×4 , parmi les directions $[k, k + 7]$, c'est la direction d'index k qui est alignée sur la direction principale. Nous retrouvons que les coordonnées des cases centrales, alignées sur la direction du descripteur sont les dimensions 41,73,81 et 49 qui correspondent bien à celles mesurées expérimentalement.

Sur la figure 2.5, on voit aussi quatre autres coordonnées importantes, selon les dimensions 9,17,105 et 113. Ces dimensions correspondent aussi à des directions orientées selon la direction principale. On pourrait toutefois s'attendre à trouver les 8 dimensions orientées selon la direction principale et avec le même éloignement du centre (9,17,105 et 113 mais aussi 33,65,57 et 89). L'hypothèse que nous émettons est que les régions d'intérêt sont souvent traversées par une structure (e.g., sur le bord d'un objet par exemple), dans ce cas, les coor-

1-8	33-40	65-72	97-104
9-16	41-47	73-80	105-112
17-24	49-56	81-88	113-120
25-32	57-64	89-96	121-128

FIGURE 2.6 – Index des dimensions utilisées dans notre implémentation des SIFT. Le descripteur est orienté vers le haut de cette grille.

données selon les dimensions 9,17,105 et 113 seront importantes alors que les coordonnées selon les dimensions 33,65,57 et 89 n’ont pas de raison particulière d’être plus élevées que la moyenne.

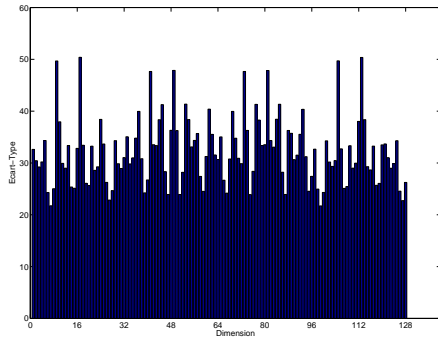
Il est aussi intéressant d’étudier l’écart-type du nuage de points, noté $\sigma(P) = (\sigma(P)_1, \sigma(P)_2, \dots, \sigma(P)_{128})$:

$$\sigma(P)_i = \sqrt{\frac{1}{n} \sum_{j=1}^n (p_i^j - \bar{P}_i)^2} \quad (2.14)$$

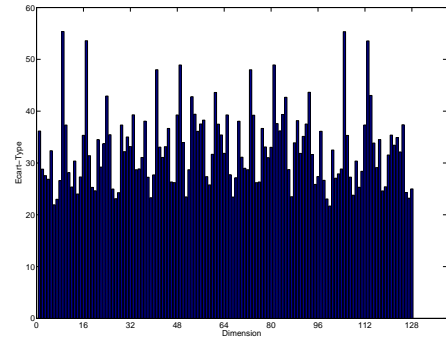
La figure 2.7 montre les écart-types $\sigma(P)$ pour les mêmes nuages que ceux traités sur la figure 2.5. Sur cette figure, les écart-types sont aussi très variables. Dans les deux cas, la valeur maximale est supérieure au double de la valeur minimale. Toutefois, il n’est pas facile de faire la correspondance entre le vecteur moyen et l’écart-type sur cette figure. Pour visualiser cette correspondance, les dimensions sont triées selon la valeur moyenne décroissante et nous affichons les écarts-types selon cet ordre (figure 2.8). Sur les vecteurs moyens, on retrouve les 4 valeurs qui se détachent, avec ensuite 4 coordonnées moyennes, et enfin toutes les autres dimensions avec une décroissance plus faible. Sur les figures où l’écart-type est trié selon le même ordre des dimensions, on voit que cet ordre correspond à une quasi-décroissance de l’écart-type. Ces résultats sont importants pour les algorithmes d’optimisation que nous souhaitons implémenter. Par exemple, pour calculer une distance partielle dans le but d’éliminer des points éloignés, ce sera beaucoup plus efficace de calculer cette distance sur un jeu de dimensions qui maximise l’écart-type (e.g. les dimensions 9,17,105 et 113).

2.4 Algorithmes utilisés pour les applications visées

Dans cette partie, nous présentons plus en détail l’algorithme SIFT+NN+AFF que nous avons utilisé pour l’application de recherche d’images similaires présentée en 1.2.1. Cet algorithme est directement issu de l’article de D. Lowe [Low04]. Nous présentons également une extension de cet algorithme pour l’utiliser sur des images de linéaires de supermarché (application de 1.2.1). Ce dernier algorithme est une contribution applicative de nos travaux. Pour la recherche d’images similaires, il est aussi possible d’utiliser la méthode Bag-Of-Features présentée dans le chapitre 5.

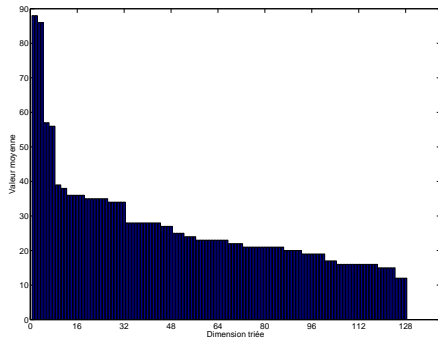


(a)

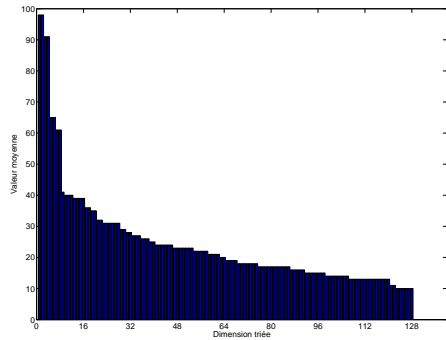


(b)

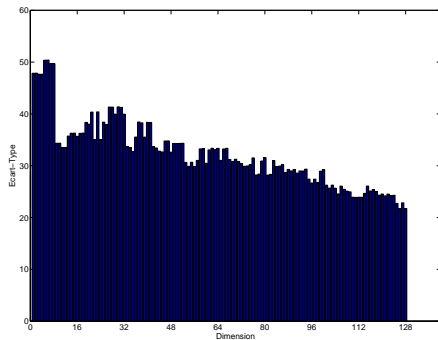
FIGURE 2.7 – Ecart-type des descripteurs SIFT sur une base (a) de 620 photos de vacances (826.10^3 descripteurs) (b) de 1000 photos de la base ALOI [GBS05] (192.10^3 descripteurs).



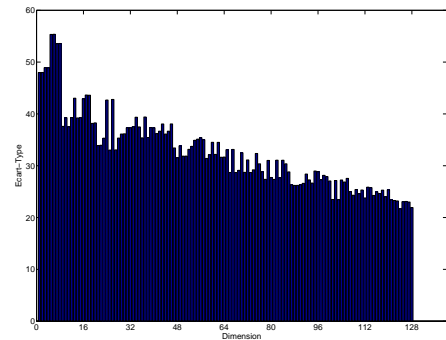
(a)



(b)



(c)



(d)

FIGURE 2.8 – (a) et (b) : Vecteur SIFT moyen sur les deux bases où les dimensions ont été ordonnées par valeur moyenne décroissante. (c) et (d) Ecart-type correspondant, en utilisant le même ordre des dimensions. (a) et (c) sont issues de la base de 620 photos de vacances (826.10^3 descripteurs), (b) et (d) de 1000 photos de la base ALOI [GBS05] (192.10^3 descripteurs).

2.4.1 L'algorithme SIFT+NN+AFF pour la recherche d'images similaires

Comme décrit en 2.1.1, cet algorithme utilisé pour retrouver des images similaires à une image requête comporte 4 étapes :

1. Calcul des descripteurs locaux SIFT des images de la base.
2. Calcul des descripteurs locaux SIFT de l'image requête.
3. Pour chaque descripteur de l'image requête, recherche de ses voisins dans les descripteurs de la base d'images.
4. Analyse des correspondances de points trouvées pour décider si l'image requête est similaire à des images de la base. Cette analyse est faite par recherche d'une transformation affine entre l'image requête et les images de la base (la transformation affine est cherchée à partir des correspondances de points trouvées).

Les deux premières étapes ne font que calculer des descripteurs SIFT. Dans les paragraphes suivants, nous présentons en détail les étapes 3 et 4 de cet algorithme.

Etape 3 : Recherche des correspondances SIFT

Dans cette étape, on cherche à trouver des correspondances entre les SIFT de l'image requête et les SIFT des images de la base. On note $(f_i^q)_{i \in 1..n_1}$ les descripteurs de l'image requête Im_q et $(f_i^{DB})_{i \in 1..n_2}$ les descripteurs de toutes les images de la base de données. n_2 peut avoir une valeur très élevée. Pour chaque descripteur de Im_q , ses voisins sont cherchés dans $(f_i^{DB})_{i \in 1..n_2}$. Pour rechercher les correspondances avec un descripteur f_i^q de Im_q , il existe trois méthodes :

1. Méthode des ratios

Pour un descripteur f_i^q , on cherche f_u^{DB} son plus proche voisin et f_v^{DB} le second plus proche voisin :

$$u = \operatorname{argmin}_{u \in 1..n_2} d(f_i^q, f_u^{DB}) \quad (2.15)$$

et

$$v = \operatorname{argmin}_{v \in 1..n_2, v \neq j} d(f_i^q, f_v^{DB}) \quad (2.16)$$

La correspondance entre f_i^q et f_u^{DB} est acceptée si le ratio $\frac{d(f_i^q, f_u^{DB})}{d(f_i^q, f_v^{DB})}$ est inférieur à un seuil. Cela met en avant que f_i^q doit ressembler beaucoup à f_u^{DB} mais peu à tous les autres descripteurs de la base. Avec cet algorithme, un descripteur de l'image requête est en correspondance avec un seul descripteur de la base.

2. k plus proches voisins (méthode notée k-NN)

On cherche les k descripteurs dans $(f_j^{DB})_{j \in 1..n_2}$ qui sont les plus proches de f_i^q . Si l'on note $G = \{g_c, c \in [1..k]\}$ les indices de ces descripteurs :

$$\forall j \in G \text{ et } \forall l \in 1..n_2, l \notin G, d(f_i^q, f_j^{DB}) < d(f_i^q, f_l^{DB}) \quad (2.17)$$

Dans cette méthode, le nombre de correspondances pour un descripteur de Im_q est k (dans cette formulation, les k voisins ne sont pas classés selon leur distance au point requête).

3. voisins selon un seuil (méthode notée r-NN)

Tous les points de $(f_j^{DB})_{j \in 1..n_2}$ qui sont à une distance avec f_i^q inférieure à un seuil r sont acceptés comme correspondances. Si l'on note $G = \{g_c, c \in [1..k]\}$ les indices de ces descripteurs :

$$\forall j \in G, d(f_i^q, f_j^{DB}) < r \text{ et } \forall j \notin G, d(f_i^q, f_j^{DB}) \geq r \quad (2.18)$$

Cette méthode génère un nombre de correspondances inconnu à l'avance.

Dans l'article qui introduit les SIFT, il est montré que la méthode des ratios donne de meilleurs résultats. Toutefois, elle n'est testée que sur de petites bases. Et d'autre part, elle fait l'hypothèse qu'une région de l'image requête n'est présente que dans une seule image de la base. Dans nos travaux, ce n'est pas toujours vérifié. Par exemple, sur les images des produits de supermarché, le logo d'une marque est présent sur beaucoup d'images. Avec la méthode des ratios, il n'y aurait aucune correspondance sur les descripteurs calculés sur ces logos. On a un problème similaire avec l'approche k-NN. Comment choisir le paramètre k ? Si un logo apparaît dans un grand nombre d'images, il faudra choisir k grand pour avoir des correspondances sur ce logo dans toutes les images qui le comportent. Mais en contrepartie, avec k grand, beaucoup de fausses correspondances vont être acceptées. Dans nos travaux, nous allons donc uniquement utiliser l'approche r-NN.

Cette étape r-NN est un algorithme de recherche des plus proches voisins (aussi appelée "range neighbors search"). C'est sur cette étape que se concentrent nos travaux. Une fois ces correspondances entre descripteurs voisins établies, on va chercher à décider si des images sont similaires ou non. Ceci est réalisé dans l'étape 4 de l'algorithme, dite de vérification affine que l'on présente maintenant.

Etape 4 : vérification affine

A partir des correspondances trouvées, on cherche à valider ou non si l'image requête et des images de la base sont similaires. Pour cela, pour chaque image de la base avec laquelle il y a au moins 3 correspondances de SIFT, une relation affine entre les deux images est cherchée, à partir des correspondances de descripteurs. C'est-à-dire qu'on cherche la matrice associée, du type :

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \quad (2.19)$$

qui transforme un point u , de coordonnées homogènes $(x, y, 1)$ de l'image Im_q en un point $v = (x', y', w')$ de l'image de la base Im_s avec :

$$v = Mu = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2.20)$$

Il serait possible de chercher à approcher cette matrice directement à partir des positions 2D des correspondances SIFT, par exemple avec une méthode par moindres carrés. Toutefois, un certain nombre de ces correspondances sont erronées (dites outliers). On pourrait utiliser un algorithme de moindres carrés robuste dit IRLS (pour Iterative Re-weighted Least Square algorithm) qui utilise des M-estimators (e.g. Tukey, Huber, voir appendice 6 de [HZ04]). Mais la proportion d'outliers est trop élevée dans certains cas. De plus, les descripteurs SIFT contiennent une orientation ainsi qu'un facteur d'échelle qu'il est difficile d'intégrer à une méthode des moindres carrés.

Pour utiliser ces informations, chaque correspondance (f_i^q, f_j^s) est associée à un point dans un espace à deux dimensions :

$$(f_i^q, f_j^s) \mapsto (sc_{i,j}, \theta_{i,j}) \quad (2.21)$$

où f_j^s est le j^{eme} descripteur de Im_s et où $sc_{i,j}$ est le ratio entre les échelles des descripteurs f_i^q et f_j^s , et $\theta_{q,s}$ est la rotation entre ces deux descripteurs. Cet espace à deux dimensions (i.e. ratio d'échelles et rotation) est discrétisé de manière régulière pour servir d'accumulateur (de la même manière que pour une transformée de Hough [DH72]). Le point (ratio d'échelle, rotation) associé à chaque correspondance de SIFT est alors projeté dans cet accumulateur dans 4 cases (selon chaque dimension, dans la case qui contient le point ainsi que la case la plus proche du point afin d'éviter les effets de bord). Chaque case avec plus de trois points correspond à une potentielle transformation affine entre les deux images. Dans l'article de D. Lowe [Low04], cette étape utilise en fait un espace à quatre dimensions pour représenter les paires de SIFT. C'est-à-dire que chaque correspondance (f_i^q, f_j^s) est associée, non plus à un point 2D (ratio d'échelle et rotation), mais à un point 4D :

$$(f_i^q, f_j^s) \mapsto (x_i^q, y_i^q, sc_{i,j}, \theta_{i,j}) \quad (2.22)$$

où (x_i^q, y_i^q) est la position du descripteur sur l'image Im_q . Le point 4D associé à chaque correspondance est alors projeté dans un accumulateur à 4 dimensions (position X, position Y, ratio d'échelles et rotation), dans les deux cases les plus proches selon chaque dimension (afin d'éviter les effets de bord), c'est-à-dire dans 8 cases au final. De même, chaque case avec plus de trois points correspond à une potentielle transformation affine entre l'image requête et l'image de la base. Que l'on utilise un accumulateur 2D ou 4D aboutit au même résultat : des clusters de correspondances de SIFT que l'on veut modéliser par des transformations affines. Pour la recherche d'images similaires, un accumulateur 2D sera plutôt utilisée. Mais pour l'analyse d'images de linéaires, on utilisera plutôt un accumulateur 4D qui prend en compte la position des descripteurs sur l'image requête.

L'étape suivante est, pour chaque case de l'accumulateur utilisé ayant plus de 3 correspondances, de trouver une matrice affine qui approche ces correspondances. Pour être robuste aux outliers (il en reste, même si l'étape précédente permet d'en éliminer beaucoup), un algorithme de type RANSAC est utilisé [FB81]. Cet algorithme est basé sur des tentatives aléatoires pour trouver le bon modèle. Pour chaque jet, l'algorithme initialise une matrice affine M_k (i.e.

une graine) en choisissant au hasard trois correspondances. Ensuite, pour cette matrice, le nombre de correspondances qui sont valides est compté (nombre d’inliers). Une correspondance (f_i^q, f_j^s) valide la matrice M_k si :

$$d(M_k f_i^q, f_j^s) < T_{inlier} \quad (2.23)$$

où T_{inlier} est un paramètre de l’algorithme. Au final, la matrice qui compte le plus de correspondances validées est choisie et est affinée par moindres carrés (en utilisant uniquement les inliers du RANSAC). Il se peut aussi qu’aucune matrice ne soit trouvée, si par exemple il n’existe pas de relation affine entre les deux images.

En fonction de la matrice obtenue et du nombre de correspondances qui valident la transformation affine, il faut décider si oui ou non, il s’agit bien d’images similaires. Nous utilisons ici un critère de décision très simple : le nombre de correspondances. D’autres méthodes pour calculer un score ont été proposées dans la littérature ([Sch99],[Low01]). Au lieu du nombre de correspondances, on peut utiliser la densité de bonnes correspondances, ou le nombre de correspondances divisé par le nombre de descripteurs, ou encore, d’une manière plus générale, on peut chercher à exprimer la probabilité qu’un même objet soit sur les deux images connaissant les correspondances de points. Toutefois, dans nos travaux, nous n’avons utilisé que le score le plus simple, c’est-à-dire le nombre de correspondances validées entre les deux images.

Finalement, l’algorithme retourne une liste des images similaires à l’image requête avec, pour chacune d’elles, un score et la transformation affine avec l’image requête. On note cette liste sous forme d’une liste de triplet :

$$(i_k, M_k, s_k)_k, k \in 1..n \quad (2.24)$$

où i_k est l’indice de l’image de la base trouvée, M_k est la matrice affine correspondante et s_k est le score associé à ce triplet.

2.4.2 Extension de l’algorithme SIFT+NN+AFF pour l’analyse de linéaire

Dans ce problème, l’image requête est une photo de linéaire telle que celle de 2.9 et la base de données contient des images des produits telles que celles de 2.10. Si l’on utilise l’algorithme SIFT+NN+AFF avec un accumulateur 4D sur ces images, on retrouvera chaque produit de la base présent sur l’image. Mais des produits non présents sur le linéaire vont certainement également être retrouvés. La raison principale est qu’il y a de nombreuses zones similaires, notamment les logos. Par exemple, sur la liste d’images de 2.10, deux produits contiennent le logo “Président”. Si l’un de ces produits est présent sur le linéaire, ces deux produits de la base vont être retrouvés. Comment décider alors quels produits garder parmi tous ceux retrouvés par l’algorithme SIFT+NN+AFF ?

Nous proposons ici une extension simple de l’algorithme SIFT+NN+AFF qui répond à ce problème. Cette extension utilise la liste des triplets résultants de SIFT+NN+AFF comme donnée d’entrée. L’objectif est de filtrer ces triplets



FIGURE 2.9 – Exemple de photo d’un linéaire de supermarché.

pour ne garder que ceux correspondant à des produits réellement présents sur l’image.

Filtrage des triplets

Un masque Im_{mask} de la même taille que l’image du linéaire est créé. Dans un premier temps, les triplets sont triés selon leur score, de manière décroissante. Ensuite, les triplets sont testés dans cet ordre sur le masque.

L’image étudiée (l’image i_k) parmi celles trouvées est projetée selon M_k^{-1} sur le masque. L’image 2.11 montre l’exemple d’un masque du linéaire de l’image 2.9 sur lequel la première image trouvée dans la base a été projetée. Sont alors comptés :

- N_{out} : le nombre de pixels de l’image du produit qui arrivent en dehors du masque.
- N_{empty} : le nombre de pixels qui arrivent dans le masque à un endroit vide du masque.
- N_{idem} : le nombre de pixels qui arrivent sur le masque à un endroit déjà occupé par une image du même produit.
- N_{other} : le nombre de pixels qui arrivent sur le masque à un endroit déjà occupé par une image d’un produit différent.

En notant N le nombre de pixels de l’image du produit, nous définissons les ratios associés : $R_{out} = \frac{N_{out}}{N}$, $R_{empty} = \frac{N_{empty}}{N}$, $R_{idem} = \frac{N_{idem}}{N}$ et $R_{other} = \frac{N_{other}}{N}$. En définissant ces quatre ratios et en leur associant quatre seuils, nous décidons d’accepter ou de rejeter le triplet étudié. A noter que la distinction entre R_{idem} et R_{other} permet de tolérer que des produits identiques se chevauchent plus (e.g. lors d’un empilement en quinconce) que des produits différents.

Si le triplet est accepté, les pixels atteints sur le masque sont marqués comme occupés par ce triplet. Au final, nous obtenons une liste de triplets correspondant aux images de la base de données que l’algorithme a trouvées sur l’image, et qui forment un linéaire valide.

Dans les deux applications que nous avons présentées, l’étape de r-NN est



FIGURE 2.10 – Quelques images d’une base de produits de supermarchés.

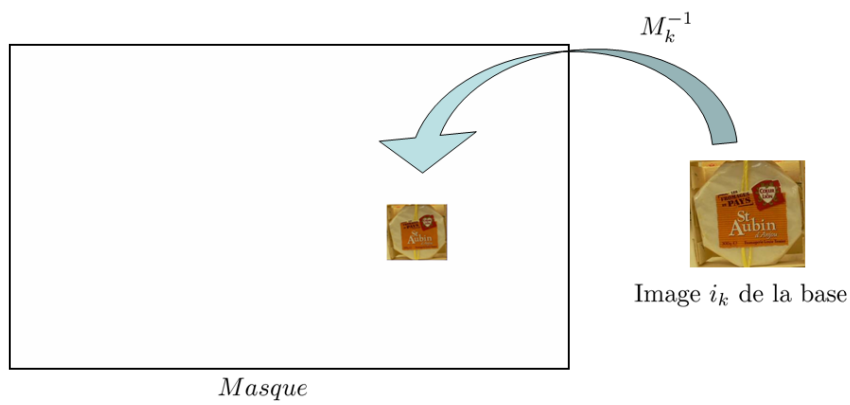


FIGURE 2.11 – Exemple de masque pour l’application d’analyse de linéaire.



FIGURE 2.12 – Sur cette dernière, des images de la base ont été plaquées aux endroits où les produits correspondants ont été détectés.

centrale. Dans le paragraphe 2.6, nous détaillons la recherche linéaire r-NN qui est l’algorithme le plus simple pour cette tâche. Toutefois, avant cela, il nous a paru important d’estimer l’ordre de grandeur du paramètre r pour les recherches r-NN sur des SIFT. En effet le choix de cette taille du rayon de recherche influe fortement sur les performances des algorithmes de recherche.

2.5 Choix de la taille des voisinages recherchés

Pour chercher des correspondances de SIFT entre plusieurs images, nous avons vu qu’un algorithme r-NN est utilisé. Pour choisir le seuil r , il est important de savoir quel est l’ordre de grandeur des distances qui séparent deux SIFT représentant le même détail de deux images. Notre but ici n’est pas de déterminer précisément le seuil de distance qui sera utilisé pour trouver des correspondances de SIFT. Nous cherchons uniquement à avoir un ordre de grandeur des distances entre vecteurs SIFT.

2.5.1 Méthodologie

Pour cela, nous procédons de la façon suivante : à l’aide des descripteurs SIFT, nous cherchons la relation affine entre une image et une déformation de cette image (pour les transformations synthétiques, cette transformation est déjà connue). Ces transformations sont par exemple du type inclinaison, flou gaussien, ajout de bruit. Ensuite, nous cherchons toutes les correspondances de SIFT qui sont validées par cette transformation. Pour chacune de ces correspondances vraies, nous calculons la distance euclidienne (en dimension 128) qui sépare les deux descripteurs. Ensuite nous étudions les distributions de ces distances pour diverses transformations. Si la transformation est simple, les descripteurs sont peu modifiés et les distances sont faibles. Inversement, si la transformation est “difficile” pour les SIFT, les distances seront plus im-



FIGURE 2.13 – Exemples de transformations d’inclinaison.

portantes. Dans les paragraphes suivants, pour plusieurs transformations, nous montrons ces histogrammes des distances entre paires de SIFT correspondant aux mêmes détails.

2.5.2 Transformation d’inclinaison

Nous commençons par étudier les transformations d’inclinaison selon l’axe horizontal. La figure 2.13 montre les trois inclinaisons étudiées pour une même image initiale. Cette transformation est intéressante car elle correspond à un léger changement de point de vue. Sur la figure 2.14, nous montrons les distributions des distances entre correspondances SIFT valides pour plusieurs images pour ces trois transformations d’inclinaison. Sur chaque histogramme, en abscisse, on a la distance entre SIFT calculée par une distance euclidienne en dimension 128. En ordonnée, il s’agit du nombre de paires de SIFT pour lesquelles cette mesure a été observée. Il est important de voir que les échelles varient entre les histogrammes pour l’axe des ordonnées. En effet, pour certaines images avec beaucoup de détails, il y aura plus de paires de SIFT dans l’histogramme que si l’image est petite avec peu de détails.

Sur la figure 2.14, il apparaît logiquement que le pic des distances est placé à une distance plus élevée pour une forte inclinaison que pour une faible inclinaison. Pour la transformation InclinaisonX 5, un seuil de 150 permettrait de retrouver une majorité des correspondances. Toutefois, si l’on considère la transformation InclinaisonX 15, il faut plutôt placer le seuil vers 200.

2.5.3 Transformation de lissage gaussien

Nous étudions aussi le lissage de l’image par une gaussienne. Cette transformation est intéressante car elle simule un léger “bougé” lors de la prise de vue, ce qui peut arriver notamment lors de photo en intérieur avec un flash mal réglé ou pas assez puissant. La transformation $Gauss_n$ correspond à un filtrage gaussien où le noyau est de taille $n \times n$. L’écart-type σ pris pour la gaussienne est alors $\sigma = (n/2 - 1) \times 0.3 + 0.8$ (permettant de conserver la majorité de la gaussienne sur cette taille de noyau). La figure 2.15 montre des exemples de cette transformation. La figure 2.16 montre les distributions des distances entre correspondances SIFT valides. Pour ces transformations, la distance entre paires de SIFT d’une correspondance valide n’augmente pas autant que pour

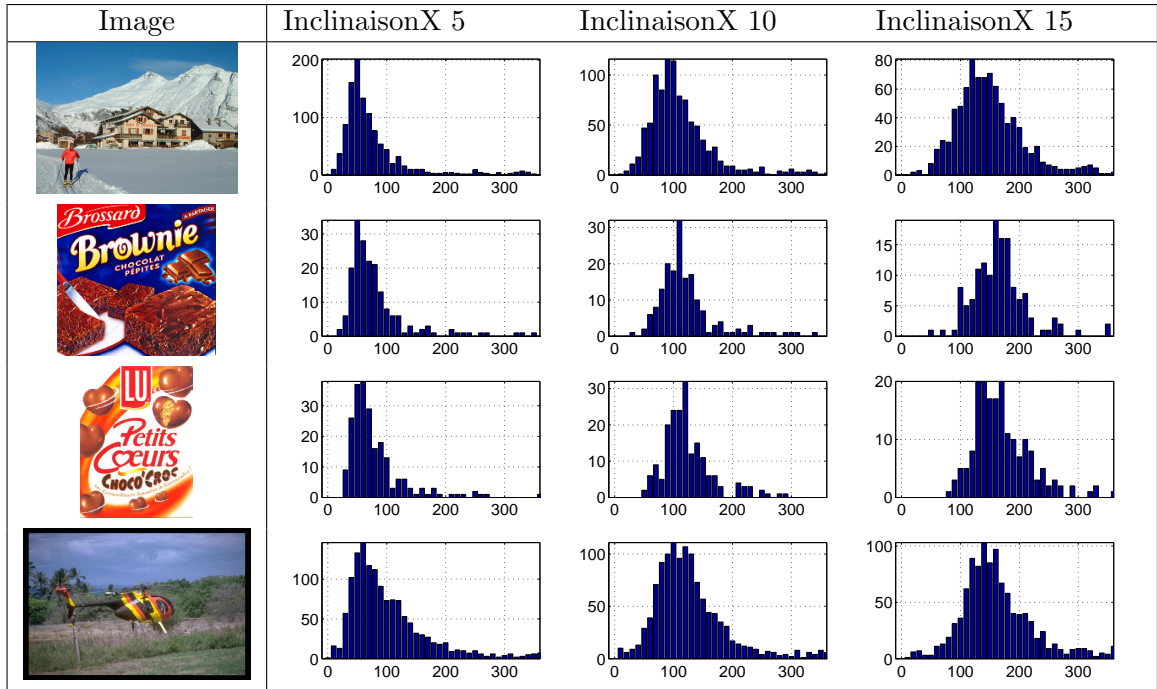


FIGURE 2.14 – Distribution des distances entre SIFT pour les correspondances valides suite à des transformations d’inclinaison.



FIGURE 2.15 – Exemples de transformations de lissage gaussien.

les transformations d’inclinaison. Si l’on considère la transformation $Gauss_5$, un seuil à 150 permet par exemple de conserver une majorité des correspondances.

2.5.4 Transformation d’ajout de bruit gaussien

Nous étudions aussi la transformation d’ajout de bruit gaussien à l’image. Cette altération permet de simuler le bruit dû à une différence de sensibilité entre deux photos. Ce bruit devient important lorsque l’appareil est réglé sur une sensibilité de 400 ISO, et devient très gênant sur les très hautes sensibilités (e.g. 1600 ISO). Dans l’application d’analyse de linéaire, les images de produits sont généralement réalisées en studio, avec appareil sur trépied, et utilisation d’une faible sensibilité pour obtenir une bonne qualité d’image. Inversement, les photos de linéaires sont prises sans trépied, en intérieur, et l’utilisateur aura souvent le réflexe de régler son appareil sur 400 ISO pour minimiser l’utilisation

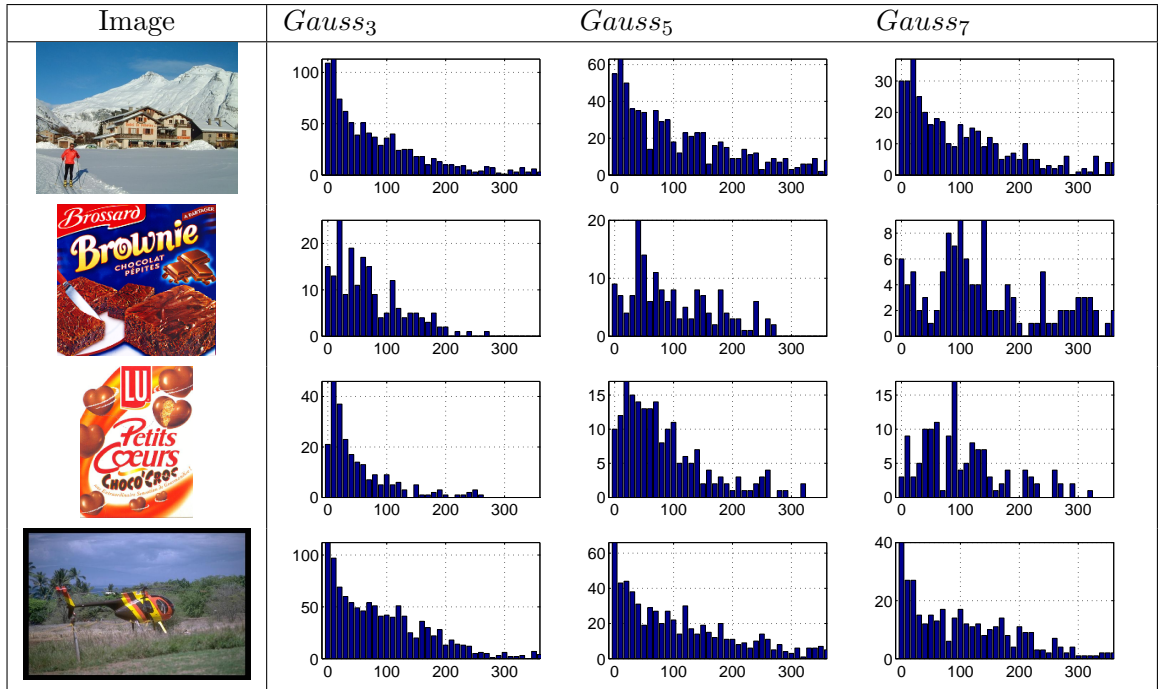


FIGURE 2.16 – Distribution des distances pour les correspondances valides suite à des transformations de lissage gaussien.



FIGURE 2.17 – Exemples de transformations d’ajout de bruit gaussien.

du flash et éviter les risques de “bougé”. Nous appelons $GaussNoise_{\sigma}$ la transformation qui ajoute au pixel un bruit gaussien d’écart-type σ . Les distances entre SIFT correspondant sont montrées sur la figure 2.18. Un seuil d’environ 200 permet encore une fois de conserver une majorité des correspondances.

2.5.5 Combinaisons de transformations précédentes

Nous nous intéressons désormais à la combinaison des transformations précédentes. Nous choisissons comme ordre des combinaisons : inclinasion puis flou gaussien puis bruit gaussien. La première transformation simule un changement de point de vue, la seconde simule un flou, enfin, la dernière simule un bruit de capteur. Nous nommons $Transfo_1$ la combinaison de $Etirement_5$, $Gauss_3$ et $GaussNoise_{10}$; $Transfo_2$ est la combinaison de $Etirement_{10}$, $Gauss_5$ et $GaussNoise_{20}$ et $Transfo_3$ est la combinaison de $Etirement_{15}$, $Gauss_7$ et

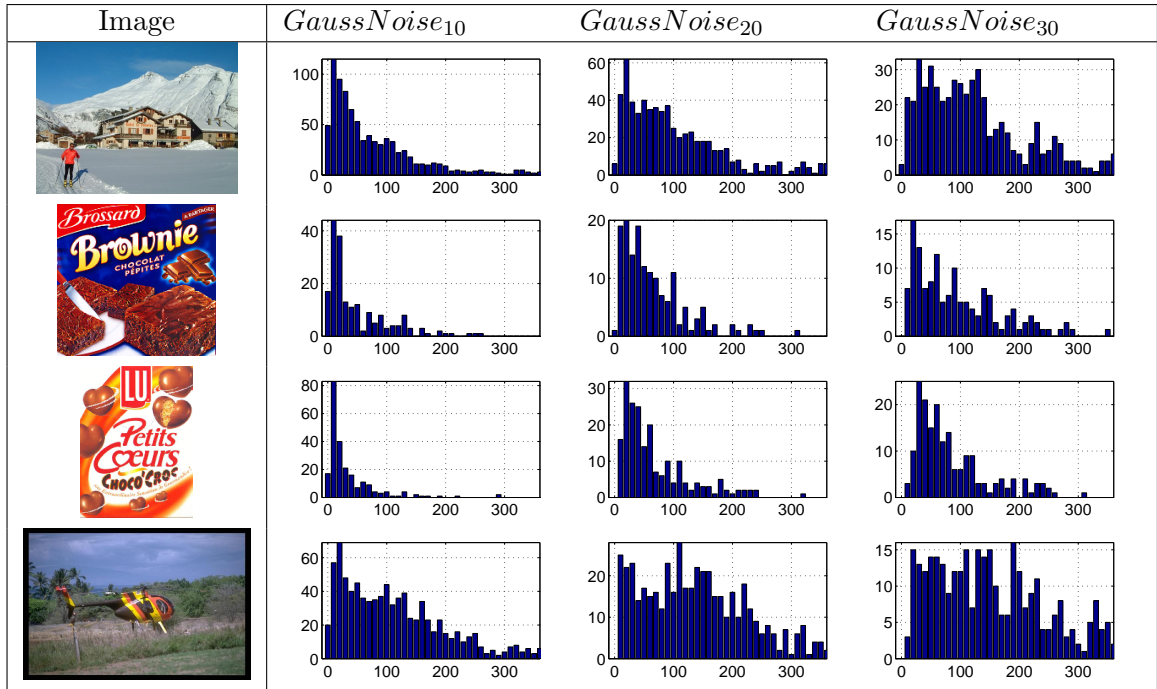


FIGURE 2.18 – Distribution des distances pour les correspondances valides suite à des transformations d’ajout de bruit gaussien.

$GaussNoise_{30}$. Ces combinaisons correspondent à un ordre de difficulté croissant. La figure 2.19 montre les résultats de ces transformations sur une image. Les distributions des distances entre SIFT correspondant valides sont montrées sur la figure 2.20. Pour la transformation la plus difficile, il est nécessaire d’utiliser un seuil égal à 250 si l’on souhaite conserver une majorité des correspondances.

2.5.6 Transformations réelles

Enfin, après nous être penchés sur des déformations synthétiques, nous nous intéressons à des déformations réelles pour des images de produits de supermarché. C’est-à-dire que nous étudions les distances entre les SIFT pour une

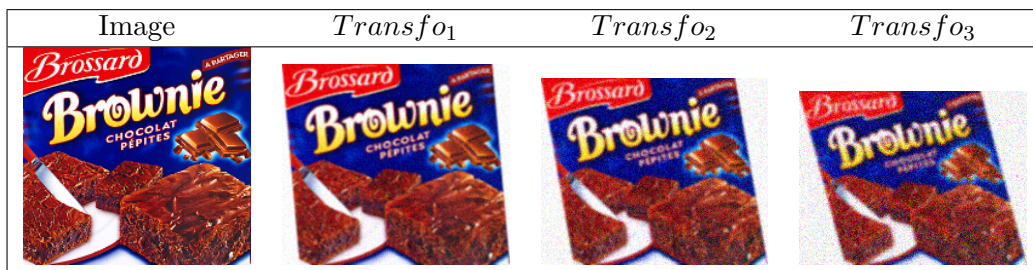


FIGURE 2.19 – Exemples de transformations combinées (Etirement puis flou gaussien puis bruit gaussien).

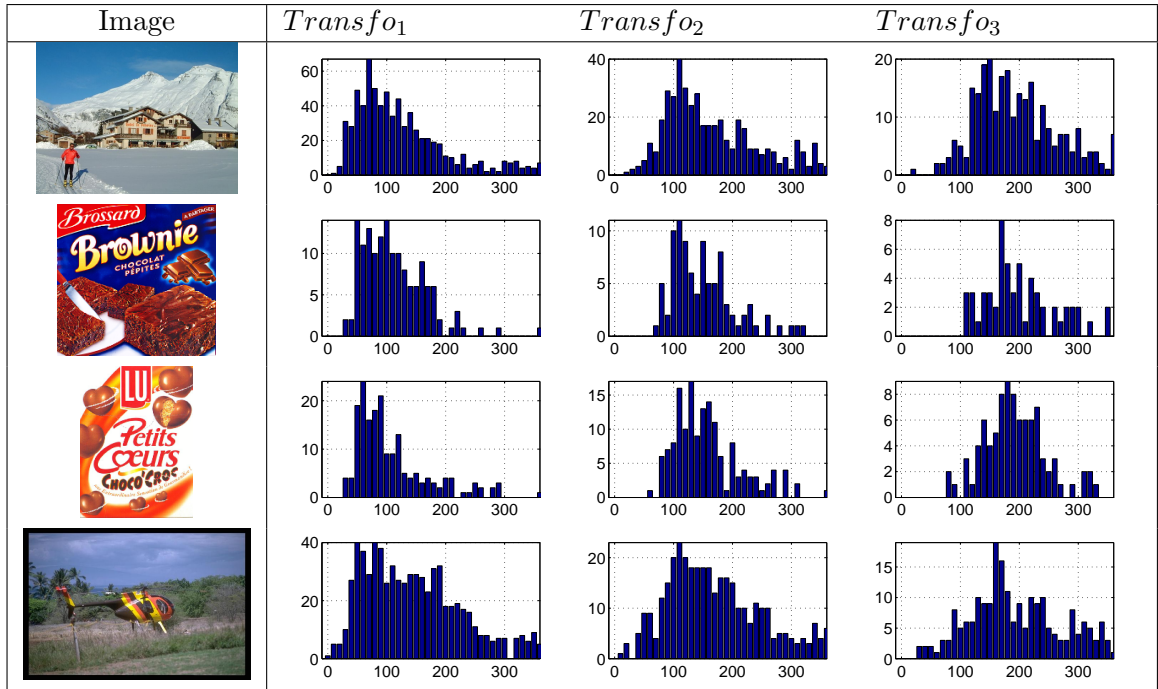


FIGURE 2.20 – Distribution des distances pour les correspondances inliers pour des transformations combinées.

image de la base et une image de ce même produit, découpée sur une photo d'un linéaire de supermarché. La figure 2.21 montre ces images ainsi que la distribution des distances associées. Ces distributions montrent que pour certaines images réelles, les distances entre SIFT correspondant aux mêmes détails sont plus importantes que lors des mesures sur des déformations synthétiques. Par exemple, pour l'un des fromages, son emballage est légèrement froissé sur une des images. Ce genre de déformation est difficile à simuler mais il existe sur ces images réelles. Ainsi, il est bon de considérer un voisinage allant jusqu'à 300. A l'inverse, nous choisissons d'utiliser 150 comme seuil minimal. En dessous de cette valeur, seules les correspondances appartenant à des images très peu différentes sont trouvées. Nous verrons dans les tests suivants que ces valeurs sont très importantes quant au choix de l'algorithme à utiliser. Certains algorithmes seront très rapides pour retrouver les SIFT voisins dans un rayon de 150 mais seront moins efficaces par rapport à d'autres algorithmes pour un seuil de 300.

Dans les paragraphes précédents, nous avons vu que nous pouvions limiter nos tests à des valeurs de r comprises entre 150 et 300. Dans le paragraphe suivant, nous présentons l'algorithme le plus simple pour faire des recherches r-NN, c'est-à-dire la recherche linéaire.

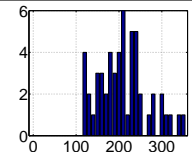
Image DB	Image extraite	Distribution des distances
		
		
		
		
		
		
		
		
		

FIGURE 2.21 – Distances entre SIFT pour des transformations réelles.

2.6 Recherche linéaire

L’algorithme r-NN le plus simple consiste à mesurer la distance euclidienne entre le point requête et tous les points de la base, et à décider quels sont les points voisins en comparant cette distance au seuil r . Nous appellerons cette méthode “recherche linéaire” (elle est parfois également appelée “brute force search”). Du fait de la malédiction de la dimension, cette méthode de recherche linéaire n’est pas à négliger. En outre, pour plus tard évaluer correctement le gain d’algorithmes plus évolués, il est nécessaire de disposer d’une implémentation optimale de la recherche linéaire. Dans ce but, nous montrons qu’en utilisant une distance partielle, cette méthode linéaire peut être simplement accélérée. Ensuite, nous mesurons les performances d’une implémentation sur carte graphique (en utilisant le parallélisme des GPU).

2.6.1 Implémentation de la méthode linéaire sur CPU

Algorithme initial

L’algorithme linéaire de recherche des plus proches voisins est très simple. Dans le cas d’une recherche r-NN, tous les points sont parcourus et on ne conserve que ceux dont la distance avec le point requête est en dessous du seuil choisi. Pour une recherche k-NN, il faut de même parcourir tous les points et une structure de queue avec priorité (implémentée sous forme de tas) est utilisée pour conserver la liste des k plus proches. La version r-NN est présentée par l’algorithme 1.

```
Input: P : nuage de points
Input: r : seuil (pour être voisin)
Input: q : point requête
Output: L : liste des points voisins
1 L vide;
2  $r_2 = r \times r$ ;
3 foreach point  $p \in P$  do
4    $d = 0$ ;
5   foreach  $i \in [1, 128]$  do
6      $diff = (p[i] - q[i])$ ;
7      $d+ = diff \times diff$ ;
8   end
9   if  $d \leq r_2$  then
10    add  $p$  to  $L$ 
11  end
12 end
```

Algorithm 1: Algorithme de recherche linéaire des plus proches voisins (recherche r-NN).

Cet algorithme est très simple à implémenter. Il faut tout de même être prudent sur la façon dont est compilé ce programme. Le tableau de la figure 2.22 montre trois temps d’exécution pour cet algorithme. La ligne “aucune op-

timisation” correspond à un code compilé où toutes les options d’optimisation du compilateur (dans notre cas Microsoft Visual C++ 7.1) ont été désactivées, la ligne “optimisation du compilateur” signifie que toutes les options du compilateur ont été activées. Cela a par exemple pour conséquence de dérouler les boucles. C’est-à-dire qu’au lieu d’exécuter une seule fois la ligne 5 de l’algorithme, puis de tester si l’on doit continuer ou non la boucle, le code fait plusieurs exécutions de cette ligne et ensuite teste s’il faut sortir de la boucle. Nous avons mesuré dans le code assembleur généré que notre compilateur déroule les boucles sur 4 lignes au maximum. C’est-à-dire qu’il exécute 4 fois la ligne 5 et l’incréméntation du compteur avant de tester s’il doit sortir de la boucle ou non (cela est possible car 128 est un multiple de 4).

Enfin, nous testons aussi une implémentation où nous déroulons manuellement toute la boucle (voir algorithme 2). Pour cet implémentation, nous conservons activées toutes les optimisations du compilateur. Les performances de cette implémentation sont présentées en dernière ligne du tableau 2.22.

Dans ce tableau, nous voyons que le temps d’exécution diminue beaucoup lorsque les optimisations du compilateur sont activées. D’autre part, la méthode de déroulement manuel de la boucle fait gagner encore 17% par rapport aux seules options de compilation du compilateur. C’est cette implémentation que nous utiliserons dans nos calculs des temps de référence pour la recherche linéaire sur le CPU.

```

Input: P : nuage de points
Input: r : seuil (pour être voisin)
Input: q : point requête
Output: L : liste des points voisins
1 L vide;
2  $r_2 = r \times r$ ;
3 foreach point  $p \in P$  do
4    $d = 0$ ;
5    $diff = (p[1] - q[1])$ ;
6    $d+ = diff \times diff$ ;
7    $diff = (p[2] - q[2])$ ;
8    $d+ = diff \times diff$ ;
9    $diff = (p[3] - q[3])$ ;
10   $d+ = diff \times diff$ ;
11  ...;
12   $diff+ = (p[128] - q[128])$ ;
13   $d+ = diff \times diff$ ;
14  if  $d \leq r_2$  then
15    | add  $p$  to  $L$ 
16  end
17 end

```

Algorithm 2: Algorithme de recherche linéaire des plus proches voisins (recherche r-NN) avec déroulement de boucle.

Variante	Temps d'exécution (ms)
Aucune optimisation, Algorithme 1	19413
Optimisation du compilateur, Algorithme 1	3362
Déroulement de boucle manuel, Algorithme 2	2782

FIGURE 2.22 – Temps d'exécution de 30 recherches de plus proches voisins (r-NN) dans un nuage de 976.000 points. Les 30 vecteurs SIFT requêtes sont extraits de l'image "lettreA.jpg" (figure 2.23) et les 976.000 points d'une base de 620 images personnelles. A noter que pour ce test, les valeurs des vecteurs SIFT n'ont pas d'influence sur le temps d'exécution de l'algorithme. Le choix de la base d'images importe donc peu.



FIGURE 2.23 – Image "lettreA.jpg"

2.6.2 Que peut-on changer ?

Il reste peu de possibilités pour diminuer le temps d'exécution de l'algorithme de recherche linéaire. Nous pouvons tout de même remarquer que comme les SIFT sont normés, nous pouvons utiliser le développement de la distance euclidienne. C'est-à-dire que pour deux vecteurs SIFT a et b , on a $\|a\| = \|b\| = 1$. La distance euclidienne entre ces deux points peut alors s'exprimer par :

$$d(a,b)^2 = \|a - b\|^2 = \|a\|^2 + \|b\|^2 - 2 \langle a|b \rangle = 2 - 2 \langle a|b \rangle \quad (2.25)$$

où $\langle .|. \rangle$ désigne le produit scalaire. Nous avons testé cette implémentation. Intéressante en théorie, elle n'apporte qu'un gain très mineur sur notre machine (moins de 5%). Nous ne la considérerons donc pas pour la suite des tests. Elle est toutefois importante à garder à l'esprit pour certaines architectures matérielles. Par exemple, les derniers processeurs Intel (familles dites Penryn et Nehalem) proposent des instructions (jeu appelé SSE4) qui exécutent un produit scalaire entre deux vecteurs à 4 dimensions en une seule opération. Sur ces processeurs, en utilisant ces instructions, on peut espérer que cette méthode pour calculer la distance entre deux SIFT soit plus rapide. A la fin de nos travaux, nous avons pu utiliser un processeur avec ce jeu d'instructions. Nous avons pu vérifier que l'exécution est quatre fois plus rapide lorsque les distances sont calculées sur des nombres réels codés sur 32 bits. Toutefois, dans notre implémentation, pour des raisons de mémoire, les SIFT sont stockés sur des entiers non signés codés sur 8 bits. Dans ce cas, la distance calculée entre "float" 32 bits n'est que deux fois plus rapide que la distance utilisée dans nos tests (calculée entre "unsigned char" 8 bits). La conclusion est qu'en utilisant ces instructions, on pourrait être deux fois plus rapide, mais cela impose de stocker les SIFT sur des réels 32 bits, et l'on consomme alors quatre fois plus de mémoire. Dans la suite des tests, les SIFT sont des vecteurs stockés sous forme d'entiers non signés 8 bits.

Un autre intérêt de cette approche par produit scalaire est de pouvoir calculer les distances entre deux nuages de points sous forme de multiplication matricielle. Si l'on note A et B les matrices contenant ces nuages de points (chaque colonne est un vecteur SIFT). On note R la matrice contenant les distances entre les points de A et les points de B , telle que $R_{i,j}$ soit la distance entre le point de la colonne i de A et le point de la colonne j de B . On peut exprimer R par :

$$R = M_2 - 2 \times A^T \times B \quad (2.26)$$

où M_2 est une matrice de taille i, j remplie de 2. Toutefois, avec notre implémentation de la multiplication matricielle, nous n'avons pas constaté de gain en temps d'exécution par rapport à l'algorithme 2.

En dehors des modifications mineures d'implémentation, il existe peu de véritables changements algorithmiques de la recherche linéaire. La seule référence que nous ayons trouvée est l'algorithme BOND (pour "Branch and Bound") de [dVMNK02] pour la recherche $k - NN$. Cet algorithme original utilise des calculs de distances partielles pour accélérer une recherche exacte

des k proches voisins. Par distance partielle, on entend une distance calculée sur un sous-ensemble des 128 dimensions :

$$d_E(f_1, f_2) = \sqrt{\sum_{i \in E} (f_1[i] - f_2[i])^2} \quad (2.27)$$

où E est un sous-ensemble de $\{0, 1, \dots, 127\}$. Les distances partielles sur un premier sous-ensemble des dimensions sont calculées entre le point requête q et les points du nuage. La distance partielle du k^{ieme} point le plus proche permet d'éliminer les points qui sont déjà trop éloignés de q (selon la distance partielle). En itérant ce processus tout en rajoutant des dimensions dans la distance partielle, on diminue progressivement la taille de l'ensemble des voisins potentiels. L'algorithme est très lié au type de données et à l'ordre dans lequel sont utilisées les dimensions pour calculer les distances partielles. Il faut en effet trouver les dimensions pour calculer les distances partielles qui vont dès le début de l'algorithme éliminer un maximum de points. Cet algorithme est utilisable uniquement pour les recherches $k - NN$. Toutefois, nous avons tenté d'utiliser l'idée de distances partielles sur une recherche r-NN.

L'idée est que, pour une recherche r-NN, une distance partielle calculée sur un petit nombre de dimensions peut permettre d'éliminer rapidement certains points qui ne peuvent pas être des voisins du point requête. On présente cette solution dans le paragraphe suivant. Enfin, une dernière approche consiste à paralléliser l'algorithme linéaire r-NN et à l'exécuter sur plusieurs processeurs. On présentera une solution de ce type dans la partie 2.6.4 où l'exécution se fait sur la carte graphique.

2.6.3 Utilisation de distances partielles pour accélérer la recherche linéaire

Au vu de l'écart-type des données SIFT (voir partie 2.3), il est intéressant d'étudier comment évolue le calcul de distance au fur et à mesure que des dimensions sont rajoutées. La figure 2.24 montre l'évolution de distances partielles au fur et à mesure que des dimensions sont rajoutées (à l'ensemble E), pour deux façons de sélectionner ces dimensions. Sur la courbe de gauche, l'ordre est l'ordre croissant des dimensions (de 1 à 128). Sur la courbe de droite, les dimensions sont triées selon l'ordre décroissant de l'écart-type des descripteurs. On voit que la distance partielle augmente plus rapidement si les dimensions sont triées selon cette seconde méthode. Par exemple, pour atteindre une distance de 250, il faut en moyenne 32 dimensions si elles ne sont pas triées contre moins de 16 si elles le sont. On peut conclure deux choses de ces courbes. Premièrement, cela peut être intéressant d'utiliser une distance partielle pour éliminer les points trop éloignés. Et deuxièmement, pour calculer cette distance partielle, il est bon de choisir en premier les dimensions qui ont un fort écart-type.

Dans la partie suivante, nous utilisons cette analyse des points SIFT pour calculer une distance partielle basée sur le tri des dimensions par écart-type décroissant.

Au vu des résultats du paragraphe précédent, nous proposons d'utiliser une

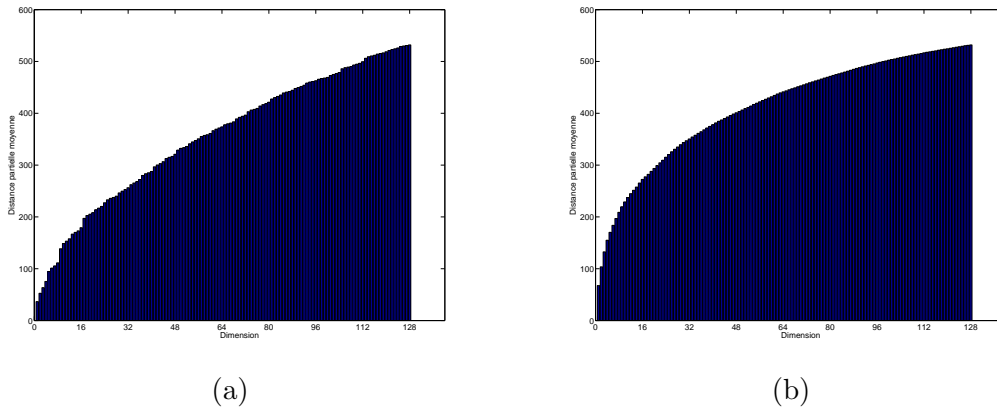


FIGURE 2.24 – Distance partielle calculée selon (a) l’ordre initial des dimensions (b) l’ordre fourni par le tri des dimensions par écart-type décroissant.

distance partielle pour accélérer la recherche des plus proches voisins par l’algorithme de recherche linéaire. Nous utiliserons les dimensions triées par ordre décroissant d’écart-type. Il faut toutefois choisir le nombre de dimensions à utiliser. Il serait possible de calculer et tester la distance partielle pour tous les nombres de dimensions entre 1 et 128. C’est-à-dire que lors du calcul de la distance entre deux points, à chaque fois qu’une dimension est rajoutée à la distance, nous testons si il faut garder le point comme un voisin potentiel ou non. Toutefois, cela introduit un saut conditionnel dans le code pour chaque dimension et ralentit ainsi beaucoup l’exécution. Nous préférons donc tester si la distance partielle est trop grande seulement à quelques dimensions. Par exemple, nous pourrions tester seulement la distance partielle après 64 dimensions, et ainsi, dans une majorité des cas quitter le calcul de distance. Nous cherchons l’algorithme optimal en temps pour ce calcul de distance partielle.

Pour étudier les performances théoriques d’un tel algorithme, nous nous plaçons dans le cadre d’une recherche r-NN et décidons de comparer la distance partielle avec le seuil choisi une seule fois au cours des 128 itérations du calcul

de distance (voir algorithme 3).

```

Input: P : nuage de points
Input: T : seuil (pour être voisin)
Input: q : point requête
Input: n : dimension à laquelle la distance partielle est testée
Output: L : liste des points voisins
1  $T_2 = T \times T$ ;
2 foreach point  $p \in P$  do
3    $d = 0$ ;
4   foreach  $i \in [1, n]$  do
5      $diff = (p[i] - q[i])$ ;
6      $d+ = diff \times diff$ ;
7   end
8   if  $d \leq T_2$  then
9     foreach  $i \in [n + 1, 128]$  do
10       $diff = (p[i] - q[i])$ ;
11       $d+ = diff \times diff$ ;
12    end
13    if  $d \leq T_2$  then
14      add  $p$  to  $L$ 
15    end
16  end
17 end

```

Algorithm 3: Algorithme de recherche linéaire des plus proches voisins (recherche r-NN) par distance partielle avec un seul test.

On note $P_T^L(n) = Pr(d_n(p, q) > T)$ la probabilité que la distance partielle qui utilise les n premières dimensions de la liste L soit supérieure au seuil T . On note L_σ la liste des dimensions triées par écart-type décroissant. Pour choisir le nombre n de dimensions optimal, nous cherchons à exprimer le gain en temps G_t de l'algorithme en fonction de n . On peut l'exprimer par :

$$G_t(n) = \frac{\text{temps nouvel algorithme}}{\text{temps ancien algorithme}} = \frac{n \times P_T^{L_\sigma}(n) + 128 \times (1 - P_T^{L_\sigma}(n))}{128} \quad (2.28)$$

où n est l'itération à laquelle la distance partielle (i.e. calculée sur n dimensions) est comparée au seuil. $P_T^{L_\sigma}(n)$ représente la probabilité que la distance partielle soit supérieure au seuil T . Dans le cas où la distance partielle est supérieure au seuil, l'algorithme sort de la boucle et aura utilisé uniquement n itérations au lieu des 128. Et si la distance partielle est inférieure au seuil (selon une probabilité $(1 - P_T^{L_\sigma}(n))$), le temps de calcul est identique à celui de l'algorithme initial (où les 128 dimensions sont utilisées). Les probabilités $P_T^{L_\sigma}$ sont mesurées expérimentalement. Nous négligeons ici l'impact du temps requis pour comparer la distance partielle au seuil.

Avec la définition du gain ci-dessus, un gain inférieur à 1 signifie que le nouvel algorithme est plus rapide que l'ancien algorithme. On cherche donc à

obtenir le gain le plus petit possible.

La figure 2.25 montre cette fonction de gain pour différentes valeurs de seuil (sur une courbe, seul le nombre de dimensions n utilisé varie). Pour chacun de ces graphiques, nous avons tracé le gain en utilisant les dimensions triées de L_σ ou les dimensions non triées. Dans chaque cas, le fait de trier les dimensions apporte un gain. Il existe un nombre de dimensions optimal pour lequel le gain en temps est maximal. Nous constatons logiquement que pour un seuil bas (e.g. 150), il faut tester la distance partielle avec peu de dimensions (e.g. 16). Par contre, pour un seuil élevé (e.g. 300), il faut tester une distance partielle qui contient plus de dimensions (e.g. 40). Cela confirme qu'il est toujours plus simple d'optimiser la recherche lorsque la taille du voisinage recherché est plus petite. On note d_1 cette fonction de calcul de distance. Pour cette fonction, l'itération à laquelle est fait le test est choisie de manière optimale, en fonction du seuil de distance, en utilisant les résultats de la figure 2.25.

Pour aller plus loin, il faut envisager de tester la distance partielle non plus à une unique itération comme pour d_1 , mais à plusieurs itérations. Pour cela, pour plusieurs valeurs de nombres de dimensions testées (paramètre g), il faudrait tracer la fonction $P_T^L(n_1, n_2, \dots, n_g)$ où (n_1, n_2, \dots, n_g) correspondant aux dimensions après lesquelles est testée la distance partielle. Toutefois, ces résultats ne sont pas visualisables à cause du nombre de paramètres. Et plus il y a de tests de la distance partielle par rapport au seuil de distance, plus ces tests eux-mêmes sont coûteux en temps. Il faudrait alors les prendre en compte dans la fonction d'estimation du gain. Du fait du nombre de paramètres, il est impossible de faire une évaluation exhaustive de toutes les fonctions distances possibles. Nous pourrions nous restreindre à étudier les fonctions pour lesquelles le test de la distance partielle par rapport au seuil est effectué toutes les s itérations. Par exemple, pour $s = 2$, on testerait la distance partielle pour les itérations 2, 4, 6..., pour $s = 3$, on testerait aux itérations 3, 6, 9... et ainsi de suite. Toutefois, ce choix n'est pas adéquat car il est plus intéressant de faire de nombreux tests au début du calcul de la distance partielle, quand celle-ci augmente rapidement (voir figure 2.24). Inversement, pour les dernières itérations, il n'est pas nécessaire de faire de nombreux tests puisque la distance augmente très peu.

Au final, en se servant des tests des paragraphes précédents, et en essayant de varier les itérations auxquelles la distance partielle est testée, nous avons choisi manuellement une fonction distance qui est quasiment la plus rapide pour un seuil de 200 (c'est cette valeur que nous utilisons généralement dans les applications de recherche d'images). Cette fonction notée d_{opti} utilise les itérations 16, 40, 64, 96 pour tester la distance partielle. La table 2.26 montre les gains en temps de calcul mesurés en fonction du seuil choisi pour la recherche de voisins. On y voit que la fonction d_{opti} est nettement meilleure que la fonction qui n'utilise qu'une seule itération pour tester la distance partielle. Pour le seuil de 200 régulièrement utilisé par la suite, le gain est de 0.28. Dans cette table, le temps de référence pour la recherche linéaire est mesuré sur l'implémentation où la boucle est déroulée. Si nous prenons comme temps de référence l'implémentation initiale (i.e. simple boucle en laissant le compilateur optimiser), le gain avec la fonction d_{opti} est de 0.20 pour un seuil de 200.

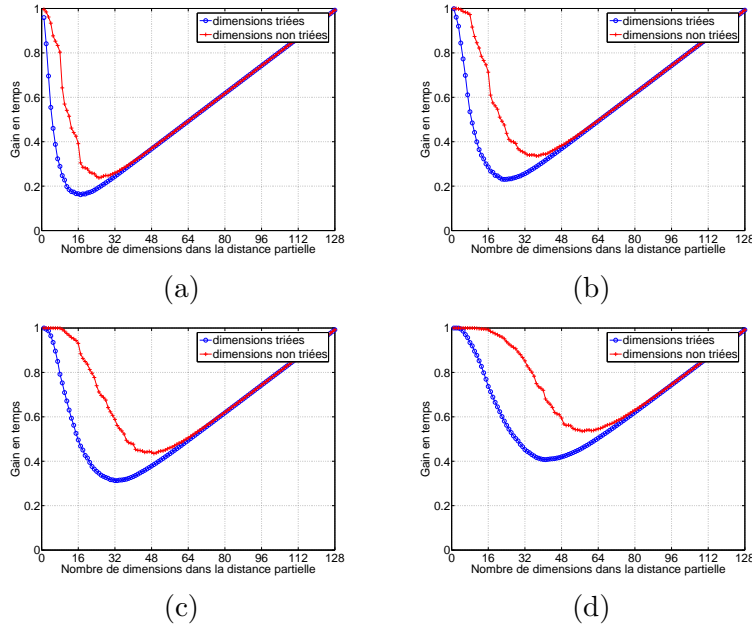


FIGURE 2.25 – Gain en temps en utilisant la distance partielle, pour un seuil de a :150, b :200, c :250 et d :300.

Seuil	q_1	d_{opti}
150	0.28	0.25
200	0.35	0.28
250	0.41	0.35
300	0.51	0.43

FIGURE 2.26 – Gain en temps avec l’utilisation de distances partielles. La fonction q_1 n’effectue qu’un seul test de la distance partielle alors que d_{opti} en effectue quatre.

Dans cette partie, nous avons étudié le gain que peut apporter l’utilisation d’une distance partielle. Par rapport à une implémentation initiale, le fait de dérouler manuellement la boucle de calcul et d’utiliser une distance partielle rend l’algorithme cinq fois plus rapide pour une recherche des voisins dans un rayon de 200 (correspondant à une transformation de difficulté moyenne, voir partie 2.5). Même si les optimisations proposées ici sont relativement simples, il nous a paru nécessaire de les présenter puisque le gain est important. Cela permettra de situer le gain pour les algorithmes d’indexation introduits dans les sections suivantes.

2.6.4 Implémentation de la méthode linéaire sur GPU

La recherche r-NN linéaire est un algorithme très simplement parallélisable. Il suffit en effet de séparer le nuage de points P en autant de sous nuages que

de processeurs disponibles. Dans un premier temps, nous avons appliqué cette méthode à des processeurs multi-coeurs. Nous avons testé l'algorithme sur un processeur Intel Dual-Core à 2.13 GHz. L'algorithme est alors deux fois plus rapide que la version exécutée sur un seul processeur. Toutefois, le parallélisme sur CPU sur des machines grand public est limité puisqu'à l'heure actuelle, les machines les plus puissantes ont seulement quatre processeurs. Nous pouvons aussi imaginer une exécution sur une carte mère contenant deux quad-core, c'est-à-dire au final huit exécutions en parallèle. D'autres options existent peut-être sur des serveurs spécifiques mais le prix de ces machines devient alors un frein pour les tests.

Une autre option consiste à exécuter l'algorithme sur les processeurs de la carte graphique. On parle alors de GPU (Graphics Processing Unit). Traditionnellement, les cartes graphiques ont pour but d'afficher des objets en 3D sur l'écran. Le pipeline général pour cet affichage est constitué de trois étapes principales :

1. Opérations sur les points à afficher (Vertex shaders). Il s'agit par exemple de projeter les sommets 3D d'une surface triangulée sur le plan image.
2. Rasterization. Cette étape transforme les objets géométriques 2D en ensemble de pixels à afficher. Par exemple, à partir d'un triangle 2D, on obtient la liste des pixels qu'il faut afficher pour visualiser ce triangle.
3. Opérations sur les pixels (Pixel shaders ou fragment shaders). Il faut décider quelle couleur attribuer à chacun des pixels à afficher.

Ce qui est intéressant est que la première et la dernière étape sont parallélisables. On peut en effet traiter les sommets d'une surface triangulée indépendamment des autres. Et c'est aussi vrai pour les pixels à afficher. Pour cette raison, les cartes graphiques ont donc naturellement évolué vers des architectures parallèles. De plus, la demande grandissante des jeux vidéo pour afficher un très grand nombre de triangles a poussé les constructeurs à développer des cartes avec un grand nombre de processeurs. La figure 2.27 montre l'évolution du maximum de GFLOPS (1 GFLOP = 1 milliard d'opérations en float : Giga Floating Point Operations Per Second) obtenus avec des architectures de cartes graphiques Nvidia ainsi qu'avec des CPU. On observe que depuis quelques années, les cartes graphiques sont devenues considérablement plus puissantes que les CPU.

Au début des cartes graphiques, le pipeline graphique était fixe. C'est-à-dire que le développeur disposait d'une librairie pour spécifier à la carte graphique le traitement attendu sur les points 3D et les pixels. Les possibilités étaient alors limitées et permettaient uniquement de faire de l'affichage simple. Par exemple, pour calculer la couleur d'un pixel appartenant à un triangle, l'utilisateur ne pouvait que spécifier le modèle d'illumination (e.g. il spécifie si l'on a une normale unique par triangle ou si l'on calcule une normale pour chaque pixel par interpolation des normales des trois sommets du triangle). Les constructeurs ont ensuite proposé le pipeline programmable. C'est-à-dire que les étapes de traitement de sommets ou de pixels peuvent être programmées. La programmation se faisait au début en un assembleur dédié aux GPU. Ensuite, plusieurs langages proches du C ont été proposés (e.g. Cg ou HLSL). Cette program-

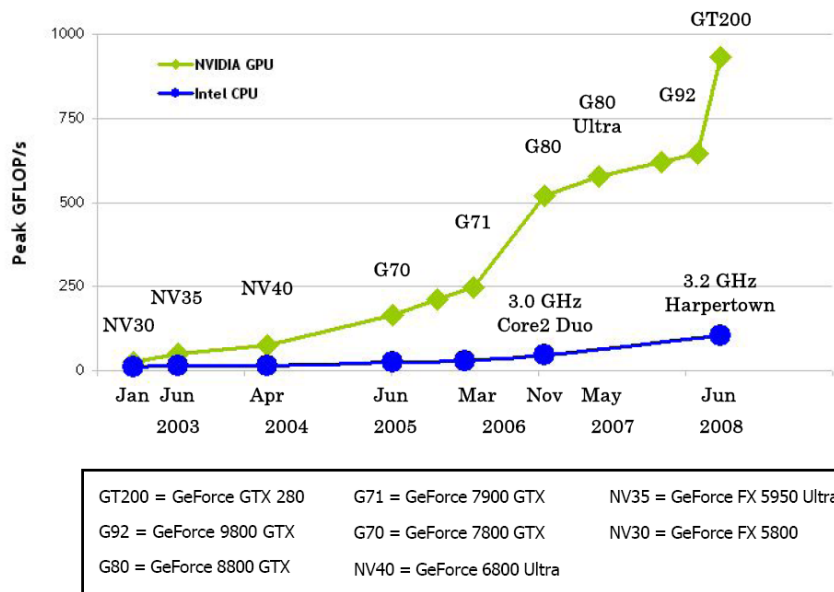


FIGURE 2.27 – Evolution des puissances des CPU et des GPU au cours des dernières années.

mation des shaders a surtout été utilisée pour obtenir des rendus variés. On peut citer par exemple des méthodes pour simuler un style cartoon, pour simuler un relief dans la texture uniquement par illumination (méthode de bump mapping consistant à déformer la normale). Toutefois, ce langage est encore très lié au pipeline d’affichage graphique. Récemment, les constructeurs ont proposé des langages permettant aux développeurs d’utiliser la puissance des cartes graphiques sans être liés au pipeline graphique. Nvidia propose le langage CUDA proche du C, ATI a un équivalent appelé CTM (Close To Machine) et l’université Stanford propose le langage BrookGPU. Le langage CUDA est bien documenté par Nvidia et par de nombreux sites Internet. Beaucoup d’exemples de programmes existent pour démarrer rapidement. Inversement, le langage de ATI semble peu utilisé pour l’instant et le constructeur propose également une version modifiée de Brook appelée Brook+. De même, le langage Brook dispose de peu de documentation. Comme ces langages sont proches du matériel, il nous a paru fondamental d’utiliser un langage bien documenté et largement utilisé. Nous nous sommes donc orienté vers le langage CUDA de Nvidia.

L’utilisation de la puissance des processeurs graphiques pour faire du calcul est un domaine qui est devenu très actif depuis quelques années. On trouve par exemple des implémentations du calcul des descripteurs SIFT sur GPU ou de suivi de points [SSG06], de reconstruction 3D à partir d’images [LKP06], de Graph-Cuts [VV08]...

2.6.5 Architecture des cartes Nvidia

Nous ne proposons pas ici une description exhaustive de l'architecture Nvidia mais seulement un résumé pour comprendre l'implémentation réalisée. Les processeurs scalaires (i.e. qui ne traitent qu'une donnée à la fois) des GPU sont regroupés par groupes de 8, appelés Streaming Multiprocessors et notés MP. Chaque MP gère donc 8 processeurs, ainsi qu'une unité de gestion d'instructions multithreading et une mémoire partagée entre les 8 processeurs. A plus haut niveau, la carte dispose d'une grille de MP. Les cartes les plus récentes disposent d'un plus grand nombre de MP (voir Annexe A de [NVI08]). Par exemple, la carte utilisée (Nvidia 8800 GT) dispose de 14 MP (et donc de $14 \times 8 = 112$ processeurs). Les derniers modèles Nvidia (GTX 280) disposent de 30 MP soit 240 processeurs. A noter que les processeurs scalaires sont souvent appelés stream processeurs sur les notices de cartes. On peut aussi coupler les cartes pour augmenter le nombre de processeurs. La configuration maximale actuelle est de coupler 4 cartes GTX 280 et ainsi de disposer de 960 processeurs. La fréquence des processeurs dépend de la carte mais change peu entre les modèles. Les processeurs de la 8800 GT fonctionnent à 1500 MHz et ceux de la GTX280 à 1296 MHz. On voit ici que le modèle le plus récent a une fréquence légèrement inférieure (mais il dispose de beaucoup plus de processeurs). Enfin, chaque carte dispose d'une mémoire globale (512Mo pour notre 8800 GT et jusqu'à 4Go pour certains modèles de GTX 280).

Pour implémenter un algorithme sur cette architecture, la première étape est de le paralléliser. Il faut ensuite décider comment répartir les threads sur les MP et spécifier combien de threads seront traités par chaque MP. On pourrait penser que 8 est adéquat car on a 8 processeurs par MP. En pratique, le MP va profiter des délais dûs aux accès mémoire pour exécuter les instructions de calcul d'autres threads. Il vaut donc mieux lancer beaucoup plus de threads par MP que 8. Il est aussi très important de bien gérer les accès mémoire qui sont très coûteux en temps. Un accès à la mémoire globale de la carte coûte entre 400 et 600 cycles d'horloge tandis qu'un accès à la mémoire partagée du MP coûte seulement 4 cycles. D'autre part, les accès à la mémoire globale doivent être ordonnés pour être réalisés en une seule opération. Par exemple, si les threads d'identifiants 0 à 31 ont besoin de lire des données à une plage d'adresses $[add, add + 31]$, il faut que le thread k lise la mémoire à l'adresse $add + k$. Si une autre option est choisie (par exemple, le thread k lit la mémoire à $add + 31 - k$), les accès mémoire ne seront pas regroupés et l'exécution sera beaucoup plus lente. Il faut aussi prêter attention à l'alignement des données. Par exemple, accéder à un élément d'un tableau de floats ne pose pas de problème alors qu'accéder aux éléments d'un tableau de "unsigned chars" est beaucoup plus lent car 3 valeurs sur 4 ne sont pas alignées sur 32 bits. Il est alors nécessaire de regrouper les "unsigned chars" par ensembles de 4 dans des structures adéquates.

Pour toutes ces raisons, la programmation GPU reste délicate pour obtenir les meilleures performances. Dans notre cas, l'implémentation développée prend en compte toutes les optimisations recommandées dans le guide de programmation CUDA [NVI08]. Toutefois, il est difficile d'assurer que cette implémentation

Méthode	temps (ms)	gain en temps
Linéaire simple (pas de déroulement de boucle)	21200	1.440
Linéaire simple, (déroulement complet de boucle)	14719	1
Distance Partielle, seuil 150	2589	0.176
Distance Partielle, seuil 200	3052	0.207
Distance Partielle, seuil 250	3899	0.265
Distance Partielle, seuil 300	5094	0.346
GPU	142	0.0096

FIGURE 2.28 – Gain en temps avec les méthodes de recherche linéaire. Pour calculer ces gains, le temps de référence est celui de la recherche linéaire simple, avec déroulement complet de la boucle (ligne 2 du tableau).

est optimale.

2.6.6 Résultats

Nous mesurons les performances des algorithmes de recherche linéaire r-NN sur une base de 9600 points et en lançant 2336 requêtes. Le CPU est un Pentium Dual-Core 2.13 Ghz (un seul coeur est utilisé) et le GPU est une carte Nvidia 8800 GT avec 512Mo de RAM. Les coordonnées des points sont des entiers positifs entre 0 et 255 codés sur 8bits (unsigned char). Nous avons choisi cette représentation car cela convient parfaitement aux points SIFT et optimise l'espace mémoire nécessaire. Lors de nos tests, le même algorithme adapté aux coordonnées codées en float sur 32 bits était 10% plus rapide environ.

Les résultats sont présentés dans le tableau 2.28. Les premières lignes reprennent les méthodes précédentes. Pour le gain en temps de calcul, nous prenons comme référence la méthode de recherche linéaire où la boucle des 128 itérations a été manuellement déroulée. En utilisant la distance partielle, nous retrouvons des gains similaires à ceux présentés dans les paragraphes précédents. Nous retrouvons le fait que le gain est dépendant du seuil choisi. Enfin, sur la dernière ligne du tableau, nous présentons les résultats obtenus avec notre implémentation sur GPU. Le gain par rapport à la méthode de référence est de 0.0096 soit un algorithme 104 fois plus rapide. Si l'algorithme de référence est celui où l'on n'a pas déroulé la boucle manuellement (i.e. première ligne du tableau), la version GPU est 149 fois plus rapide.

Une autre méthode très simple pour accélérer la recherche d'images par descripteurs locaux est évidemment de diminuer le nombre de descripteurs par image. Toutefois, pour faire cela, il faut s'assurer qu'en diminuant le nombre de descripteurs, nous conservons les performances en terme de recherche d'images. Dans le paragraphe suivant, nous présentons le protocole de test que nous utiliserons ensuite (en partie 2.8) pour vérifier qu'il est possible de diminuer largement le nombre de descripteurs par image sans affecter la qualité de la recherche. Ce protocole sera aussi réutilisé dans les chapitres suivants.

2.7 Protocole d'évaluation pour la recherche d'images

Dans un premier paragraphe, nous décrivons la provenance des images que nous utilisons. Ensuite, dans le paragraphe 2.7.2, nous présentons les déformations de ces images que nous utilisons pour générer des images similaires.

2.7.1 Bases d'images utilisées

Pour ces tests, nous utiliserons deux bases d'images afin de valider notre algorithme dans des situations variées. La première base notée DB_{4000} est constituée à partir de deux jeux d'images. Le premier jeu est construit à partir de 50 images variées téléchargées depuis Internet. Ces 50 images correspondent aux 50 requêtes utilisées pour tester l'algorithme. Nous appliquons à chacune de ces images 53 transformations (détaillées dans le paragraphe 2.7.2). Nous avons alors 2650 images (50×53) à retrouver dans notre jeu de données. Nous ajoutons 1350 images quelconques prises sur Internet pour perturber l'algorithme. Pour mesurer le rappel et la précision, nous effectuons les 50 requêtes, et pour chacune d'elles, nous comptons le nombre d'images retrouvées parmi les 53. Nous pouvons aussi mesurer le rappel et la précision pour chaque transformation. Nous aurions aussi pu ne mettre que les 50 images choisies dans la base et lancer les 2.650 requêtes (pour chacune d'elles, il faudrait alors retrouver quelle est son image d'origine). Toutefois, cette seconde approche est beaucoup plus longue (et fastidieuse pour tester divers paramètres) car il faut alors appeler 2.650 fois l'algorithme de recherche des plus proches voisins. Nous nous sommes donc concentrés sur la première méthode (i.e. 50 requêtes et 53 images à retrouver dans la base par requête).

La seconde base, notée DB_{32k} consiste en 32650 images. Elle contient les 2.650 images précédentes ainsi que 30.000 images de la base "holiday" de l'INRIA. L'objectif de cette base est de montrer que notre algorithme s'adapte à des bases de taille moyenne.

Au cours de notre thèse, nous avons aussi cherché à adapter notre algorithme à de très grandes bases. Pour cela, nous avons créé une base de 510.000 images. Ces images sont issues de plusieurs collections. La base holiday de l'INRIA de 100.000 images est incluse, ainsi que la base de Nister (voir 5.9). Les 2.650 images à retrouver dans nos tests sont aussi incluses. Le reste des images provient de divers sites Internet. Pour obtenir ces images, nous avons utilisé le logiciel Neodownloader [Neo] qui visite un grand nombre de sites Internet pour télécharger des images.

Pour les deux premières bases, nous commençons par mesurer les valeurs de rappel et précision ainsi que le temps de traitement en utilisant une recherche r-NN linéaire. Ensuite, nous cherchons comment ces mesures sont modifiées lorsque nous utilisons un algorithme r-NN approché.

Enfin, nous proposons une implémentation de notre algorithme dédiée aux très grandes bases, telles que celle de 510.000 images que nous utilisons.

2.7.2 Transformations d'images utilisées

Dans [LJA06], les auteurs utilisent le logiciel StirMark pour dégrader des images mais sans spécifier précisément les paramètres des déformations utilisées. Nous ne pouvons donc pas garantir de reproduire exactement les tests à l'identique. Ils utilisent aussi le logiciel ImageMagick pour augmenter ou diminuer le contraste des images. Dans [FS07], les auteurs reprennent la liste de transformations utilisées dans [KSH04] et en rajoutent quelques unes. Les transformations sont précisément expliquées et donc facilement reproductibles. Nous allons donc reprendre ces transformations. Toutefois, certaines sont trop faciles et ne présentent que peu d'intérêt (e.g. ajout d'une bordure de couleur autour de l'image). Au contraire, il manque certaines dégradations qui nous semblent importantes.

Nous nous sommes donc inspirés de ces articles pour produire une liste de transformations qui nous paraît plus complète. Il ne sera donc pas possible de comparer le rappel et la précision globalement avec ces articles. Il est toutefois possible de comparer nos résultats pour certaines des transformations qui sont identiques.

La liste des transformations est la suivante (nous donnons un nom unique à chaque transformation, et le nombre entre parenthèses est le nombre d'images transformées générées) :

1. COULEUR (3) : Augmenter la composante (a) rouge, (b) verte ou (c) bleue de 10%.
2. CONTRASTE (4) : (a) Augmenter ou (b) diminuer le contraste selon le paramètre par défaut d'ImageMagick, (c) augmenter (d) diminuer le contraste en utilisant trois fois le paramètre par défaut.
3. CROP (5) : Recadrer l'image en supprimant (a) 10%,(b) 25%, (c) 50%,(d) 75%, ou (e) 90% de l'image (en conservant la zone centrée autour du centre de l'image). L'image est ensuite redimensionnée à la taille de l'image initiale.
4. DESPECKLE (1) : utilisation de l'opérateur despeckle de ImageMagick avec les paramètres par défaut de ImageMagick (i.e. parfois appelé déparasitage). Cet opérateur effectue un filtre moyen qui préserve les contours. Pour un pixel donné, la moyenne et l'écart-type des intensités d'un voisinage sont calculés. Si pour ce pixel, son écart à la moyenne est supérieur à l'écart-type (multiplié par une constante), le pixel est considéré comme aberrant et son intensité est remplacée par la valeur moyenne.
5. GIF (1) : Convertir l'image d'entrée au format GIF. Cela compresse l'espace des couleurs sur 8 bits.
6. ROTATION (3) : Tourner l'image autour du centre de l'image de (a) 10°, (b) 45° ou (c) 90°.
7. REDIM (6) : Redimensionner l'image pour que sa taille soit multipliée par (a) 2, (b) 4, (c) 8 ou divisée par (d) 2, (e) 4, (g) 8 (en utilisant les paramètres de filtrage par défaut de ImageMagick).
8. SATURATION (5) : Changer la saturation de l'image par (a) 70%, (b) 80%,(c) 90%, (d) 110% ou (e) 120%.

9. INTENSITY (6) : Multiplier l'intensité de l'image par (a) 50%, (b) 80%,(c) 90%, (d) 110%,(e) 120%, (f) 150%.
10. NETETE (2) : Opérateur Sharpen de ImageMagick avec paramètre (a) 1 et (b) 3. Cet opérateur applique un filtre appelé "unsharp" ou "sharpening mask". Une image des hautes fréquences est calculée en soustrayant l'image initiale à une version lissée par une gaussienne de l'image. Ensuite, cette image des hautes fréquences est rajoutée à l'image initiale, ce qui renforce la "netteté" de celle-ci. Nous utilisons aussi les paramètres par défaut de ImageMagick.
11. EMBOSS (1) : Opérateur Emboss de ImageMagick, paramètre à 1. La documentation de ImageMagick ne permet pas de comprendre précisément cet opérateur. Nous pouvons juste dire qu'en général, un opérateur "emboss" est un opérateur qui pour chaque pixel, calcule une "inclinaison" (comme si l'image était une carte d'élévation ou l'altitude est l'intensité lumineuse) en utilisant le gradient local. Cette "inclinaison" est ensuite utilisée pour modifier l'intensité du pixel.
12. JPEG (2) : Fixer la qualité JPEG à (a) 80 et (b) 15.
13. MEDIAN (1) : filtre median 3x3.
14. ROTCROP (2) : Tourner l'image de (a) 10° et (b) 45° et recadrer pour ne conserver que 50% de l'image.
15. ROTSCALE (2) : Transformation ROTCROP et REDIM pour diviser la taille de l'image par 4.
16. INCLINAISON (4) : Incliner l'image selon l'axe x de (a) 5°, (b) selon l'axe x et y de 5°, (c) selon l'axe x de 15° et (d) selon les deux axes de 15°.
17. GAUSS (3) : Flou gaussien de rayon (a) 3, (b) 5 et (c) 7.
18. GAUSSNOISE (2) : Ajout de bruit gaussien d'écart-type (a) 10 et (b) 20.

La figure 2.1 montre quelques unes des images transformées parmi les plus difficiles.

2.8 Modification de l'étape de sélection des régions d'intérêt

Dans un problème de recherche des plus proches voisins r-NN, le temps d'exécution est proportionnel à la taille du nuage de points. Le moyen de plus simple d'accélérer les requêtes est donc de diminuer le nombre de points. C'est-à-dire qu'il faut vérifier si, lorsque l'on diminue le nombre de descripteurs par image, les images similaires sont toujours retrouvées.

Dans de nombreux articles, aucune limite n'est imposée et en moyenne, chaque image génère environ 1000 descripteurs SIFT. Toutefois, il nous paraît important de choisir le nombre de SIFT par image de manière optimale. En effet, si un trop grand nombre de descripteurs est détecté, l'algorithme de recherche des plus proches voisins en sera d'autant plus lent et gourmand en mémoire. Cela a par exemple un rôle important si l'algorithme de recherche sur GPU est utilisé (voir section 2.6.4), à cause de la ressource mémoire limitée.











 <p>Image originale</p>	 <p>2.(d) Diminution du contraste x3</p>
 <p>2.(c) Augmentation du contraste x3</p>	 <p>3.(e) Suppression de 90 % de l'image</p>
 <p>9.(a) Diminution de l'intensité de 50%</p>	 <p>11.(f) Augmentation de l'intensité de 150%</p>
 <p>7.(g) Division de la taille par 8</p>  <p>11.(a) Transformation Emboss</p>	 <p>16.(d) Inclinaison X et Y de 15° et 15°</p>  <p>13.(a) Application filtre median 3x3</p>

TABLE 2.1 – Exemples des transformations d'images utilisées.

2.8.1 Méthodes existantes

Foo et al. montrent dans [FS07] qu'il est possible de fortement diminuer le nombre de descripteurs à utiliser (e.g. en ne gardant que 10% des SIFT) tout en conservant quasiment les mêmes résultats en terme de rappel/précision sur un algorithme de recherche d'images. Leur méthode de sélection des SIFT est basée sur le contraste (expliqué en 2.2.3). Dans l'algorithme initial des SIFT, ce contraste n'est utilisé qu'avec un seuil pour garder ou non les points d'intérêt détectés. Dans leur méthode, les auteurs trient les régions d'intérêt selon ce contraste et ne conservent que les n régions qui ont le plus fort contraste. Mais ils ne comparent pas plusieurs méthodes de sélection. De plus, comme expliqué dans [LAJA06], les points correspondant aux basses fréquences ont un contraste moins élevé. En appliquant la sélection par contraste, la majorité de ces points sont supprimés. Or il s'avère que ces descripteurs sont très importants pour certaines déformations. Prenons l'exemple où nous appliquons un lissage gaussien à une image Im_1 pour obtenir une image Im_g . Dans l'image Im_1 , les descripteurs sélectionnés selon leur contraste seront ceux à haute fréquence. Inversement, dans l'image Im_g , comme le lissage a supprimé les hautes fréquences, le détecteur ne trouvera que des descripteurs correspondant aux basses fréquences. Au final, peu de descripteurs communs seront détectés sur ces deux images.

Nous pourrions utiliser la méthode par correction Gamma de [LAJA06] pour contrer ce problème. Elle consiste à ajouter une correction gamma lors de la création de la pyramide de gaussiennes dans l'étape de détection des SIFT. Les auteurs constatent en effet qu'avec cette variante, le contraste des SIFT détectés est plus homogène quel que soit le facteur d'échelle. Toutefois, nous préférons introduire une méthode de sélection qui ne modifie pas l'algorithme de détection des SIFT.

2.8.2 Méthode proposée

Dans ce paragraphe, nous présentons une contribution de nos travaux relativement à la sélection des régions d'intérêts pour la recherche d'images similaires.

Nous appelons n_{max} le nombre maximum de descripteurs par image. Pour choisir n_{max} SIFT parmi les N descripteurs d'une image, nous proposons simplement de garder les $\alpha \times n_{max}$ descripteurs avec le plus grand facteur d'échelle, et parmi les $N - \alpha \times n_{max}$ restant, le critère de contraste est utilisé pour n'en garder que $(1 - \alpha) \times n_{max}$ (i.e. les points ayant le plus fort contraste sont gardés). Le paramètre α est pris dans $[0, 1]$. Si $\alpha = 0$, cela revient à utiliser la méthode de sélection par contraste de [FS07]. Inversement, si $\alpha = 1$, nous ne gardons que les descripteurs qui ont le plus grand facteur d'échelle.

Cette méthode simple a l'avantage de répartir les descripteurs sélectionnés. C'est-à-dire que l'on va garder $\alpha \times n_{max}$ descripteurs détectés à grande échelle. Ceux-ci sont importants pour certaines images comme expliqué ci-dessus. Et nous conservons aussi $(1 - \alpha) \times n_{max}$ descripteurs qui ont un très fort contraste. Ceux-ci auront une forte probabilité d'être détectés après une modification de l'image.

Pour tester cette méthode de sélection des descripteurs pour la recherche

$\alpha \backslash n_{max}$	128	256	512	1024
0.	0.52/1	0.72/1	0.91/1	0.99/1
0.25	0.80/1	0.92/1	0.97/1	0.99/1
0.50	0.87/1	0.95/1	0.99/1	0.99/1
0.75	0.85/1	0.98/1	0.99/1	0.99/1
1.00	0.87/1	0.97/1	0.99/1	1.00/1

TABLE 2.2 – Couples rappel/precision obtenus en faisant varier les paramètres n_{max} entre 128 et 1024 et α entre 0.25 et 1.. La première ligne (avec $\alpha = 0$) correspond à la méthode de sélection par contraste de [FS07].

d’images similaires (de paramètres n_{max} et α), nous utilisons le protocole expérimental décrit en 2.7. Les résultats sont présentés dans le paragraphe suivant.

2.8.3 Evaluation

Pour chercher le paramètre α optimal, nous mesurons les couples rappel/précision obtenus pour plusieurs valeurs de n_{max} . En pratique, nous sélectionnons uniquement un sous-ensemble de 10 images requêtes qui sont difficiles (i.e. des images pour lesquelles certaines images transformées ne sont pas retrouvées) et nous mettons dans la base d’images uniquement les 10 transformations les plus difficiles de ces images (i.e. les transformations pour lesquelles l’algorithme échoue parfois). Les résultats sont présentés sur le tableau 2.2. Pour ces tests, nous avons utilisé un algorithme de recherche linéaire exact r-NN avec un seuil de distance r égal à 200.

La première remarque est que la précision est toujours de 1. Dans tous les tests, la méthode de sélection que nous proposons est meilleure que la sélection qui utilise uniquement le contraste (méthode de [FS07]). Nous pouvons aussi voir que le seul couple de paramètres qui atteint un rappel de 1 est $\alpha = 1$ et $n_{max} = 1024$. Pour $n_{max} = 128$, trop peu de descripteurs sont conservés et le rappel est faible. Au final, nous choisissons d’utiliser le couple $\alpha = 0.75$ et $n_{max} = 256$ pour plusieurs raisons. Le rappel obtenu de 0.98 est très proche de ceux obtenus lorsque l’on garde 2 ou 4 fois plus de points. Ce rappel de 0.98 signifie que seulement 2 images sur les 100 à retrouver ont été manquées. Il s’agit d’une image qui a subi un recadrage de 90% de sa surface ainsi que d’une image inclinée sur X et Y de 15°. Pour nos tests, il nous paraît acceptable que quelques exemples de ces transformations très difficiles soient manquées si en contrepartie, le gain en temps et occupation mémoire est important.

A noter que pour être plus efficace sur la transformation d’inclinaison, il faudrait changer de descripteur et utiliser un descripteur affine invariant. Comme ce n’est pas le but ici, nous acceptons de manquer quelques images de ce type. De plus, en choisissant seulement 256 descripteurs par image, cela permet de coder les index des descripteurs sur 8 bits. Cela est non négligeable en terme de gain d’occupation mémoire ainsi que pour optimiser certaines implémentations.

Pour confirmer ces résultats, nous appliquons la recherche d’images sur les

algorithme	temps recherche r-NN (ms)
CPU (sans déroulement de boucle)	241 898
CPU (déroulement de boucle complet)	167 418
CPU distance partielle	40 487
GPU	4 756

TABLE 2.3 – Temps nécessaire à la recherche des voisins des descripteurs par image pour une recherche exacte (une image contient 256 descripteurs).

50 requêtes, dans la base DB_{4k} . En sélectionnant 256 SIFT par image selon la méthode par contraste de [FS07], nous obtenons un rappel de 0.977 et une précision et 0.996. Avec notre méthode et $\alpha = 0.75$ et $n_{max} = 256$, nous obtenons un rappel de 0.994 et une précision de 0.997. Ce test confirme bien que notre méthode de sélection est meilleure que celle par contraste de [FS07].

Dans tous les tests réalisés dans les parties suivantes, sauf précision contraire, nous sélectionnons 256 SIFT par image en utilisation la méthode de sélection décrite ci-dessus avec $\alpha = 0.75$.

2.9 Résultats avec la recherche linéaire

Enfin, avant de conclure ce chapitre, nous mesurons les performances de l'algorithme SIFT+NN+AFF lorsqu'une recherche r-NN exacte est utilisée. Cette recherche nous sert de référence pour le rappel et la précision ainsi qu'en termes de temps d'exécution. Concernant le temps d'exécution, nous séparons d'une part la recherche des plus proches voisins et d'autre part l'étape de vérification affine. Concernant cette seconde étape, notre algorithme n'est pas optimisé et les temps sont seulement présents pour indication.

Sur DB_{4000} , le rappel obtenu avec la recherche exacte linéaire et un seuil de 200 est de 0.994 et la précision est de 0.997. Cela signifie que sur les 2650 images à retrouver lors des 50 requêtes, l'algorithme n'a manqué que 16 images. Il s'agit de 9 images déformées par inclinaison selon X et Y de 15° , 6 images dues à la transformation CROP90 et une image de la transformation EMBOSS.

Le temps de recherche des voisins par r-NN dépend de l'algorithme utilisé. Le tableau 2.3 montre les temps de recherche des voisins des descripteurs en fonction de l'algorithme. Ces temps sont donnés pour une image requête (contenant en moyenne 256 descripteurs). Le temps de vérification affine (i.e. l'étape de recherche de matrices affine par RANSAC dans l'algorithme SIFT+NN+AFF) est de 2100 ms. Il ne dépend pas de l'algorithme car cette étape prend en entrée le résultat de l'algorithme r-NN (ce sont tous ici des algorithmes r-NN exacts). L'implémentation sur GPU est toujours la plus rapide. Toutefois le gain sur la méthode de référence CPU (avec déroulement de boucle complet) n'est que de 0.029, ce qui est plus faible que lors des tests de la partie 2.6.4. Cela vient du fait que notre implémentation est plus adaptée pour chercher les voisins d'un grand nombre de descripteurs simultanément. Or dans nos tests, nous procédons image requête par image requête et chacune d'elles contient au maximum 256 descripteurs.

2.10 Conclusion

L'objectif de ce chapitre était d'abord d'introduire toutes les briques de la recherche d'images similaires par descripteurs locaux et de proposer une méthode pour analyser automatiquement une image de linéaires de supermarché. Nous souhaitons aussi établir des premiers éléments de performances en temps, pour servir de références pour les prochains chapitres dans lesquels nous chercherons à accélérer ces applications.

Dans une première partie, nous avons présenté l'algorithme SIFT+NN+AFF de D. Lowe [Low04] que nous avons utilisé pour chercher des images similaires dans une base. Nous avons ajouté une extension à cet algorithme pour traiter des images de linéaires de supermarché (Auclair et al. [ACV07a]).

Dans ces deux applications (i.e. recherche d'images similaires et analyse de linéaires), l'algorithme r-NN de recherche de voisins SIFT est l'étape qui prend le plus de temps. Nous en avons étudié la version la plus simple : la recherche linéaire. En analysant les données des vecteurs SIFT, nous avons proposé une heuristique (l'utilisation de distances partielles) permettant un gain important (i.e. 5 fois plus rapide qu'une implémentation standard de la recherche linéaire). Ensuite, nous avons mesuré les performances de la recherche linéaire (sans distance partielle) sur GPU. Avec notre implémentation, nous obtenons un algorithme jusqu'à 100 fois plus rapide qu'un algorithme linéaire sur CPU.

Notre conclusion est que pour la recherche linéaire, la meilleure option est d'utiliser le GPU. L'architecture des cartes graphiques se prête tout à fait à la parallélisation de cet algorithme et les gains obtenus sont très importants. Pour conclure cette étude, il serait intéressant de tester l'utilisation de distances partielles dans cet algorithme sur GPU. Il faudrait également tester les bibliothèques optimisées de multiplication matricielle de type BLAS pour effectuer le calcul de distances euclidiennes entre nuages de points.

Pour accélérer l'algorithme de recherche d'images similaires, nous avons aussi introduit une méthode qui réduit le nombre de descripteurs SIFT par image tout en conservant un rappel quasi identique. Cette méthode privilégie les points détectés à l'échelle la plus large et parmi les points à plus petite échelle, elle conserve ceux ayant le plus fort contraste (selon le sens décrit en 2.2.3). Dans nos tests, cette méthode obtient de meilleurs scores que la méthode de sélection par contraste de Foo et al. [FS07].

Il faut noter que dans cette partie, les nuages de points utilisés étaient relativement petits. Par exemple, notre implémentation GPU requiert 142ms pour chercher les voisins de 2336 points dans un nuage de 9600 points. Dans le cas de la recherche d'images dans des grandes bases, la taille des nuages de points sera beaucoup plus grande. Comme l'algorithme utilisé jusqu'ici est linéaire en fonction du nombre de points, les temps d'exécution deviendront rapidement très importants. C'est-à-dire que pour traiter des très grands nuages de SIFT, la recherche linéaire ne sera pas adaptée. Dans le chapitre suivant, nous allons donc nous intéresser aux structures de données qui permettent d'indexer efficacement les vecteurs SIFT pour déterminer quelles méthodes permettent

d'être plus rapides qu'une implémentation GPU de la recherche linéaire.

Chapitre 3

Etat de l'art de la recherche des plus proches voisins

Nous avons vu dans le chapitre précédent que la recherche linéaire r-NN ne permettait pas de traiter très rapidement des grands nuages de points. Dans ce chapitre, nous présentons d'autres algorithmes r-NN de la littérature qui cherchent à diminuer le temps de recherche. Toutefois, du fait du grand nombre de dimensions de l'espace des SIFT, ces algorithmes exacts n'apportent que peu de gain par rapport à la recherche linéaire (ils sont confrontés à la “malédiction de la dimension”, voir 2.1.2). Nous allons donc également présenter les algorithmes r-NN approchés. Ces algorithmes acceptent de manquer des points voisins de la requête, mais en contrepartie, ils sont beaucoup plus rapides.

Enfin, nous présenterons les limitations des algorithmes actuels et ce que nous chercherons à améliorer dans les chapitres suivants.

Sommaire

3.1	Introduction	69
3.2	Méthodes exactes à base d'arbres	69
3.3	Autres méthodes exactes	73
3.4	Méthodes approchées à base d'arbres	73
3.5	Méthodes approchées à base de hachage	74
3.6	Autres méthodes approchées	80
3.7	Diminution de la dimension de l'espace	82
3.8	Taille des bases de données dans la littérature	83
3.9	Méthodes d'évaluation de la recherche r-NN approchée	83
3.10	Conclusion	86

3.1 Introduction

Dans les parties suivantes, nous présentons un état de l'art des algorithmes de recherche des plus proches voisins. L'article [BBK01] constitue un état de l'art de ce problème que nous complétons ici avec d'autres méthodes. Nous regroupons les algorithmes en deux grandes classes : ceux exacts et ceux approchés. Les algorithmes exacts retournent tous les points dont la distance avec le point requête est inférieure au seuil r . Inversement, les algorithmes approchés vont manquer certains points qui sont des voisins. Pour les algorithmes exacts, nous créons deux sous-classes comprenant ceux à base d'arbres et les autres approches. Pour les algorithmes approchés, nous créons trois sous-classes, respectivement : ceux à base d'arbres, ceux à base de hachage et les autres. C'est à dire que nous avons divisé notre état de l'art comme suit :

- Algorithmes r-NN exacts
 - A base d'arbres
 - Autres
- Algorithmes r-NN approchés
 - A base d'arbres
 - Méthodes de hachage
 - Autres

3.2 Méthodes exactes à base d'arbres

Les arbres sont utilisés pour diviser de manière hiérarchique l'espace étudié ou le nuage de points sur lequel les recherches sont effectuées. Pour un arbre qui divise l'espace étudié, chaque noeud divise une région en sous-régions. L'intersection de ces sous-régions est en général vide mais certains algorithmes utilisent des sous-régions ayant un léger recouvrement. Un arbre est couplé à un algorithme de parcours qui spécifie comment trouver les voisins d'un point requête en utilisant au mieux la structure. Le principe est similaire pour les arbres qui divisent le nuage de points étudié.

Pour chaque arbre, on distingue trois étapes :

- Création de l'arbre
- Ajout d'un point dans l'arbre
- Recherche des voisins d'un point requête

3.2.1 Division de l'espace par un $2^d - Tree$

En 2D, les arbres de division de l'espace les plus connus sont les QuadTree. Chaque noeud de l'arbre représente la division d'une région en quatre sous-régions rectangulaires. Le noeud supérieur divise tout l'espace en quatre sous-régions. Chaque sous-région est subdivisée en quatre par un nouveau noeud, et ainsi de suite. L'équivalent en 3D est l'OctTree. Dans un espace à d dimensions, le même principe peut tout à fait être appliqué (l'arbre est alors un $2^d - Tree$). Il existe plusieurs variantes de création de $2^d - Tree$, mais ce type d'arbre n'est

généralement pas utilisé pour les algorithmes de recherche des plus proches voisins. D'autres arbres ont été développés plus spécifiquement pour ce problème.

3.2.2 R-Tree

Cet arbre (fig 3.1) est introduit dans [Gut84]. Chaque noeud est une liste de n boîtes englobantes de points (aussi appelées MBR pour Minimum Bounding Rectangle). Chacune de ces boîtes englobantes pointe vers ses noeuds enfants. Ses enfants sont des boîtes englobantes plus petites qu'elle contient.

Création Pour créer l'arbre, on y insère les points du nuage un par un. Quand un point est ajouté dans l'arbre, on le place dans la feuille pour laquelle il faudra le moins agrandir sa boîte englobante. Un paramètre est le nombre de points maximal dans une feuille. Dès qu'une feuille est pleine, elle est divisée en deux rectangles. Il existe différentes versions pour savoir comment diviser une feuille lorsque celle-ci est pleine. Par exemple, pour les R^*Tree , introduits dans [BKSS90], les auteurs essaient d'éviter au maximum que les boîtes englobantes ne s'intersectent. Les R^+Tree [SRF87] sont des $RTree$ pour lesquels les boîtes englobantes ne s'intersectent pas. On peut aussi citer comme variante les TV-Tree [LJF94] ou bien encore les X-Tree [BKK96].

Recherche des voisins d'un point requête Quand on fait une recherche, il faut décider quels rectangles englobants peuvent contenir des voisins et les parcourir ([BBK01]). Pour cela, on commence par le noeud racine et l'on regarde quels sous-rectangles intersectent la boule de centre le point requête q et de rayon r (notée $B(q, r)$). Pour chaque sous-rectangle où l'intersection est non vide, on réitère ce processus jusqu'aux feuilles. Pour une feuille, si son rectangle englobant est entièrement inclus dans $B(q, r)$, tous les points sont voisins, sinon, pour chacun des points qu'elle contient, sa distance euclidienne avec q est calculée pour décider si c'est un voisin de q ou non.

3.2.3 SS-Tree : Similarity Search Tree

Ces arbres introduits dans [WJ96] sont similaires aux R-Tree mais utilisent des sphères englobantes au lieu de rectangles englobants.

Construction Lors de la construction, un nouveau point est placé dans la feuille dont la sphère a le plus proche centre. La figure 3.2 illustre le principe du SS-Tree. Comme pour le R-Tree, un paramètre important est le nombre maximal de points par feuille.

Recherche des voisins d'un point requête La recherche est identique que pour le R-Tree. On peut aussi citer les SR-Tree [KS97] qui utilisent à la fois des rectangles englobants et des sphères englobantes. Les principes de construction et de recherche sont également très similaires.

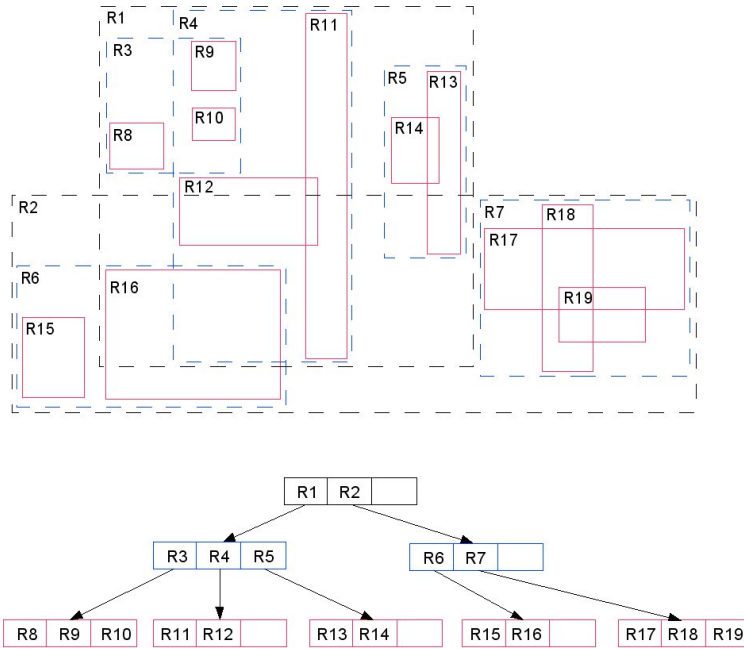


FIGURE 3.1 – Exemple de R-Tree.

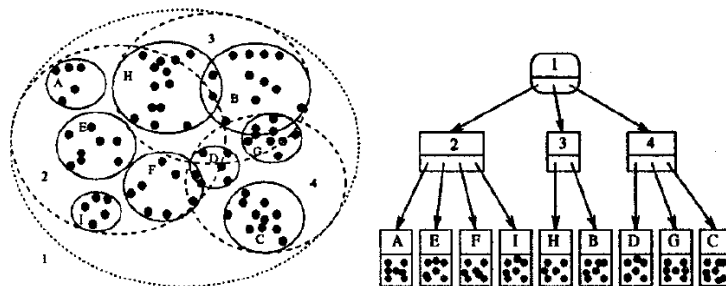


FIGURE 3.2 – Exemple de SS-Tree.

3.2.4 Kd-tree

Introduits dans [Ben75], ces arbres binaires divisent l'espace de manière hiérarchique.

Construction Initialement, tous les points sont mis dans le noeud racine. Chaque noeud interne divise son nuage de points en deux ensembles. Cette division se fait en choisissant un hyperplan, et en divisant le nuage entre les points d'un côté de cet hyperplan et ceux de l'autre côté. Cela revient à projeter les points sur l'un des axes principaux de l'espace et à seuiller selon cet axe. En général, on choisit la dimension sur laquelle on a la plus forte variance pour diviser les points. Comme seuil (i.e. position de l'hyperplan), on choisit la valeur médiane des points projetés sur l'axe choisi. Ce processus itératif de subdivision s'arrête quand toutes les feuilles contiennent un nombre de points inférieur à un seuil.

Recherche des voisins d'un point requête Pour rechercher les plus proches voisins d'un point requête q , on commence par faire une recherche en privilégiant la profondeur (depth-first). A chaque noeud, on vérifie de quel côté de l'hyperplan séparateur se trouve q et on continue le parcours dans le noeud enfant associé. Enfin, les points de la feuille atteinte sont sélectionnés. Ensuite il faut faire une phase de parcours inverse (back-tracking). On remonte noeud après noeud et l'on décide si l'on doit parcourir ou non les branches non visitées. Pour cela, à chaque noeud visité, il faut regarder si la branche non visitée peut contenir des voisins (dans le cas r-NN) ou des voisins plus proches (dans le cas k-NN). Si c'est le cas, on visite ces branches. Cet algorithme de recherche est surtout adapté pour le cas $k-NN$. En effet, avec une recherche depth-first, on a plus de chance de trouver les plus proches voisins au début, et donc de limiter le nombre de branches à visiter ensuite. Pour une recherche r-NN, on pourrait utiliser une recherche identique à celle utilisée pour les $R-Tree$.

3.2.5 Metric-Tree

Ces arbres sont des arbres binaires qui décrivent un nuage de points de manière hiérarchique, en utilisant une métrique. Un Kd-Tree peut être vu comme un cas particulier de Metric-Tree.

construction Comme pour un Kd-Tree, chaque noeud représente un nuage de points. Le noeud racine représente tous les points du nuage. Les points d'un noeud interne sont partagés en deux sous-ensembles représentés par ses deux enfants. La division du nuage de points d'un noeud en deux sous-ensembles varie selon le type de Metric-Tree. Par exemple, une possibilité de division choisit deux points parmi le nuage. Les points du nuage plus proches du premier point sont attribués au premier noeud enfant, les points plus proches du second point sont attribués au second noeud enfant (dans ce cas précis, l'arbre résultant est un Kd-Tree). On peut aussi choisir un unique point du nuage. On divise alors les points entre ceux dont la distance à ce point est inférieure à un seuil, et les

autres (arbres dits VP-Tree, [Yia93]). On trouve d'autres variantes dans [Uhl91], [CPZ97]. Dans [LMGY04], les auteurs présentent une version d'un Metric-Tree où les recouvrements sont autorisés (arbres appelés Spill-Tree).

Recherche des voisins d'un point requête L'algorithme de recherche est identique à celui d'un Kd-Tree.

3.3 Autres méthodes exactes

Une autre classe d'algorithme exact est bien sûr celle de la recherche linéaire. Cette méthode et ses variantes ont été présentées dans la partie 2.6 du chapitre précédent. Nous présentons seulement ici une approche appelée VA-file.

VA-file Dans Weber et al., [WSB98], les auteurs mesurent les performances des arbres de type R-Tree et X-Tree. Leur conclusion est qu'à partir d'une dimension égale à 10, ces structures n'apportent aucun gain ou sont moins performantes qu'une recherche linéaire. Ils introduisent aussi une structure appelée VA-File (VA pour Vector Approximation) qui vise à accélérer la recherche linéaire. Chaque dimension i est divisée en 2^{b_i} intervalles qui contiennent le même nombre de points. Cela revient à diviser l'espace en 2^n hyper-rectangles (i.e. $n = \sum_{i=1}^d b_i$). Chaque hyper-rectangle a pour identifiant une chaîne de n bits correspondant à sa position. Chaque point du nuage est approché par la chaîne de bits de l'hyper-rectangle qui le contient. Lors de la recherche des voisins, les frontières des hyper-rectangles permettent de savoir s'il faut calculer précisément la distance entre le point requête et le ou les points contenus dans cette zone ou bien si l'on peut exclure ce bloc de la recherche.

Les performances exprimées dans [WSB98] montrent que cet algorithme très simple est plus performant que les arbres classiques pour un nombre de dimensions supérieur à 10 et sur certaines données.

3.4 Méthodes approchées à base d'arbres

Les algorithmes exacts présentés ci-dessus n'apportent pas un gain suffisant pour traiter des nuages de très grande taille dans un espace à 128 dimensions tel que celui des SIFT. Dans les paragraphes suivants, nous présentons des algorithmes et des structures de données qui permettent de faire des recherches r-NN approchées. Ces recherches renvoient un sous-ensemble des voisins du point requête, mais peuvent être beaucoup plus rapides que les algorithmes exacts. Dans cette partie, nous nous attardons sur les méthodes approchées qui utilisent un arbre comme structure de données. Dans la partie 3.5, nous étudions les méthodes à base de hachage et enfin, dans la partie 3.6 nous présentons d'autres méthodes qui ne rentrent pas dans ces deux premières catégories.

3.4.1 Recherche partielle dans un arbre

Pour chacun des arbres présentés dans la partie précédente, l'algorithme de recherche que nous avons décrit était un algorithme qui trouvait tous les voisins du point requête. Une méthode simple pour accélérer la recherche est de la stopper lorsque l'on pense avoir visité un nombre de branches de l'arbre suffisant pour avoir trouvé une bonne proportion des voisins. Ce concept est applicable à la plupart des arbres. Toutefois, dans la littérature, nous ne l'avons trouvé que pour l'arbre Kd-Tree. Nous appelons cet algorithme de recherche partielle sur un Kd-Tree une recherche Best-Bin-First (nom donné dans les travaux de D. Lowe qui l'utilise pour des SIFT [Low04]).

3.4.2 Algorithme Best-Bin-First sur Kd-Tree

Dans cet algorithme, les points du nuage sont d'abord utilisés pour construire un Kd-Tree tout à fait classique. Ensuite, l'étape de recherche des voisins $k - NN$ est modifiée.

Dans les travaux de D. Lowe [Low04], l'auteur constate que la majeure partie du temps de recherche des voisins est due à l'étape de backtracking et qu'après quelques branches revisitées, cette étape n'apporte que peu de nouveaux voisins. Ils proposent alors de limiter le nombre de branches visitées et de ne parcourir que les branches qui ont le plus de chance de contenir des voisins. Pour choisir les branches à visiter, une liste des noeuds déjà visités est conservée en mémoire et triée selon la distance entre le point requête q et l'hyperplan séparateur pour le noeud en question. L'algorithme va d'abord parcourir les branches que l'on avait évitées pour lesquelles ce choix était le moins légitime (le point q était proche de l'hyperplan). Une fois le nombre de branches maximales atteint (c'est un paramètre de l'algorithme), l'algorithme s'arrête. Il s'agit là d'un algorithme approximatif car certains voisins sont manqués, mais en contrepartie l'algorithme est plus rapide. Cet algorithme de parcours avait été introduit initialement dans [BL97]. Nous l'appellerons l'algorithme Best-Bin-First. Cet algorithme est utilisé dans [Low04] pour faire des recherches $k - NN$ avec $k = 2$. Mais il est tout à fait possible de l'utiliser à l'identique pour faire des recherches r-NN.

3.5 Méthodes approchées à base de hachage

Le hachage est une approche qui permet d'indexer des données selon des clés obtenues par des fonctions de hachage. Prenons l'exemple d'un hachage qui indexe des chaînes de caractères. Une fonction de hachage très simple est une fonction qui retourne la première lettre de la chaîne de caractères (i.e. la clé de hachage). Une chaîne de caractères sera alors enregistrée dans une table, à la case correspondant à sa première lettre. Pour stocker plusieurs chaînes dans une case, une table de hachage est généralement une table de listes chaînées. La figure 3.3 résume ce principe.

Pour savoir si une table de hachage contient un élément q , la première étape est de calculer sa clé de hachage. Ensuite, les éléments x^i enregistrés dans la

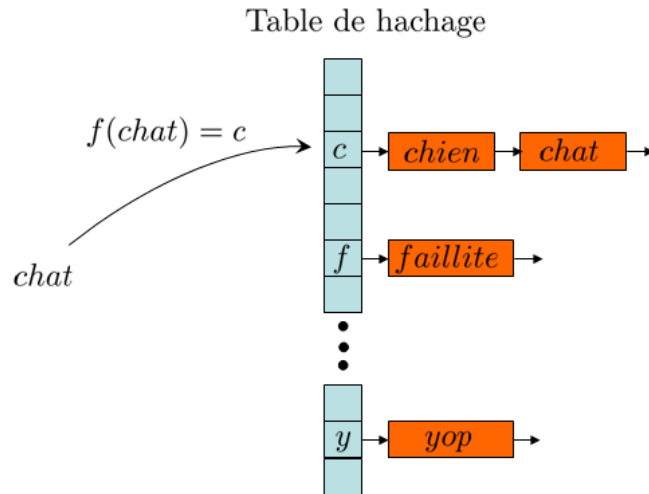


FIGURE 3.3 – Exemple simple de hachage. L’espace de départ de la fonction de hachage f est l’ensemble des chaînes de caractères. L’espace d’arrivée est l’intervalle $[0, 255]$ en considérant qu’un caractère est représenté par un entier entre 0 et 255.

case de la table correspondant à cette clé sont testés de manière exhaustive. Par exemple, pour les chaînes de caractères, on appliquera l’opérateur égalité des chaînes de caractères. Si la table de hachage indexe des points pour un algorithme r-NN, on va plutôt calculer la distance euclidienne entre les points x^i de la case et le point requête pour tester si ce sont des voisins.

En pratique, la fonction de hachage utilisée introduit des collisions (i.e. deux éléments différents obtenant la même clé de hachage). Il est alors être utile d’utiliser un moyen pour détecter rapidement ces collisions (on parle parfois de “checksum”). Par exemple, pour une chaîne de caractères, on pourra simplement enregistrer sa taille dans la table. Lors d’une requête, on n’appliquera l’opérateur égalité sur les chaînes ayant la même clé de hachage que la chaîne requête et ayant aussi la même taille (voir exemple de la figure 3.4).

Dans le cadre de la recherche r-NN, une fonction de hachage doit hacher les points proches avec la même clé et des points éloignés avec des clés différentes. Dans les paragraphes suivants, nous présentons le hachage LSH qui utilise cette méthode.

3.5.1 Locality-Sensitive Hashing

Cette classe de méthodes, notée LSH, basée sur le hachage est apparue dans Indyk et al. [IM98]. Une fonction de hachage est dite “locality sensitive” si deux points proches obtiennent la même clé avec une forte probabilité et que deux points éloignés obtiennent la même clé avec une faible probabilité. Plus formellement, une famille de fonctions $\mathcal{H} \{h : S \mapsto U\}$ est sensible de type

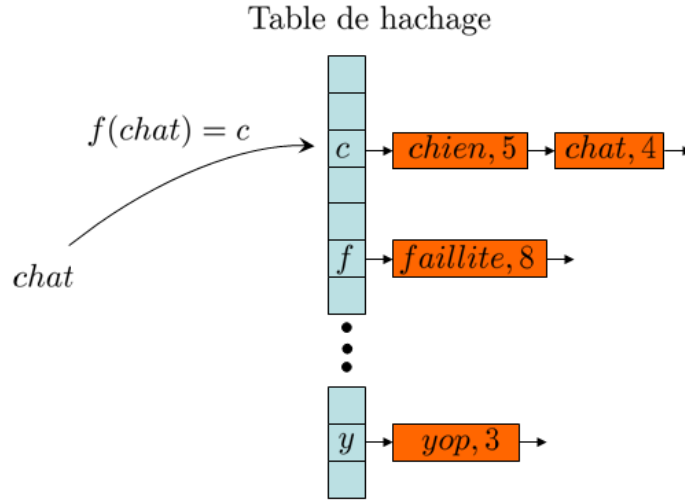


FIGURE 3.4 – Même exemple que sur la figure 3.3 avec détection de collisions. Pour chaque chaîne de caractères enregistrée dans la table, on enregistre aussi sa taille.

(r_1, r_2, p_1, p_2) avec $r_1 < r_2$ et $p_1 > p_2$ si on a les propriétés suivantes :

1. $\forall p \in B(q, r_1)$, alors $Pr_{h \in \mathcal{H}} [h(q) = h(p)] \geq p_1$
2. $\forall p \notin B(q, r_2)$, alors $Pr_{h \in \mathcal{H}} [h(q) = h(p)] \leq p_2$

où $B(q, r)$ est la boule de centre q et de rayon r .

3.5.2 Hachage LSH pour la distance de Hamming

Dans Gionis et al. [GIM99], les auteurs présentent une famille de fonctions LSH pour la distance de Hamming (nous nommerons cet algorithme *LSH1*). Cette famille est une amélioration de celle proposée dans les travaux de Indyk et al. [IM98]. On émet l'hypothèse que les coordonnées des points sont des entiers positifs. On note g_i les l fonctions de hachage de la famille proposée. Cela signifie que les points seront stockés dans l tables de hachage différentes. L'algorithme est paramétré par le nombre de dimensions qui seront hachées, noté k . Chaque fonction g_i est paramétrée par deux vecteurs :

$$D_i = \langle D_0^i, D_1^i, \dots, D_{k-1}^i \rangle \quad (3.1)$$

et

$$T_i = \langle t_0^i, t_1^i, \dots, t_{k-1}^i \rangle \quad (3.2)$$

Les coordonnées de D_i sont choisies de manière aléatoire dans $[0, d-1]$ où d est le nombre de dimensions de l'espace. Les coordonnées de T_i sont des seuils, choisis aléatoirement dans l'intervalle $[0, C]$, où C est la plus large coordonnée de tous les points du nuage. Chaque fonction g_i projette un point p de $[0, C]^d$ dans $[0, 2^k - 1]$ de telle manière que $g_i(p)$ est calculé comme une chaîne de k bits notés $b_0^i, b_1^i, \dots, b_{k-1}^i$ tels que :

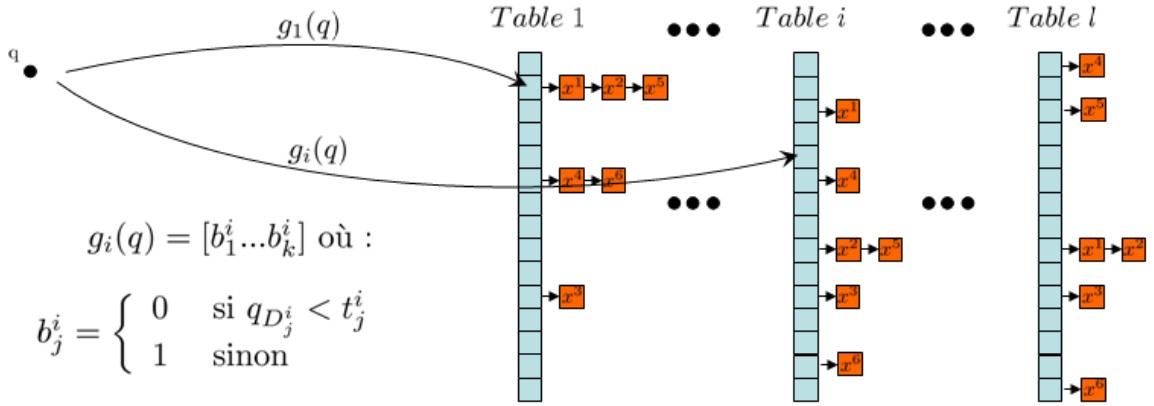


FIGURE 3.5 – Illustration du LSH1.

$$b_j^i = 0 \text{ si } (p_{D_j^i} < t_j^i) \text{ sinon } 1. \quad (3.3)$$

où l'on a noté $p_{D_j^i}$ la coordonnée de p selon la dimension D_j^i . La chaîne de k bits est la clé du point dans la i^{eme} table de hachage. Pour chercher les voisins d'un point requête q , on calcule sa clé de hachage pour chacune des tables. Ensuite, on applique une recherche linéaire sur les points des cases correspondantes. Les paramètres l et k permettent de choisir entre vitesse et précision. Ce principe est repris sur la figure 3.5.

Comme k peut être relativement élevé (e.g. $k = 32$), l'espace de destination des fonctions de hachage peut être trop grand pour tenir en mémoire. On rajoute une seconde fonction de hachage pour projeter le résultat des fonctions g_i sur un domaine plus petit ([SDI06a]). Toutefois, cela peut créer artificiellement des collisions. Pour détecter ces collisions, une somme de contrôle (aussi appelée checksum) est calculée sur chaque chaîne de k bits. Ce checksum permet d'identifier simplement de nombreux cas où deux chaînes de k bits sont différentes. Lors du parcours linéaire des points d'une case, on ne calcule la distance que si ce checksum est identique pour le point en cours et le point requête. Cet algorithme a par exemple été utilisé avec succès dans [KSH04].

3.5.3 Hachage LSH pour la distance euclidienne

On trouve dans [DIIM04],[SDI06b] une famille de fonctions LSH plus récente et adaptée à la norme euclidienne (que nous nommerons $LSH2$). On note $\mathcal{H} = \{h_i\}_{i \in 1..m}$ cette famille. Un point p de l'espace est haché par une fonction de hachage :

$$h_{a_i, b_i}(p) = \left\lfloor \frac{\langle p | a_i \rangle - b_i}{w} \right\rfloor \quad (3.4)$$

où a_i est un vecteur de norme unitaire, w est la quantification choisie en fonction des données et b_i est un décalage choisi de manière aléatoire dans $[0, w)$. Chaque fonction est paramétrée par un couple (a_i, b_i) . L'opération $\langle x|a_i \rangle$ projette le point p sur la direction i . Le résultat de la fraction est arrondi pour aboutir à une valeur entière. Chaque projection divise l'espace selon des plans orthogonaux au vecteur a_i . Dans [DIIM04], les auteurs montrent que la famille \mathcal{H} est LSH lorsque les coordonnées des a_i sont choisies aléatoirement avec une distribution p -stable (voir [DIIM04]). Pour que la famille soit LSH selon la norme L_2 , on peut tirer les coordonnées des a_i selon une distribution normale (gaussienne) d'expression :

$$g(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \quad (3.5)$$

De la même façon que pour le LSH adapté à la norme L_1 , on utilise l fonctions de hachage. Chaque fonction de hachage g_j est une concaténation de k fonctions issues de \mathcal{H} :

$$g_j = (h_{a_1, b_1}^j, h_{a_2, b_2}^j, \dots, h_{a_k, b_k}^j) \quad (3.6)$$

On peut alors indexer un point selon l vecteurs de k entiers. Au final, cela revient à diviser l'espace selon une grille non orthogonale (voir figure 3.6) et un vecteur de k entiers indique une case de cette grille. Pour calculer une clé de hachage sur $[0, c[$ à partir d'un vecteur de k entiers, une autre fonction de hachage classique u est utilisée [SDI06b]. Pour un point $x = (x_1, \dots, x_k)$, on choisit :

$$u(x) = \left(\left(\sum_{i=1}^k r_i x_i \right) \text{mod} P \right) \text{mod} c \quad (3.7)$$

où P est un nombre premier (e.g. $P = 2^{32} - 5$) et les r_i sont des entiers aléatoires. Le résultat de $u_i(p)$ est la clé de hachage du point p dans la table i . De la même façon que pour le LSH L_1 , si c est choisi petit, on aura des collisions non souhaitées. On peut alors utiliser la même méthode de checksum.

Enfin, l'algorithme de recherche est identique à celui du LSH L_1 . On calcule les clés de hachage du point requête. Ensuite, on applique une recherche linéaire sur les points contenus dans ces cases. Cet algorithme a par exemple été utilisé par Chum et al. [CPIZ07] pour faire de la recherche d'images similaires en utilisant des histogrammes hiérarchiques de couleurs.

A noter que chaque table de hachage LSH peut-être vue comme un partitionnement aléatoire de l'espace. La clé de hachage obtenue pour un point est l'index de la partition à laquelle il appartient. Cette idée de partitionnement aléatoire a aussi été reprise dans les forêts aléatoires [MTJ07]. Ces forêts sont composées d'arbres binaires, où chaque noeud divise l'espace en deux sous-ensembles. La décision de chaque noeud est basée sur un seuil aléatoire, selon une dimension choisie aléatoirement. La différence avec le LSH est que l'on réitère ce tirage aléatoire tant que la partition du noeud en cours de création ne satisfait pas un certain score qui évalue sa qualité.

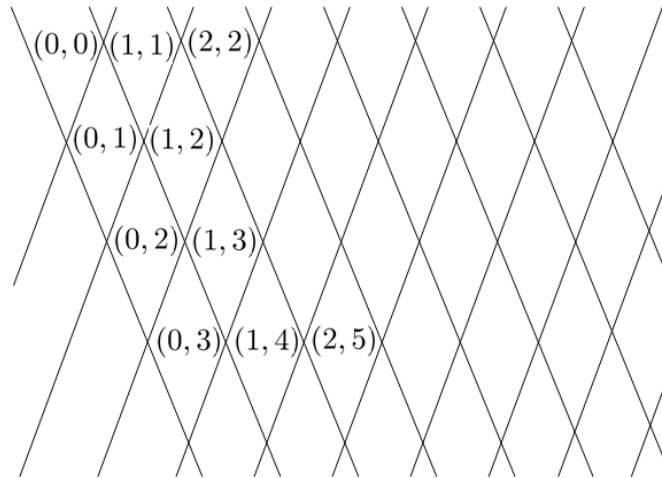


FIGURE 3.6 – Illustration de la segmentation de l’espace selon des projections non orthogonales telle que générées par le LSH. Chaque case de la grille est indexée par un vecteur de k entiers.

3.5.4 Variations autour du LSH

Un des problèmes de l’algorithme LSH est sa forte consommation de mémoire à cause de la nécessité de créer de multiples tables de hachage. Une modification possible de l’algorithme est de ne générer qu’une seule table de hachage et pour un point requête de chercher ses voisins dans plusieurs cases de cette table (i.e. un point requête génère donc plusieurs clés de hachage). Dans les travaux de Panigrahy [Pan06], l’auteur propose une méthode pour calculer ces clés multiples pour un point requête. Il propose de générer plusieurs points aléatoires, proches du point requête, et de chercher les voisins du point requête dans les cases de la table correspondant aux clés de ces points aléatoires. Ce genre d’approche qui indexe les points de la base une unique fois, mais parcourt plusieurs “régions” de l’indexation pour une requête nous paraît très intéressante.

Une autre variante présentée dans les travaux de Jegou et al. [JASG08] consiste à choisir un sous-ensemble de fonctions parmi les l fonctions de hachage du LSH. L’idée est qu’une fonction de hachage est performante pour un point requête si elle projette celui-ci à proximité du centre d’une case de la segmentation de l’espace (i.e. selon la grille non orthogonale, voir figure 3.6). Inversement, si la projection du point requête est situé à proximité des frontières d’une case de la grille, la fonction de hachage est moins intéressante. Les auteurs utilisent ce principe pour sélectionner, pour chaque point requête les l' (avec $l' < l$) fonctions de hachage les plus intéressantes. L’inconvénient de cette approche est sa consommation mémoire (l’indexation des points de la base est identique au LSH).

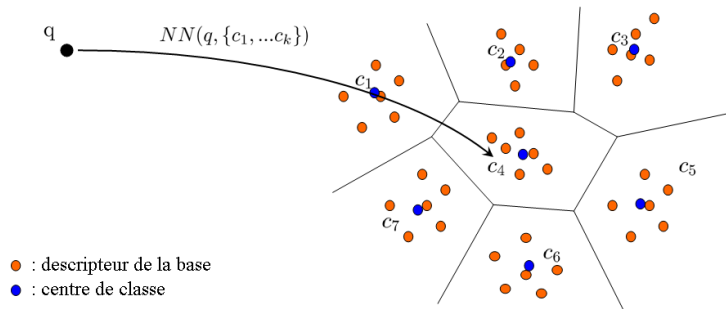


FIGURE 3.7 – Recherche de voisins par segmentation K-means.

3.6 Autres méthodes approchées

3.6.1 Segmentation par K-means

Cette approche a été initialement utilisée sur des descripteurs type SIFT dans les travaux de Sivic [SZ03] pour de la recherche par Bag-Of-Features. Le détail de cette méthode de recherche d’images est présentée dans le chapitre 5. Dans cette partie, nous présentons seulement cette approche comme un algorithme de recherche des plus proches voisins.

Dans cet algorithme, le nuage de points est segmenté en K classes en utilisant un algorithme K-means (on note K le nombre de classes d’un K-means, et k le nombre de voisins cherchés pour une recherche $k - NN$). Ensuite, pour un point requête q , on va chercher sa classe et ne tester comme potentiels voisins que les points du nuage qui appartiennent à cette classe. Cette segmentation du nuage de points peut aussi être vue comme une segmentation de l’espace par un diagramme de Voronoï qui aurait comme graines les K centres de classes. La figure 3.7 illustre ce principe.

Cette approche présente l’inconvénient suivant : pour trouver les voisins d’un point requête, il faut commencer par trouver à quelle classe il appartient. Cette étape est elle-même un algorithme $k - NN$ (avec $k = 1$), qui peut prendre du temps si le nombre de classes du K-means augmente.

Pour contourner cette difficulté, dans Nister et al. [NS06], les auteurs utilisent un K-means hiérarchique. Chaque nuage d’un noeud de l’arbre est divisé en K classes par un K-means (avec K de l’ordre de 10). En itérant cette subdivision, l’algorithme aboutit à un grand nombre de classes. Pour une requête, un algorithme depth-first est utilisé pour trouver la feuille dans laquelle chercher les voisins. Cette structure peut être vue comme un Metric-Tree non binaire. A noter que cette structure d’arbre construite à partir d’un K-means avait été introduite précédemment dans Fukunaga et al. [KF75] et Brin et al. [Bri95]).

3.6.2 PvS framework

Le PvS framework est une structure de données, introduite dans [LÁJA05]. Comme le LSH, elle utilise des projections sur des droites aléatoires mais couplées avec l’algorithme OMEDRANK [FKS03] pour trouver les plus proches voisins.

OMEDRANK (pour “median rank”)

Dans cet algorithme, un ensemble de n droites aléatoires est tiré. Elles sont notées (d_i) . Pour chacune de ces droites, un $B^+ - Tree$ est créé (un $B^+ - Tree$ permet de stocker efficacement des données triées). Chaque point du nuage est ensuite projeté sur toutes les droites. Pour la droite d_i , on enregistre dans son arbre associé le couple (j, v_i^j) où j est l’identifiant du point p_j projeté et v_i^j est la position du projeté de p_j sur la droite d_i . Les couples insérés dans les $B^+ - Tree$ sont triés selon cette position. Pour effectuer une recherche, le point requête q est d’abord projeté sur chacune des droites d_i . On trouve pour chaque B^+Tree la valeur la plus proche du projeté du point requête. On initialise un curseur qui ira dans le sens positif et un curseur pour le sens négatif. Cela crée 2 curseurs pour chaque arbre. On déplace itérativement les curseurs de tous les arbres d’une case. Un compteur mémorise combien de fois chaque point est rencontré avec ce processus. Dès qu’un point est rencontré plus de $d/2$ fois (d est la dimension de l’espace), il est ajouté à la liste des plus proches voisins. On arrête de déplacer les curseurs quand k voisins ont été trouvés.

PvS

Dans [LÁJA05], les auteurs constatent que OMEDRANK n’est pas très efficace dans le cas de grandes bases de données. Ils le modifient en remplaçant les B^+Tree par une structure appelée PvS. Leur constat est que deux points proches peuvent se trouver séparés par beaucoup d’autres points lorsque le nuage est projeté sur une droite. Pour diminuer cet effet, pour une arborescence PvS, les points sont projetés sur une première droite et segmentés selon la position de cette projection (avec un léger recouvrement). Chaque sous-ensemble de points est ensuite reprojecté sur une nouvelle droite aléatoire et re-segmenté. Lorsque le nombre de points dans un sous-ensemble est petit, on arrête de projeter et on utilise un B^+Tree pour stocker les identifiants des points. En pratique, les auteurs utilisent plusieurs arborescences PvS, c’est à dire que le nuage initial est projeté sur plusieurs droites, et l’on démarre une arborescence pour chacune de ces droites. Le nom de structure PvS regroupe les deux idées de projections hiérarchiques sur des droites aléatoires et de segmentation selon ces droites avec un léger recouvrement. L’étape de recherche est basée sur l’algorithme OMEDRANK mais en utilisant les k B^+Tree obtenus en projetant le point requête dans les k arborescences PvS. L’article [LÁJA05] est spécialement intéressant dans le cadre de cette thèse puisqu’il utilise le schéma PvS sur des descripteurs SIFT.

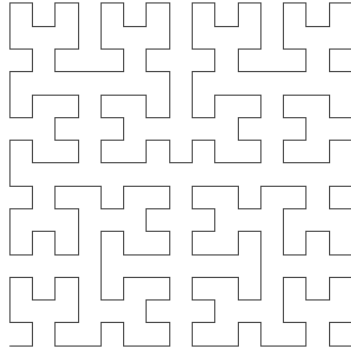


FIGURE 3.8 – Exemple d’une courbe de Hilbert calculée sur un espace 2D.

3.6.3 Space filling curve

Les algorithmes de cette classe utilisent un mapping entre l’espace initial à d dimensions et un espace à une dimension (i.e. la courbe). Les diverses méthodes ont la propriété que deux points proches sur la courbe obtenue, sont aussi proches dans l’espace de départ. Mais un problème est que l’inverse n’est pas vérifié, deux points proches dans l’espace de départ ne sont pas toujours projetés à des positions proches sur l’espace 1D.

On peut citer les algorithmes de Z-ordering [Mor66], les GrayCode [Fal86] ou encore les courbes de Hilbert [Jag90],[LK01] (voir figure 3.6.3). Ces dernières ont par exemple été utilisées dans une application de recherche d’images similaires (travaux de Valle et al. [VCPF08b]), où les auteurs utilisent plusieurs courbes, chacune étant calculée sur un sous-espace de l’ensemble de départ. On trouve aussi une utilisation des courbes de Hilbert pour des recherches de vidéos similaires [JFB04]. Dans ce dernier article, le point requête est projeté sur une courbe et l’on en déduit les portions de cette courbe où l’on doit chercher les plus proches voisins. C’est à dire qu’une recherche exhaustive est appliquée sur les points de ces portions. L’apport de l’article est de sélectionner les portions de courbes à explorer pour que la probabilité de trouver les vrais voisins du point requête soit supérieure à un certain seuil.

3.7 Diminution de la dimension de l’espace

Le temps d’exécution d’un algorithme dépend de la taille du nuage de points mais aussi de la dimension de l’espace. Il est donc toujours intéressant d’étudier s’il est possible de réduire la dimension de l’espace. L’outil classique utilisé pour cela est l’analyse en composantes principales [Fod02]. Dans notre étude, nous considérons que c’est lors de la construction du nuage de points (calcul des descripteurs locaux dans notre cas) que nous pourrions appliquer cette étape de réduction de la dimension. Nous avons présenté dans la section 2.2.4 deux modifications de l’algorithme SIFT (i.e. PCA-SIFT et Gloh-SIFT) qui utilisent la réduction de dimension par analyse en composantes principales.

Nom et référence de l'algorithme	Description	Taille du nuage de points	Dimension de l'espace	Nombre d'images
Article initial sur les SIFT [Low04]	Article qui introduit les descripteurs SIFT. L'objectif n'est pas de travailler sur des grandes bases. L'algorithme utilisé est un Kd-Tree avec l'algorithme Best-Bin-First	40.000	128	32
Algorithme BOND [dVMNK02]	Données synthétiques et histogramme couleurs	100.000	64 ou 128	
Etude comparative sur le PvS [LAJA06]		$200 \cdot 10^6$	72	316.000
Méthode basée sur les courbes de Hilbert [JFB04]	Utilisé dans un système de recherche de vidéos similaires	$750 \cdot 10^6$		14.000 heures de vidéo
Méthode basée sur les Spill-Tree [LMGY04]		275.465	20	
Réduction du nombre de SIFT utilisés [FS07]		$10 \cdot 10^6$	128	100.000 images
Utilisation du LSH sur des PCA-SIFT [KSH04]		environ $18 \cdot 10^6$	36	18.722 images
Moteur de recherche d'images similaires [NS06]	Utilise un seul descripteur par image, représenté sous forme d'arbre.			10^6 images

TABLE 3.1 – Tailles des nuages de points dans la littérature.

3.8 Taille des bases de données dans la littérature

Dans cette partie, nous présentons les tailles des bases de données dans les différents articles étudiés. Certaines bases consistent simplement en un nuage de points. C'est le cas par exemple pour les algorithmes génériques de recherche des plus proches voisins. Pour les algorithmes de recherche d'images similaires par descripteurs locaux, la taille du nuage de points est différente de la taille de la base d'images. Dans ce cas, nous notons le nombre d'images et le nombre de descripteurs. Enfin, nous étudions aussi les bases utilisées pour les algorithmes de recherche d'images par descripteur global pour lesquels la taille du nuage de points étudié est égale au nombre d'images dans la base. Les valeurs collectées sont exprimées dans le tableau 3.1.

3.9 Méthodes d'évaluation de la recherche r-NN approchée

Pour la recherche r-NN exacte, la seule mesure pour comparer les algorithmes est le temps d'exécution. Pour les algorithmes de r-NN approchés, nous voulons que l'algorithme soit rapide mais qu'en plus il retrouve une grande majorité des vrais voisins. Nous présentons ici comment nous allons évaluer les algorithmes selon ces deux critères.

3.9.1 Critères possibles d'évaluation

Un algorithme r-NN approché peut être vu comme une recherche d'information et les performances peuvent alors se mesurer par un couple rappel/précision. Ces valeurs sont mesurées par rapport à une vérité terrain qui est le résultat de la recherche exacte des voisins. On note $(vt_i), i \in 1..n$ les n voisins trouvés par une recherche r-NN exacte (ils constituent une vérité terrain) et $(app_j), j \in 1..m$ les m points retournés par une recherche approchée.

Le rappel est le ratio de points trouvés qui appartiennent à la vérité terrain. Nous avons :

$$rappel = \frac{\text{nombre de points retournés corrects}}{\text{nombre de voisins attendus}} = \frac{\#((vt) \cap (app))}{\#(vt)} \quad (3.8)$$

où $\#$ désigne le cardinal d'un ensemble. Le rappel est une mesure comprise entre 0 et 1 de l'efficacité d'un algorithme pour trouver les vrais voisins. Un rappel de 0 signifie que l'algorithme n'a trouvé aucun des vrais voisins. Un rappel de 1 signifie que l'algorithme retrouve bien tous les voisins.

La précision est le ratio de points retournés qui appartiennent bien à la vérité terrain. Avec les mêmes notations, nous avons :

$$précision = \frac{\text{nombre de points retournés corrects}}{\text{nombre de points retournés}} = \frac{\#((vt) \cap (app))}{\#(app)} \quad (3.9)$$

La précision indique le ratio de bons points parmi tous ceux retournés. La précision est aussi comprise entre 0 et 1. Une précision de 0 signifie que tous les points retournés sont faux. Une précision de 1 signifie que tous les points retournés sont des vrais voisins. Au lieu de mesurer la précision, on peut parfois mesurer le taux d'erreur parmi les voisins retournés, exprimé par $1 - \text{précision}$.

La F-mesure combine le rappel et la précision en une seule valeur :

$$F - \text{mesure} = \frac{2 \times \text{rappel} \times \text{précision}}{\text{rappel} + \text{précision}} \quad (3.10)$$

La formule ci-dessus est aussi appelée $F_1 - \text{mesure}$ car il s'agit de la moyenne harmonique des valeurs de rappel et précision, non pondérés. Mais il est possible d'introduire d'autres mesures en modifiant les pondérations pour privilégier plutôt le rappel ou plutôt la précision dans la mesure finale.

Dans [LMGY04], [AMN⁺98], [GIM99], les auteurs utilisent l'"effective distance error" pour mesurer l'erreur lors d'une recherche des k plus proches voisins :

$$E = \sum_{i=1}^k \left(\frac{d(q, p_i^{app})}{d(q, p_i)} - 1 \right) \quad (3.11)$$

où les points (p_i) sont les k plus proches voisins (vérité terrain) triés selon leur distance au point requête q , et (p_i^{app}) sont les voisins retournés par l'algorithme, triés de la même manière. Pour un algorithme de recherche exacte,

$E = 0$ alors que pour un algorithme approché, $E \geq 0$. Contrairement aux mesures de rappel et précision, cette mesure permet d'avoir une indication si les points retournés sont presque des bons voisins ou s'ils sont très éloignés du point requête. Toutefois, cette mesure ne s'applique pas aux algorithmes r-NN auxquels nous nous intéressons.

3.9.2 Evaluation des algorithmes

Dans ce travail, nous cherchons à obtenir un algorithme qui retourne le plus rapidement possible une liste approchée des voisins recherchés. Pour mesurer la rapidité des algorithmes de manière indépendante de l'implémentation, nous considérons ces algorithmes comme des algorithmes de filtrage. C'est à dire qu'au lieu de calculer les distances euclidiennes avec tous les points du nuage, un algorithme sélectionne un sous-ensemble de ce nuage et ne calcule les distances que sur ce sous-ensemble. Le temps de recherche des voisins d'un point requête q est alors égal à :

$$T(q) = T_f(q) + T_l(q) \quad (3.12)$$

où $T_f(q)$ est le temps nécessaire pour filtrer les points du nuage. Par exemple, si la méthode est basée sur un hachage, ce temps sera celui nécessaire pour calculer la fonction de hachage pour le point q . Ensuite, $T_l(q)$ est le temps nécessaire pour calculer les distances euclidiennes entre le point q et tous les points du sous-ensemble filtré. Le temps $T_f(q)$ est constant et le temps $T_l(q)$ est linéaire par rapport au nombre de points retournés, c'est donc ce dernier qui nous intéresse pour les grands nuages de points que nous traitons. Au lieu de mesurer le temps d'exécution de ces algorithmes pour les comparer, il est plus intéressant de mesurer le ratio entre ce nombre de points filtrés et le nombre de points du nuage initial (car indépendant de l'implémentation). Si le nuage de points est assez grand, on a $T_f(q) \ll T_l(q)$ et le temps de calcul est alors linéairement proportionnel au nombre de points dans le sous-ensemble filtré. Nous noterons *RatioFilter* ce ratio entre le nombre de points du sous-ensemble et le nombre de points du nuage initial.

Au final, pour évaluer les algorithmes, nous mesurerons le rappel obtenu ainsi que *RatioFilter*. Un algorithme sera d'autant plus efficace qu'il aura un rappel proche de 1 pour un *RatioFilter* proche de zéro.

Pour le lecteur habitué à des courbes rappel/précision, ce choix peut paraître peu habituel. Toutefois, il est important de garder à l'esprit qu'une mesure *RatioFilter*/rappel contient les mêmes informations qu'une mesure rappel/précision. Le lien entre *RatioFilter* et rappel/précision peut être obtenu par les égalités suivantes :

$$\begin{aligned} \text{RatioFilter} &= \frac{\text{nombre de points filtrés}}{\text{nombre de points total}} & (3.13) \\ &= \frac{\text{nombre de voisins trouvés}}{\text{précision} \times \text{nombre de points total}} \\ &= \frac{\text{rappel} \times \text{nombre de voisins à trouver}}{\text{précision} \times \text{nombre de points total}} \end{aligned}$$

Cela signifie qu'une courbe *RatioFilter* en fonction du rappel comporte la même information qu'une courbe rappel/précision. Toutefois, dans ce travail, nous préférons la première car elle met plus en avant les deux critères recherchés : temps d'exécution et rappel.

Nous n'utilisons pas la *F - mesure* dans nos travaux car il est difficile d'en déduire de manière intuitive la qualité de l'algorithme évalué. Nous n'utilisons pas non plus l'effective distance error puisqu'elle est limitée aux recherches $k - NN$.

3.10 Conclusion

Dans les paragraphes précédents, nous avons passé en revue les algorithmes qui nous ont paru intéressants dans le cadre de la recherche des plus proches voisins r-NN. Contrairement à certains articles cités, nous cherchons les plus proches voisins d'un type de points particuliers que sont les descripteurs SIFT. Au vu des résultats obtenus par les algorithmes détaillés précédemment, nous avons décidé de nous intéresser particulièrement aux méthodes basées sur le hashing LSH. Ce type de méthode semble en effet donner les meilleurs performances en terme de vitesse de recherche des voisins. De plus, un point qui a motivé nos recherches est qu'il est certainement possible d'adapter les familles de fonctions LSH génériques pour les données SIFT. Cette adaptation aux données a aussi été relevée comme très importante dans l'article présentant l'algorithme BOND présenté en partie 2.6.2 (en choisissant sur quelles dimensions il est le plus important de calculer une distance partielle). Toutefois, un inconvénient majeur des méthodes LSH est leur forte consommation mémoire.

Dans le chapitre suivant, nous allons tout d'abord modifier l'algorithme LSH pour adapter les fonctions de hachage choisies aux données SIFT. Ensuite, nous proposerons une autre approche de recherche r-NN dont l'objectif est d'être aussi rapide que le LSH, mais en consommant beaucoup moins de mémoire. Enfin, nous comparerons ces approches avec l'implémentation de la recherche linéaire sur GPU présentée dans le chapitre précédent.

Chapitre 4

Méthode de hachage proposée

Dans ce chapitre, nous introduisons d'abord une modification de l'algorithme LSH1. Cette modification permet d'adapter les fonctions de hachage aux données SIFT. Dans une seconde partie, nous introduisons notre contribution principale. Il s'agit d'un nouveau hachage qui exploite les caractéristiques des points SIFT. Nous appelons HASHDIM l'algorithme r-NN qui utilise ces fonctions de hachage. De premiers résultats sont exposés afin d'étudier l'impact des paramètres. Nous comparons ensuite les performances de HASHDIM à d'autres algorithmes r-NN. Enfin, nous évaluons HASHDIM dans le cadre de la recherche d'images similaires et dans celui de l'analyse de linéaires de supermarchés.

Sommaire

4.1 Paramétrage du hachage LSH1	88
4.2 LSH-Opti : une modification de l'algorithme LSH1	88
4.3 HASHDIM : un algorithme r-NN approché utilisant des fonctions de hachage basées sur un ordre des dimensions	94
4.4 Evaluation	100
4.5 Recherche d'images similaires	104
4.6 Tests sur une grande base	106
4.7 Résultats pour l'application d'analyse de linéaires	110
4.8 Conclusion	116
4.9 Appendice	118

4.1 Paramétrage du hachage LSH1

L'un des désavantages du LSH1 est qu'il nécessite de régler deux paramètres pour obtenir des résultats optimaux. Il s'agit du nombre de tables de hachage noté m , et du nombre de dimensions hachées, noté k . Dans notre cas, nous cherchons à obtenir ces paramètres optimaux pour des recherches des plus proches voisins dans un voisinage limité par une distance euclidienne à 150,200,250 ou 300. Sur la figure 4.1 nous montrons les courbes *RatioFilter*, *Rappel* obtenues pour différentes valeurs de seuil et différents paramètres. Un algorithme est d'autant plus performant que le *RatioFilter* est petit mais que le rappel est grand (i.e. l'algorithme élimine beaucoup de points du nuage initial mais trouve quand même une majorité des voisins). C'est-à-dire qu'une courbe représente un bon algorithme si elle approche le coin en bas à droite sur les figures montrées.

Pour connaître à quels couples de paramètres (m, k) chaque mesure correspond, le lecteur peut consulter la figure 4.15 en appendice (cette figure n'est lisible clairement que sur une version électronique zoomée, du fait du nombre d'étiquettes présentes). Sur la figure 4.1, nous pouvons lire que les performances augmentent avec le nombre de tables et que les meilleures sont obtenues pour un nombre de tables de 100 ou 120. Nous n'avons pas affiché les courbes pour un nombre de tables supérieur à 120 car elles n'apportent pas un gain significatif. Le principal problème de cette méthode est la consommation mémoire nécessaire. Pour un nuage de n points, la mémoire nécessaire est de $n \times m \times 8$ car chaque SIFT est indexé dans chaque table par sa référence et un checksum (chacun occupant 4 octets). Par exemple, pour $m = 120$, un nuage de 1 million de points nécessite $10^6 \times 120 \times 8 = 960.10^6$ octets. Cela peut devenir très pénalisant si la mémoire nécessaire aux tables de hachage est supérieure à la mémoire vive de la machine. Dans un tel cas, l'algorithme serait très fortement pénalisé par les échanges entre le disque dur et la mémoire (swap).

Dans le paragraphe suivant, nous proposons de modifier les fonctions de hachage LSH pour obtenir des performances similaires mais en limitant le nombre de tables utilisées et donc la consommation mémoire.

4.2 LSH-Opti : une modification de l'algorithme LSH1

On a vu dans la partie concernant les distances partielles que l'adaptation de l'algorithme aux données peut aboutir à des gains importants en temps de calcul. On cherche ici à modifier l'algorithme LSH pour qu'il offre de meilleures performances pour la recherche des plus proches voisins SIFT.

Une fonction de hachage LSH1 choisit k dimensions aléatoirement et pour chacune d'elles choisit un seuil dans l'intervalle $[0, C]$ où C est la valeur maximale des coordonnées. Dans notre implémentation, les coordonnées SIFT sont codées sur un entier 8 bits, c'est-à-dire que l'on a $C = 255$. Toutefois, lorsque l'on étudie les données SIFT, choisir les seuils du LSH dans $[0, 255]$ ne nous paraît pas adapté.

Sur les figures 4.2, on montre la distribution des valeurs des descripteurs

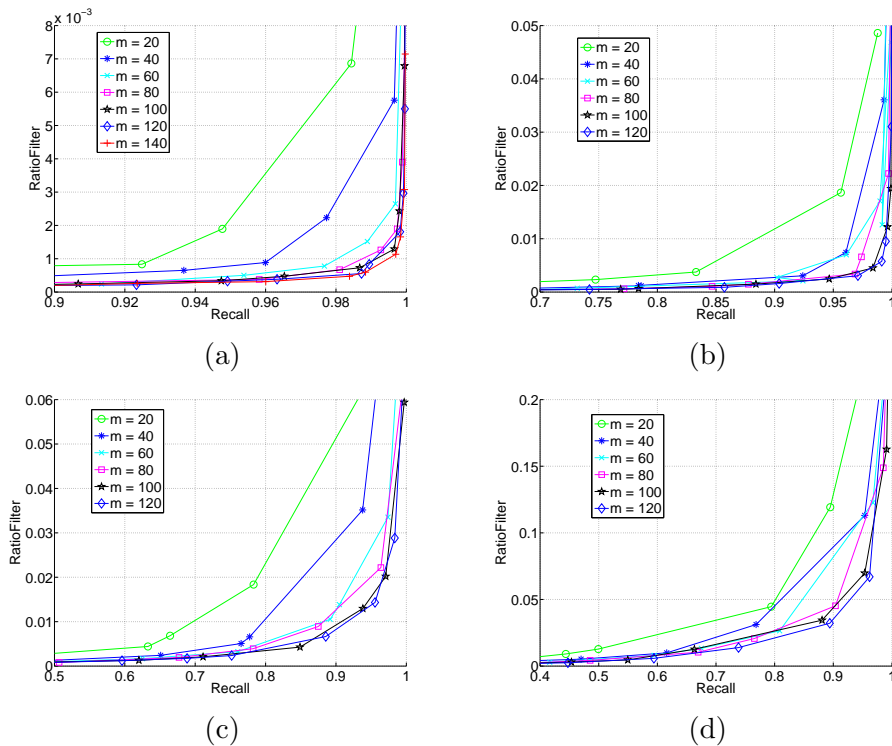


FIGURE 4.1 – Résultats du hachage LSH pour des seuils de a :150, b :200, c :250, d :300.

SIFT selon plusieurs dimensions. Pour la plupart des dimensions, la distribution est similaire à celles des figures 4.2.a et 4.2.b. Seules quelques dimensions ont des distributions proches de celles des figures 4.2.c et 4.2.d, il s’agit de celles orientées selon la direction principale et proches du centre du descripteur (voir 2.3).

Si pour une des dimensions choisies par la fonction de hachage, le seuil tiré aléatoirement est proche de zéro, comme beaucoup de points ont une valeur proche de ce seuil alors de nombreux points potentiellement proches seront projetés dans des cases différentes de la table de hachage. Inversement, si l’on choisit un seuil trop élevé, tous les points auront une valeur inférieure au seuil et seront projetés dans la même case. Il nous a paru intéressant de choisir un intervalle paramétrable $[S_{min}, S_{max}]$ au lieu d’utiliser $[0, C]$ avec $C = 255$. Nous appellerons LSH-Opti cet algorithme, similaire au LSH1, mais qui utilise $[S_{min}, S_{max}]$ au lieu de $[0, C]$.

Pour choisir les paramètres S_{min} et S_{max} , il n’y a pas d’optimalité puisque les performances sont mesurées selon deux critères *RatioFilter* et *Rappel*. De plus, ces deux valeurs sont fonction de quatre paramètres (i.e. $RatioFilter, Rappel = f(m, k, S_{min}, S_{max})$). Nous avons donc procédé de manière expérimentale, sans être exhaustif. Dans le paragraphe précédent, nous avons obtenu les meilleures performances pour $m = 120$ avec une consommation mémoire très importante. On cherche ici à améliorer la consommation mémoire. On va donc utiliser un petit nombre de tables et voir si modifier S_{min} et S_{max} permet de retrouver les

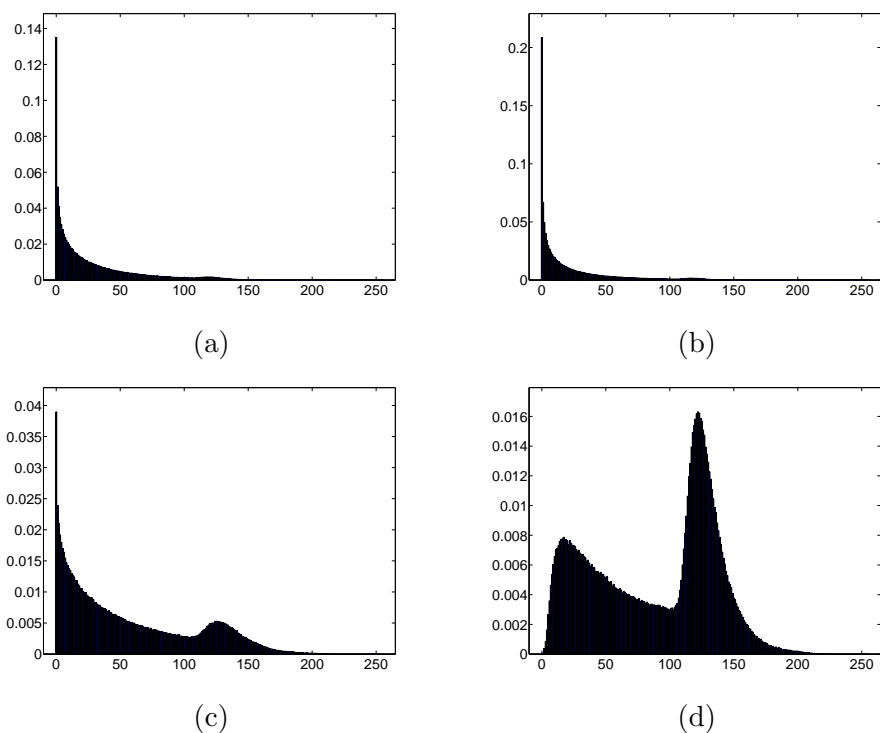


FIGURE 4.2 – Répartition des valeurs des SIFT selon la dimension a : 1, b : 11, c :9 et d :49 sur une base de 600 photos de vacances (976.10^3 descripteurs). Par exemple, l'histogramme de la figure (a) signifie que pour un très grand nombre de descripteurs SIFT, la coordonnée selon la première dimension vaut 0. L'histogramme de la figure (d) signifie que selon la dimension 49, il y a de nombreux SIFT qui ont une valeur entre 120 et 150.

performances obtenues pour $m=120$. En pratique, nous avons procédé comme suit :

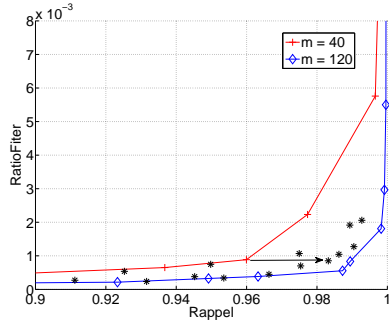
1. Nous fixons m et k
2. Pour différentes valeurs de S_{min} et S_{max} , nous mesurons le couple *RatioFilter,Rappel*.
3. Nous choisissons un couple S_{min}, S_{max} qui permet d'obtenir des performances proches de celles obtenues pour la courbe de $m = 120$.

Cette approche est partielle car si l'on choisit un autre couple m, k , le résultat sera différent. Toutefois, les résultats montrent que cette approche permet de choisir des valeurs de S_{min} et S_{max} qui améliorent nettement les performances du LSH1. Les résultats sont présentés pour les seuils de distance 150, 200, 250 et 300 sur les figures 4.3.a, 4.4.a, 4.5.a et 4.6.a. Sur ces figures, le processus en trois étapes ci-dessus est illustré par une flèche. L'origine de la flèche représente le résultat de l'algorithme LSH1 pour le couple m, k choisi. L'extrémité de la flèche représente le résultat pour l'algorithme LSH-Opti (c'est-à-dire pour lequel nous avons changé S_{min} et S_{max}). Cette flèche représente l'amélioration des performances lorsque l'on utilise LSH-Opti au lieu de LSH. Le choix de l'extrémité de la flèche est subjectif. Pour chaque figure, il est possible de choisir n'importe quel résultat représenté par une étoile sur ces figures. Dans les paragraphes suivants, nous détaillons ces résultats.

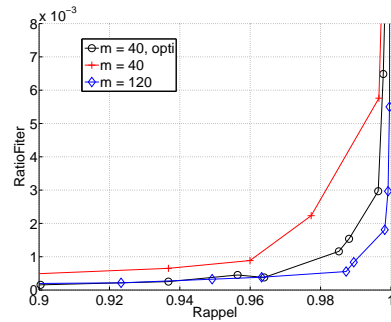
Seuil de 150 Pour un seuil de 150, nous obtenons les meilleurs résultats en cherchant à améliorer les résultats obtenus pour $m = 40$. Sur la figure 4.3.(a), nous reprenons les courbes de performances obtenues pour $m = 40$ (performances que l'on cherche à améliorer) et pour $m = 120$ qui obtient les meilleures performances. Chaque point affiché (sous forme d'étoile) correspond à une mesure de performance faite en modifiant S_{min} et S_{max} pour le couple $(m, k) = (40, 100)$. Les performances obtenues sont très similaires à celles lorsque $m = 120$ alors qu'on utilise 3 fois moins de mémoire. La même figure avec plus d'annotation est visible en annexe (4.16). On y voit que le couple $[S_{min}, S_{max}] = [20, 180]$ offre un bon compromis. Sur la figure 4.3.b, on utilise ces valeurs $[S_{min}, S_{max}] = [20, 180]$ pour $m = 40$ tout en faisant varier k . Cette courbe a pour légende $m = 40, opti$. On voit qu'au final, on obtient des performances similaires à la courbe $m = 120$.

Seuil de 200 Pour un seuil de 200, on cherche à améliorer la courbe obtenue pour $m = 20$. La figure 4.4.(a) montre les performances obtenues en modifiant S_{min} et S_{max} pour $m, k = 20, 70$. Pour le couple $[S_{min}, S_{max}] = [40, 140]$, les performances sont équivalentes à celles obtenues pour $m = 120$ en utilisant 6 fois moins de mémoire. On utilise cette valeur ($[S_{min}, S_{max}] = [40, 140]$) pour mesurer les performances finales présentées sur 4.4.(b). Pour toutes valeurs de k utilisées, on obtient des performances similaires à $m = 120$.

Seuil de 250 Pour un seuil de 250, on cherche à améliorer la courbe obtenue pour $m = 20$. La figure 4.5.(a) montre les performances obtenues en modifiant

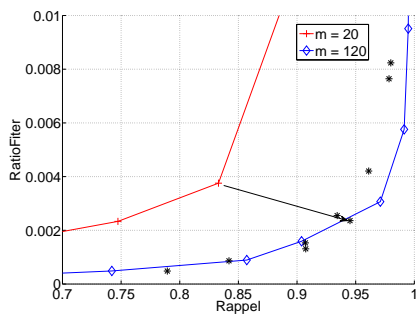


(a)

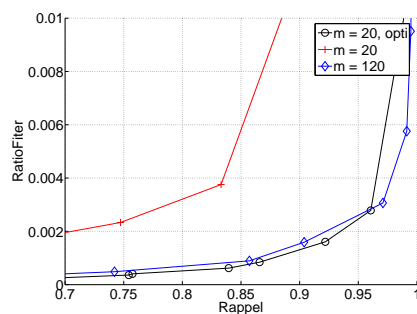


(b)

FIGURE 4.3 – Sur chaque figure, les courbes $m = 40$ et $m = 120$ sont des rappels des performances du LSH obtenues pour $[Smin, Smax] = [0, 255]$. Sur la figure (a), chaque étoile correspond à une mesure de performance obtenue en modifiant $Smin, Smax$ pour le couple $(m, k) = (40, 100)$. Sur la figure (b), k varie pour $m = 40$ et $[Smin, Smax] = [20, 180]$. Sur les deux figures, le seuil de distance est de 150.



(a)



(b)

FIGURE 4.4 – Sur chaque figure, les courbes $m = 20$ et $m = 120$ sont des rappels des performances du LSH obtenues pour $[Smin, Smax] = [0, 255]$. Sur la figure (a), chaque étoile correspond à une mesure de performance obtenue en modifiant $[Smin, Smax]$ pour le couple $m, k = 20, 70$. Sur la figure (b), k varie pour $m = 20$ et $Smin, Smax = [40, 140]$. Sur les deux figures, le seuil de distance est de 200.

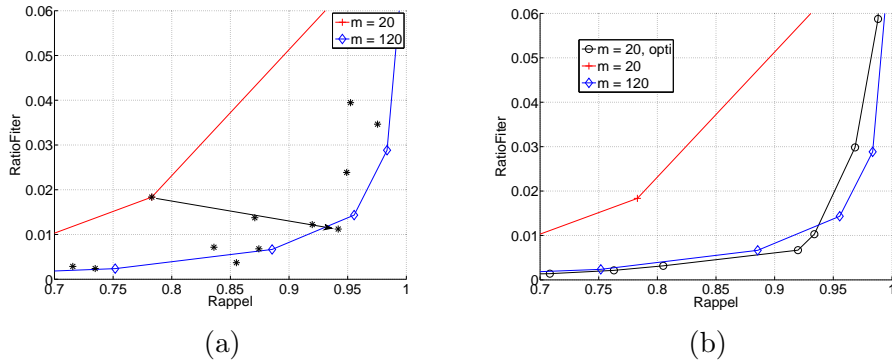


FIGURE 4.5 – Sur chaque figure, les courbes $m = 20$ et $m = 120$ sont des rappels des performances du LSH obtenues pour $S_{min}, S_{max} = 0, 255$. Sur la figure (a), chaque étoile correspond à une mesure de performance obtenue en modifiant $[S_{min}, S_{max}]$ pour le couple $m, k = 20, 60$. Sur la figure (b), k varie pour $m = 20$ et $S_{min}, S_{max} = [60, 120]$. Sur les deux figures, le seuil de distance est de 250.

S_{min} et S_{max} pour $m, k = 20, 60$. Pour le couple $S_{min}, S_{max} = 60, 120$, les performances sont équivalentes à celles obtenues pour $m = 120$ en utilisant 6 fois moins de mémoire.

Seuil de 300 Enfin, pour un seuil de 300, on modifie S_{min}, S_{max} pour le couple $m, k = 20, 50$. La figure 4.6 montre les performances obtenues. On choisit le couple $S_{min}, S_{max} = 60, 120$ qui aboutit aux mêmes performances que $m = 120$ et consomme 6 fois moins de mémoire.

4.2.1 Conclusion

Nous avons montré dans les paragraphes précédents que le LSH permet de filtrer beaucoup de points du nuage. Cela permet de ne calculer la distance euclidienne que sur un petit sous-nuage ainsi filtré. Toutefois, l'inconvénient majeur du LSH est la mémoire requise. Nous avons montré que sur nos jeux de points, les meilleures performances sont obtenues lorsque nous utilisons 120 tables de hachage. Cela implique forcément une importante consommation mémoire et réduit la possibilité de travailler sur de très grands nuages en mémoire vive. Nous avons alors proposé de travailler avec un nombre de tables plus faible mais de modifier les fonctions de hachage utilisées. Nos résultats montrent qu'en utilisant notre méthode (i.e. le LSH-Opti), l'utilisation de 20 tables en moyenne aboutit à des performances similaires à 120 tables sur le LSH classique. La consommation mémoire est alors réduite par un facteur 6.

Toutefois, cela n'est parfois pas suffisant si l'on souhaite travailler sur des nuages de plusieurs centaines de milliers de points. Par exemple, pour une base de 200.000 images générant chacune 1000 SIFT, le nuage de points final fait $2 \cdot 10^8$ points. Même avec seulement 20 tables utilisées pour le LSH, la mémoire nécessaire à l'indexation est de $2 \cdot 10^8 \times 20 \times 8 = 32Go$ (chaque point enregistré dans une table nécessite 8 octets : 4 pour son identifiant et 4 pour le checksum).

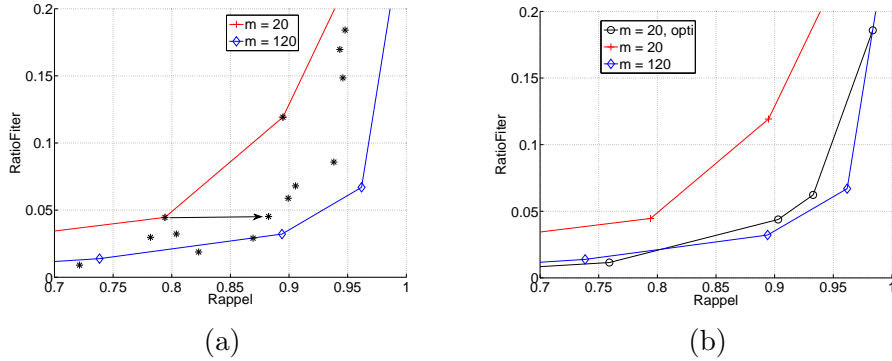


FIGURE 4.6 – Sur chaque figure, les courbes $m = 20$ et $m = 120$ sont des rappels des performances du LSH obtenues pour $S_{min}, S_{max} = 0, 255$. Sur la figure (a), chaque étoile correspond à une mesure de performance obtenue en modifiant S_{min}, S_{max} pour le couple $m, k = 20, 50$. Sur la figure (b), k varie pour $m = 20$ et $S_{min}, S_{max} = [60, 120]$. Sur les deux figures, le seuil de distance est de 300.

Dans la section suivante, nous proposons donc une méthode de hachage qui est plus optimale en mémoire. C'est-à-dire qu'un point n'est haché qu'une seule fois dans une seule table de hachage.

4.3 HASHDIM : un algorithme r-NN approché utilisant des fonctions de hachage basées sur un ordre des dimensions

Les fonctions de hachage que nous introduisons ici sont inspirées de l'algorithme BOND présenté en 2.6.2 et de notre analyse de l'utilisation de distances partielles pour la recherche linéaire exposée en 2.6.3. Dans ces deux méthodes, certaines dimensions parmi les 128 de l'espace sont choisies pour calculer une distance partielle et ainsi éliminer rapidement des points qui ne peuvent pas être des voisins du point requête. Ces dimensions sont choisies globalement dans l'algorithme BOND et aussi dans l'heuristique que nous avons proposée pour accélérer la recherche linéaire en utilisant des distances partielles. Nous proposons de détecter ces dimensions pour chaque point et de s'en servir pour construire des clés de hachage et indexer les points.

Dans un algorithme de type LSH (i.e. LSH1, LSH2, LSH-Opti), chaque point de la base est haché dans plusieurs tables et pour chaque point requête, on parcourt une seule case de chaque table. Un des problèmes de cette approche est la consommation mémoire nécessaire pour stocker les tables de hachage. On a vu précédemment que pour des valeurs de seuil de voisinage faible, un très grand nombre de tables (e.g. 100) est nécessaire pour obtenir les meilleures performances. Dans notre approche, nous utilisons une seule table de hachage et un point de la base de données n'est haché qu'à une seule position (i.e. une seule clé par point). C'est lors de la recherche des voisins d'un point requête

que l'on va parcourir plusieurs cases de la table. Cela permet d'être optimal en terme de consommation mémoire. Sur ce point, notre algorithme s'inspire des travaux de Panigrahy [Pan06] (voir paragraphe 3.5.4). Toutefois, l'auteur indique que pour obtenir les meilleurs résultats, il est nécessaire de conserver plusieurs tables de hachage, ce que nous souhaitons éviter pour optimiser la consommation mémoire au mieux. Ce principe de multiples recherches est aussi utilisé par Joly et al. [JFB04]. Dans cet article, au lieu de visiter plusieurs cases d'une table de hachage, l'algorithme cherche les voisins d'un point requête dans plusieurs portions d'une courbe de Hilbert (voir partie 3.6.3).

Dans les paragraphes suivants, nous présentons notre contribution qui consiste en de nouvelles fonctions de hachage pour un algorithme r-NN approché, spécialement étudié pour les descripteurs SIFT. Ces fonctions permettent de faire une indexation proche du LSH mais optimale en quantité mémoire requise.

4.3.1 Description des fonctions de hachage

Pour chaque point de la base de données, on détecte ses k dimensions les plus distinctives, avec k plus petit que le nombre total de dimensions (i.e. 128 pour les SIFT). Le vecteur de ces k dimensions est utilisé pour générer la clé de hachage de ces points (en utilisant une fonction g que nous introduisons plus tard). L'idée centrale est que deux points qui partagent les mêmes dimensions distinctives ont une bonne probabilité d'être proches. Intuitivement, un point est distinctif selon une dimension s'il est éloigné de la valeur moyenne selon cette dimension. Une autre idée est que les dimensions avec une forte variance sont les plus distinctives.

Pour traduire ces idées, nous introduisons les notations suivantes : $x^i = \langle x_1^i, x_2^i, \dots, x_{128}^i \rangle$ le i^{eme} descripteur SIFT de la base de données et β une fonction qui mesure la distinctivité des descripteurs, sur chaque dimension :

$$\beta(x_j^i) = |\bar{x}_j - x_j^i| \sigma_j^\alpha \quad (4.1)$$

où \bar{x}_j est la valeur moyenne des SIFT selon la dimension j , σ_j est l'écart-type des SIFT selon cette même dimension et α est un paramètre. Cette fonction retourne une valeur élevée si une coordonnée est loin de la valeur moyenne de cette coordonnée où l'écart-type est important. Le paramètre α permet de pondérer l'écart à la moyenne de ce point et l'écart-type de cette dimension. Nous avons expérimenté plusieurs valeurs de α et $\alpha = 0.5$ donne les meilleurs résultats. Pour un point x^i , nous notons $D(x^i) = \langle D_1(x^i), D_2(x^i), \dots, D_{128}(x^i) \rangle$ le vecteur des dimensions (avec des valeurs dans [1..128]) triées par ordre décroissant de distinctivité :

$$\beta(x_{D_1(x^i)}^i) > \beta(x_{D_2(x^i)}^i) > \dots > \beta(x_{D_{128}(x^i)}^i) \quad (4.2)$$

Ainsi, $D_1(x^i)$ est la dimension selon laquelle le point x^i est le plus distinctif. $D_{128}(x^i)$ est la dimension selon laquelle le point x^i est le moins distinctif.

L'idée sous-jacente à notre structure d'indexation est que si deux points q et x^i sont proches, les premières valeurs des vecteurs $D(q)$ et $D(x^i)$ seront

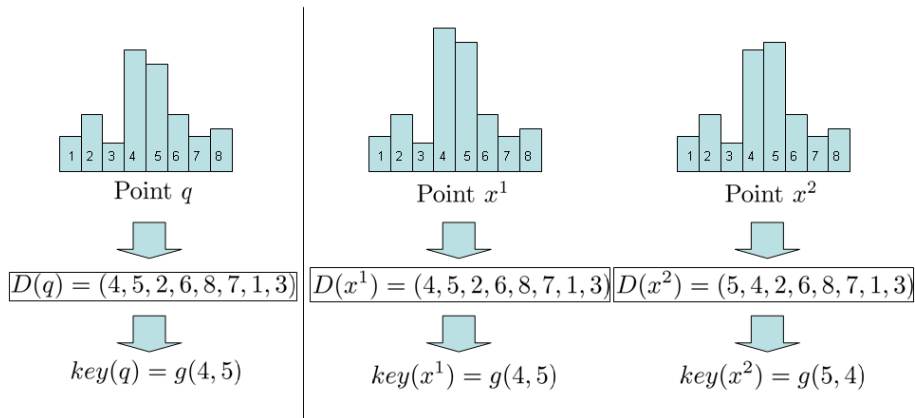


FIGURE 4.7 – Exemple de notre hachage sur trois points dans un espace à 8 dimensions. Pour des raisons de simplicité, nous avons ici pris $\beta(x_j^i) = x_j^i$. Les points q et x^1 sont similaires et seront bien hachés avec la même clé (i.e. $g(4, 5)$). Toutefois, le point x^2 , lui aussi similaire à q sera haché avec une clé différente.

identiques (ou presque identiques, par exemple l'ordre peut varier). En d'autres termes, deux points proches sont distinctifs selon les mêmes dimensions. Pour un point x^i , les k premières valeurs du vecteur $D(x^i)$ seront utilisées pour générer sa clé de hachage : $key(x^i) = g(D_1(x^i), \dots, D_k(x^i))$ (où g est une fonction de hachage classique que nous introduisons plus tard). Un exemple simple est d'utiliser les deux premières dimensions : les points de la base de données sont stockés dans une table de manière à ce que le point x^i soit enregistré dans la case indexée par $g(D_1(x^i), D_2(x^i))$. Ensuite, pour un point requête q , la recherche de ses voisins est limitée aux points de la case indexée par $g(D_1(q), D_2(q))$. La figure 4.7 illustre cet exemple sur un point requête et deux points de la base (dans un espace à 8 dimensions).

Le problème de cette approche est que de nombreux voisins ne sont pas retrouvés (e.g. cas du point x^2 de la figure 4.7) car la fonction de hachage est trop sélective. Même si deux points x^i et q sont très proches, les premières valeurs de $D(x^i)$ et $D(q)$ ne sont pas forcément exactement les mêmes ou elles peuvent être ordonnées différemment. Cette difficulté est similaire à celle expliquée dans [GIM99] lors de l'utilisation d'une seule fonction LSH. Leur solution est d'utiliser plusieurs fonctions de hachage, chacune associée à une table. Comme nous l'avons dit précédemment, cette solution a l'inconvénient de consommer beaucoup de mémoire. Dans notre solution, un point de la base est haché dans une seule case de la table mais plusieurs cases sont parcourues lors de la recherche des voisins (voir 4.3.2 pour cet algorithme de recherche). Afin d'éviter que deux vecteurs de k dimensions contenant des valeurs similaires mais dans deux ordres différents soient hachés dans deux cases différentes, les valeurs des vecteurs $D(x^i)$ sont triées par ordre croissant avant de générer la fonction de hachage. C'est-à-dire que la clé de hachage d'un point x^i est obtenue par $key(x^i) = g(tri(D_1(x^i), \dots, D_k(x^i)))$. La figure 4.8 illustre ce principe.

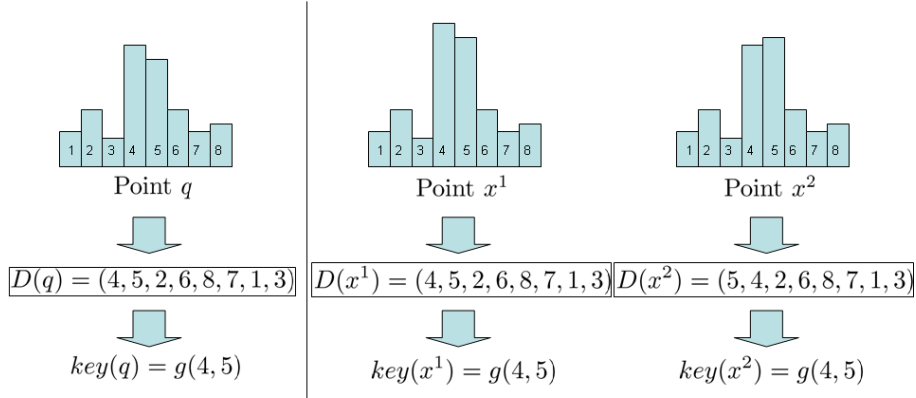


FIGURE 4.8 – Exemple reprenant les points de la figure 4.7. Pour un point x^i , la fonction de hachage est maintenant appliquée aux valeurs de $D(x^i)$ triées. Par rapport à l'exemple de la figure 4.7, le point x^2 obtient la même clé que le point q .

Nous souhaitons d'autre part que notre algorithme soit particulièrement efficace pour les vecteurs SIFT. Comme on l'a vu dans l'étude statistique des points SIFT de la section 2.3, 4 dimensions ont une variance plus élevée que les autres dimensions. En conséquence, ces dimensions seront très souvent présentes dans le vecteur $D(x^i)$ et donc ne sont pas distinctives pour un point. Il est donc inutile de s'en servir pour calculer la clé de hachage. On force donc : $\beta(x_j^i) = 0$ pour $j \in \{41, 73, 81, 49\}$ (les 4 dimensions notées ici correspondent à celles de plus forte variance dans notre implémentation), de manière à ne les choisir que si l'on considère quasiment toutes les dimensions (i.e. $k > 124$).

La table de hachage finale notée H est un tableau à 1 dimension, de taille c . A noter qu'il aurait été possible d'utiliser un tableau G à k dimensions, et d'enregistrer le point x^i en $G[D_1(x^i)][D_2(x^i)]\dots[D_k(x^i)]$, mais cette solution consomme beaucoup trop de mémoire (i.e. 128^k). C'est pour cette raison qu'un point x^i est enregistré dans la case $H[g(tri(D_1(x^i), \dots, D_k(x^i)))]$. Pour la fonction g , nous avons le choix parmi toutes les fonctions de hachage classiques qui vont de $[0, 127]^k$ dans $[0, c - 1]$. Dans notre implémentation, nous utilisons une fonction g du type donné dans [SDI06b] :

$$g(a_1, a_2, \dots, a_k) = \left(\left(\sum_{i=1}^k r_i a_i \right) \bmod P \right) \bmod c \quad (4.3)$$

où P est un nombre premier et r_i sont des entiers aléatoires.

Toutefois, cette fonction g introduit de nouvelles collisions qui n'auraient pas eu lieu avec un tableau à k dimensions (i.e. pour $(a_1, a_2, \dots, a_k) \neq (b_1, b_2, \dots, b_k)$, il est possible d'avoir $g(a_1, a_2, \dots, a_k) = g(b_1, b_2, \dots, b_k)$. Pour détecter ces collisions, nous utilisons une seconde fonction de hachage g' (similaire à g) pour calculer une somme de contrôle (dite checksum) à partir du vecteur à k dimensions $tri(D_1(x^i), \dots, D_k(x^i))$. Dans notre implémentation, nous choisissons :

$$g'(a_1, a_2, \dots, a_k) = \left(\left(\sum_{i=1}^k r'_i a_i \right) \text{mod } P \right) \quad (4.4)$$

où P est un nombre premier et r'_i sont aussi des entiers aléatoires. Au final, chaque point x^i de la base est stocké dans la table sous forme d'une référence (i.e. son id i) et le checksum associé (i.e. $g'(tri(D_1(x^i), \dots, D_k(x^i)))$).

4.3.2 Algorithme de recherche

Pour un point requête q , on cherche à savoir quelle case de la table H il faut parcourir pour retrouver avec une forte probabilité ses voisins. Pour cela, le vecteur de ses dimensions les plus distinctives $D(q)$ est calculé. Il serait possible, comme pour un point de la base de calculer une unique clé pour q par : $key(q) = g(tri(D_1(q), \dots, D_k(q)))$ et ensuite de chercher les voisins en parcourant linéairement les points de $H[key(q)]$. Pour détecter les collisions, nous calculons aussi le checksum du point requête $g'(tri(D_1(q), \dots, D_k(q)))$ et la distance euclidienne entre q et un point x^i n'est calculée que si les deux checksum sont égaux. Toutefois, avec cette méthode, une grande majorité des voisins de q ne sont pas retrouvés. Nous proposons donc une méthode pour chercher les voisins de q dans plusieurs cases de la table H .

L'objectif est de faire en sorte que, si deux points q et x^i ont des vecteurs $D_1(q), \dots, D_k(q)$ et $D_1(x^i), \dots, D_k(x^i)$ qui contiennent quasiment les mêmes valeurs, le point x^i soit testé comme voisin potentiel (i.e. il faut calculer la distance entre q et x^i). Nous avons donc cherché à faire des permutations entre les valeurs de $D_1(q), \dots, D_k(q)$ et des valeurs de $D_{k+1}(q), \dots, D_{128}(q)$ pour générer de multiples vecteurs de k valeurs (autres que $D_1(q), \dots, D_k(q)$) et pouvoir les utiliser pour calculer de multiples clés pour q . Pour choisir ces permutations, on remarque qu'il est plus intéressant de permuter des éléments de $D_1(q), \dots, D_k(q)$ avec $D_{k+1}(q)$ qu'avec $D_{128}(q)$.

Nous introduisons un nouveau paramètre n (avec $n \geq k$) qui représente le nombre de dimensions que nous considérons pour générer des permutations pour le point q . Nous proposons de choisir comme permutations toutes les combinaisons de k valeurs parmi les n plus distinctives pour le point q . C'est-à-dire toutes les combinaisons de k valeurs dans $D_1(q), \dots, D_n(q)$. Nous notons (c_u) ces combinaisons (il y en a $\binom{n}{k}$ différentes avec $C\binom{n}{k} = \frac{n!}{k!(n-k)!}$). Pour chaque permutation c_u , nous calculons une clé de hachage $g(tri(c_u))$ ainsi que le checksum associé $g'(tri(c_u))$. Pour chaque combinaison c_u , la case de la table de hachage en $g(tri(c_u))$ est parcourue linéairement et pour chaque point ayant un checksum égal à $g'(tri(c_u))$, sa distance euclidienne avec q est calculée pour tester si c'est un voisin ou non. La figure 4.9 illustre ce principe. Avec cette approche, aussi appelée "assignement multiple", la probabilité de trouver les voisins du point requête est augmentée.

Un élément très important à remarquer est la faible consommation mémoire de cet algorithme. Un point de la base est indexé à un seul endroit tandis que beaucoup d'autres méthodes indexent les points dans plusieurs cases ou plusieurs tables pour obtenir de fortes valeurs de rappel.

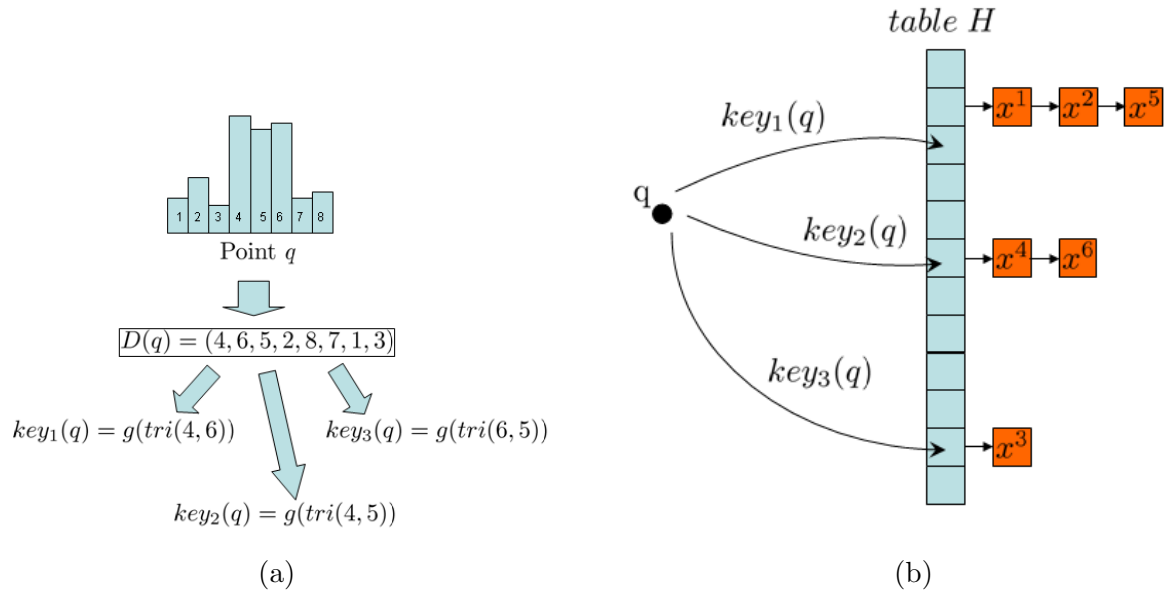


FIGURE 4.9 – Génération de plusieurs clés de hachage pour un point requête. Ici, nous avons choisi $n = 3$ et $k = 2$. (a) Les trois clés de hachage générées par le point q . (b) Les trois cases de la table de hachage qui sont parcourues pour chercher des points voisins de q .

4.3.3 Premiers résultats

Dans cette partie, nous cherchons à présenter les premiers résultats obtenus par cette méthode. Le but ici est de choisir le meilleur jeu de paramètres et de les utiliser dans les parties suivantes pour comparer cet algorithme à d'autres méthodes. Les figures 4.10 montrent les courbes *RatioFilter*-Rappel obtenues pour différentes valeurs de couples (n, k) . Chaque courbe correspond à une unique valeur n et k varie. Comme pour les tests précédents, chaque figure correspond à un seuil de voisinage parmi 150, 200, 250, 300. En appendice de ce chapitre, la figure 4.17 montre les mêmes résultats mais en affichant les indices k sur chaque mesure.

Sur ces figures, on peut voir que plus on augmente n , meilleurs sont les résultats. Toutefois, les courbes obtenues pour $n = 14$ apportent un gain très faible. On choisira donc plutôt comme paramètre de référence $n = 12$ et l'on fera varier k pour choisir entre rappel et *RatioFilter*. Sur une courbe (i.e. à n fixé), plus k augmente, plus le nombre de clés de hachage généré pour un point requête diminue (car un point requête a $\binom{n}{k}$ clés de hachage et $\binom{n}{k}$ décroît avec k à n fixé). Et si le nombre de clés de hachage d'un point requête diminue, moins de voisins sont testés, donc le rappel diminue et le *ratioFilter* diminue aussi. Dans la partie suivante, nous comparons les résultats du hachage HASHDIM à d'autres algorithmes dans le cadre de la recherche r-NN sur des points SIFT.

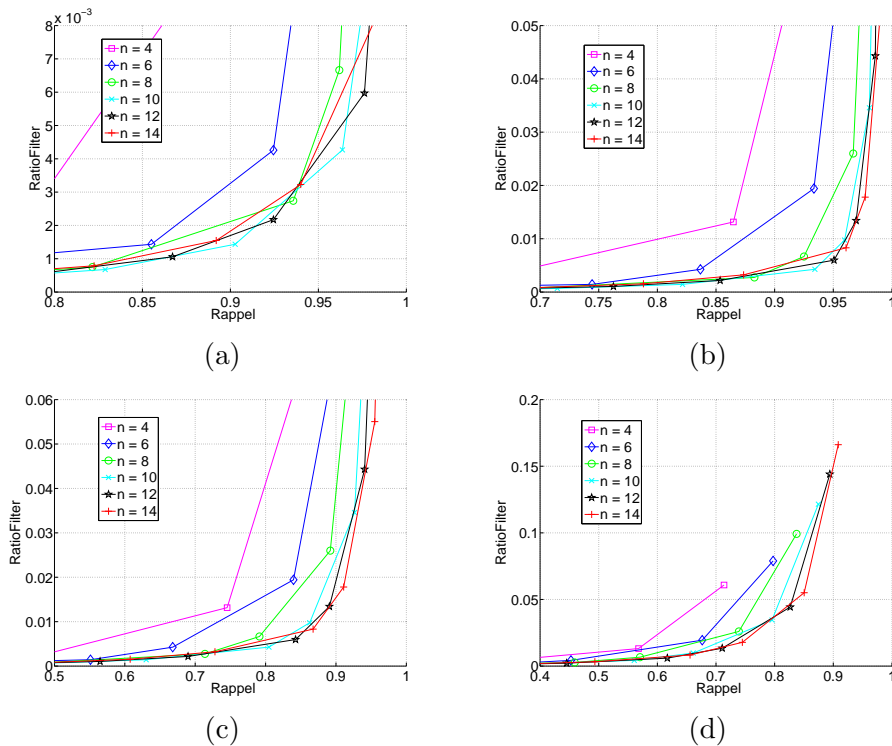


FIGURE 4.10 – Résultats du hachage pour des seuils de (a) 150, (b) 200, (c) 250, (d) 300.

4.4 Evaluation

Dans cette partie, nous comparons les performances de HASHDIM avec celles obtenues par d'autres algorithmes de recherche des plus proches voisins. Tous nos tests concernent le problème r-NN. La liste des algorithmes que nous comparons est :

- HASHDIM
Algorithme que nous avons introduit dans les paragraphes précédents.
- BBF
Notre implémentation de l'algorithme Best-Bin-First sur KdTree, présenté en 3.4.2. Nous incluons cet algorithme dans nos tests car c'est celui utilisé par D. Lowe dans l'article initial sur les SIFT.
- LSH1
Notre implémentation de l'algorithme du LSH présenté en 3.5.1, d'après les travaux de Gionis et al. [GIM99].
- LSH-Opti
Notre implémentation de la modification du LSH1 que nous avons proposée (décrite dans la section 4.2).
- LSH2
Seconde version du LSH, décrite en 3.5.3, d'après les travaux de Datar et al. Package E2LSH fourni par les auteurs de [DIIM04].
- KMeans

Notre implémentation de la recherche de voisins par segmentation K-means du nuage de points, telle que décrite en section 3.6.1.

– HKMeans

Notre implémentation de la recherche de voisins par K-means hiérarchique, tel que décrit en 3.6.1, d’après les travaux de Nister et al. [NS06].

Cette liste ne vise pas l’exhaustivité mais plutôt à faire ressortir les algorithmes, qui à la date de nos expérimentations, nous paraissent les plus adaptés à notre problème. Avant de présenter les résultats de tests exécutés avec ces méthodes, nous avons choisi de présenter les résultats qui comparent uniquement notre algorithme HASHDIM avec l’algorithme BBF. En effet, ce dernier algorithme n’est pas adapté aux recherches r-NN et nous ne l’inclurons donc pas dans les tests complets.

Enfin, dans chacun des tests effectués dans les paragraphes suivants, nous lançons des recherches r-NN en utilisant comme valeurs de r : 150, 200, 250, 300. Le choix de cette gamme de valeurs est justifié dans la partie 2.5.

4.4.1 Comparaison avec l’algorithme Best-Bin-First

Il nous paraît nécessaire d’inclure l’algorithme Best-Bin-First dans notre comparatif puisqu’il s’agit de celui utilisé dans l’article initial sur les SIFT (voir section 3.4.2 pour la description de cet algorithme). Le paramètre permettant de faire varier les performances entre rappel et *RatioFilter* est le nombre de branches revisitées que nous notons n_{BRANCH} . Pour n_{BRANCH} faible, l’algorithme parcourt très peu de branches et est donc très rapide, mais il manque certains voisins. Si n_{BRANCH} est élevé, beaucoup de branches sont parcourues, l’algorithme est donc lent mais retrouve plus de voisins.

La figure 4.11 montre les courbes de performance pour cet algorithme. Sur ces figures, la courbe HASHDIM correspond aux performances obtenues par notre hachage. On voit que notre algorithme obtient de bien meilleurs résultats. Dans les paragraphes suivants, nous n’afficherons plus les performances de l’algorithme Best-Bin-First pour nous concentrer sur l’algorithme HASHDIM et d’autres algorithmes qui offrent des résultats proches.

A noter que ce résultat concernant l’algorithme BBF peut paraître surprenant. Dans l’article initial sur les SIFT [Low04] où il est utilisé, les auteurs en sont satisfaits. Ils affirment que pour une base de 100.000 points, cet algorithme donne un gain de deux ordres de magnitude et manque moins de 5% des voisins. Toutefois, ils utilisent cet algorithme pour trouver les deux plus proches voisins d’un point. A partir de ces deux plus proches voisins retournés par l’algorithme p_1 et p_2 , on déclare le point requête q en correspondance avec le plus proche voisin si le ratio des deux distances est inférieur à 0.8 (i.e. $\frac{d(q,p_1)}{d(q,p_2)} < 0.8$).

Appelons p_1^t et p_2^t les deux véritables plus proches voisins d’un point requête q . Si p_1^t est proche de q et éloigné de p_2^t , l’algorithme BBF va trouver p_1^t comme plus proche voisin de q avec une forte probabilité (c’est un cas simple pour le kd-tree). Et donc pour ce point, on va déclarer p_1^t en correspondance avec q avec une forte probabilité, ce qui est correct. Inversement, si la région autour de p_1^t est dense, q et p_1^t ne sont certainement pas voisins selon le critère du ratio.

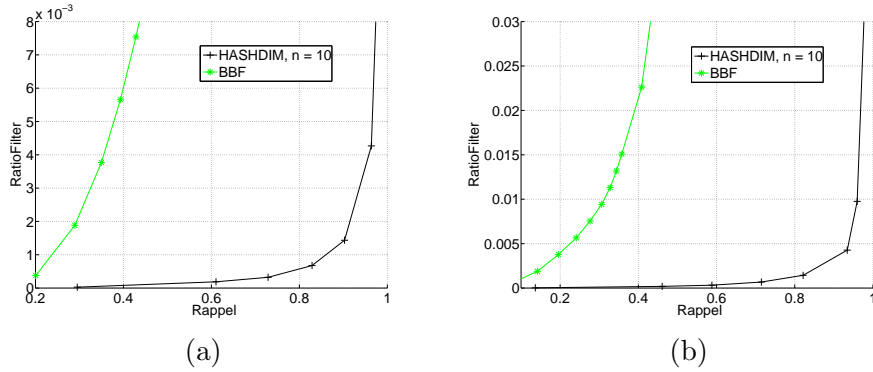


FIGURE 4.11 – Comparaison de l’algorithme BBF avec notre hachage HASHDIM, pour un seuil de : (a) 150 et (b) 200. Le nuage de points est composé de 265.000 points et les résultats sont moyennés sur 1.760 requêtes.

n_{BRANCH}	ratioFilter	rappel
265	0.001	0.69
2650	0.01	0.80
26500	0.1	0.89

TABLE 4.1 – Résultats obtenus pour l’algorithme BestBinFirst pour la recherche des voisins selon la méthode utilisant un ratio (pris à 0,8 ici), sur un nuage de 265.000 points, moyennés sur 1.760 requêtes.

Si le voisinage de p_1^{\dagger} est dense, l’algorithme va certainement se tromper de plus proches voisins, mais comme ils seront proches, au final, il trouvera que q n’a pas de voisins au sens du ratio. C’est-à-dire que même si l’algorithme se trompe sur quels points sont les vrais voisins, sa réponse finale (correspondance ou non au sens du ratio) sera correcte avec une forte probabilité.

Pour vérifier que l’algorithme BestBinFirst est bien adapté lorsque l’on recherche des voisins en utilisant la méthode des ratios des distances aux deux plus proches voisins, nous en mesurons les performances. Nous fixons le seuil de ratio des distances à 0.8, comme dans [Low04]. Le tableau 4.1 montre les valeurs de rappel obtenues pour certaines valeurs de n_{BRANCH} . Par exemple, nous obtenons un couple RatioFilter et Rappel de 0.01 et 0.80 alors que pour une recherche r-NN avec $r = 200$ (i.e. figure 4.11.b), pour un RatioFilter de 0.01, nous obtenons un rappel proche de 0.3, beaucoup moins bon. Les résultats du BBF sur ce problème de la recherche de voisins par $k - NN$ (avec $k=2$ et utilisation de ratio de distances) sont nettement meilleurs que ceux obtenus pour le problème de la recherche r-NN. Pour le problème de la recherche r-NN, l’algorithme BestBinFirst n’est pas adapté, et nous ne l’incluerons donc plus dans les tests avec les autres algorithmes présentés dans le paragraphe suivant.

4.4.2 Comparaison avec d’autres algorithmes

Nous nous intéressons désormais aux résultats de l’algorithme HASHDIM par rapport aux performances d’autres algorithmes r-NN listés en introduction

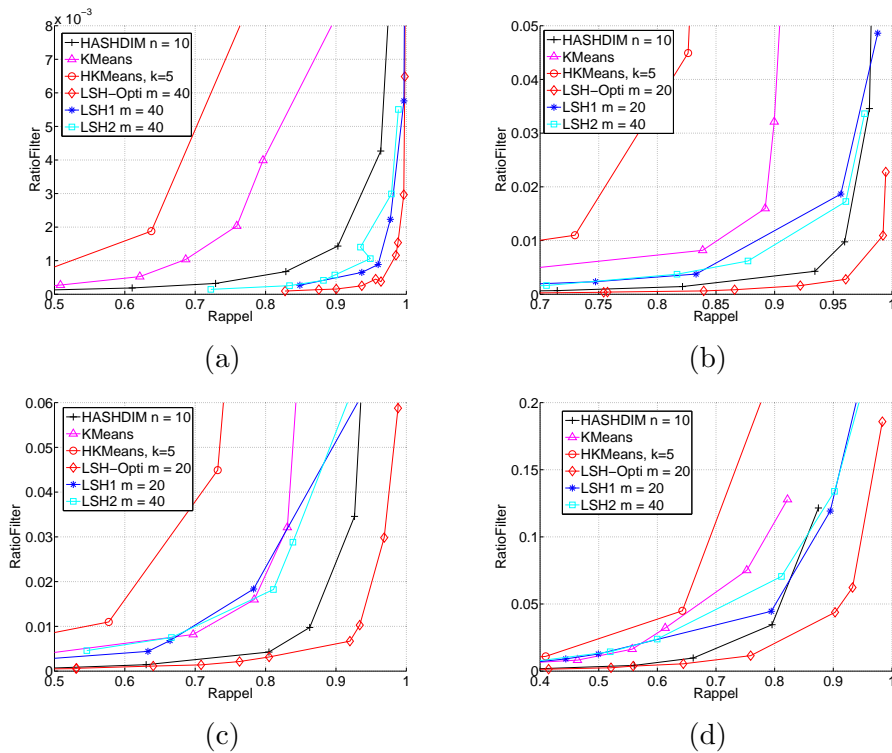


FIGURE 4.12 – Comparaison de l’algorithme HASHDIM avec d’autres algorithmes, pour un seuil de : (a) 150, (b) 200, (c) 250 et (d) 300, sur un nuage de 265.000 points.

de la partie 4.4. Sur la figure 4.12, nous présentons ces mesures. La même figure avec les indices pour chaque point de mesure est fournie en annexe (figure 4.18). Pour les courbes LSH, nous avons choisi le nombre de tables qui donne les meilleures performances et c’est le nombre de dimensions hachées qui varie (i.e. paramètre k). Pour l’algorithme par segmentation K-means, les points sont obtenus en faisant varier le nombre de classes. Pour cet algorithme, la classification a été faite sur la base de tests.

Ces courbes nous permettent de tirer plusieurs conclusions :

- Les deux algorithmes LSH1 et LSH2 offrent des performances très similaires. A noter que l’implémentation du LSH2 utilisée ici est la notre. Nous avons aussi testé le package E2LSH fourni par les auteurs de [DIIM04]. Les résultats sont quasi identiques.
- Pour toutes les valeurs de seuil, le LSH-Opti offre les meilleures performances.
- Pour les seuils supérieurs ou égaux à 200, le hachage HASHDIM offre des performances meilleures que le LSH original mais moins bonnes que le LSH-Opti.
- Pour le seuil de 150, HASHDIM est légèrement moins performant que les approches de type LSH.
- HASHDIM est toujours meilleur qu’une recherche basée sur une méthode K-means (à plat ou hiérarchique).

Ces comparaisons montrent que le seul algorithme qui paraît plus intéressant que HAHS-DIM est LSH-Opti. Toutefois, ce dernier nécessite beaucoup plus de mémoire. Par rapport à un LSH-Opti paramétré avec 20 tables, HASHDIM consomme 20 fois moins de mémoire. Cet élément peut être essentiel si cette différence de mémoire nécessite de “swaper” une partie des tables de hachage du LSH du disque dur vers la RAM. Prenons l’exemple d’une base de 200.000 images, chacune générant 1.000 SIFT soit un nuage final de 2.10^8 descripteurs SIFT. Avec 20 tables, le LSH nécessitera $2.10^8 \times 20 \times 8 = 32Go$. Cela nécessite de charger ces tables depuis le disque lors des recherches de voisinages. L’algorithme sera alors très fortement ralenti. Au contraire, notre algorithme ne nécessite que $2.10^8 \times 8 = 1,6Go$ et tient donc en mémoire vive sur une machine récente. Les performances ne seront donc pas ralenties par des accès disque très coûteux en temps.

4.5 Recherche d’images similaires

Dans les tests précédents, nous avons montré les performances obtenues par certains algorithmes de recherche des plus proches voisins r-NN dans des nuages de points SIFT. Ces résultats sont obtenus sous forme de couples *Rappel – RatioFilter*. Ces valeurs sont intéressantes afin de comparer les algorithmes de recherche des plus proches voisins entre eux. Toutefois, il est nécessaire d’évaluer les algorithmes étudiés dans le cadre applicatif qui est notre motivation initiale. Dans cette partie, nous allons étudier les résultats obtenus avec plusieurs méthodes dans le cadre de la recherche d’images similaires.

Pour cela, nous intégrons les algorithmes de recherche de voisins dans la méthode de recherche d’images similaires décrites en 2.4.1 (i.e. algorithme SIFT+NN+AFF). C’est-à-dire que dans un premier temps, toutes les correspondances sont recherchées entre les descripteurs de l’image requête et les descripteurs de la base. Ensuite, à partir de ces correspondances, une étape de vérification affine est effectuée.

4.5.1 Résultats avec notre hachage

Nous présentons ici les résultats des tests de recherche d’images similaires en utilisant comme algorithme de recherche des voisins le hachage proposé, avec un seuil de distance de 200. Ces résultats sont présentés dans le tableau 4.2. La première conclusion de ce tableau est que les descripteurs utilisés ne sont pas adaptés à la transformation *INCLINAISON15_15* et que celle-ci représente la grande majorité des images que l’algorithme manque. Nous ne la prendrons donc pas en compte dans l’analyse des performances. Pour améliorer les performances sur cette transformation, il faudrait utiliser un descripteur invariant par transformation affine (e.g. MSER [MCMP02] ou Hessian Affine [MS02]). Ensuite, les deux transformations qui ne sont parfois pas retrouvées sont CROP90 et EMBOSS. La transformation CROP90 pose aussi problème dans l’article [FS07] où les auteurs obtiennent un rappel de 0.66 lors des requêtes concernant 3 transformations CROP90, CROP50 et CROP40 (qu’ils regroupent sous le nom de “severe crop”). Dans ces trois transformations, les deux dernières

sont beaucoup plus faciles que la première, nous pouvons alors faire l’hypothèse que la majorité des images manquées par leur algorithme pour ce groupe de transformations sont dues à CROP90. Dans [KSH04], les valeurs de rappel et précision ne sont données que globalement et nous ne pouvons donc pas comparer par transformation. Et dans [LAJA06], les auteurs n’ont pas utilisé la transformation CROP90. Dans ce dernier article, la transformation de recadrage la plus difficile est CROP75, pour laquelle ils obtiennent un rappel de 1. Dans notre cas, pour cette transformation, la majorité de nos tests donnent aussi un rappel de 1. La transformation EMBOSS est aussi difficile. Parmi les trois articles comparés, seul l’article de [LAJA06] l’utilise. Dans cet article, il s’agit de la transformation la plus souvent manquée par l’algorithme (leur rappel pour cette transformation est de 0.8 en utilisant leur descripteur Eff^2 , ce qui correspond pour notre test à manquer au maximum 10 images sur 50). On voit dans le tableau 4.2 que nous obtenons des scores similaires pour cette transformation pour de nombreux couples de paramètres.

Au niveau du temps de calcul, nous remarquons que le temps nécessaire pour trouver les voisins des descripteurs d’une image requête varie environ entre 4.500ms et 30ms. Ces temps sont très intéressants puisque le temps de l’algorithme de recherche exacte linéaire est de 167.418ms (voir tableau 2.3). Si on prend comme comparaison notre hachage avec le couple de paramètres $n, k = 12, 4$ qui manque en tout 14 images comme la recherche linéaire, son temps de recherche des voisins est de 1604ms par image soit un algorithme 104 fois plus rapide. Notre algorithme est aussi 3 fois plus rapide que notre implémentation de la recherche linéaire sur GPU. Si l’on accepte de manquer quelques images par rapport à la recherche linéaire, notamment parmi les trois transformations difficiles, alors il est possible d’atteindre des gains nettement supérieurs. Par exemple, avec les paramètres $n, k = 12, 10$, le rappel n’est plus que de 0.967 (nous ne manquons que 2 images en dehors des trois transformations les plus difficiles), et dans ce cas, le temps de recherche des voisins est de 62ms soit un algorithme 2700 fois plus rapide que la recherche linéaire sur CPU.

Ces résultats montrent l’intérêt de l’algorithme proposé. Par rapport à une implémentation d’une recherche linéaire sur GPU, notre algorithme est plus rapide, ne nécessite aucun matériel spécifique et obtient des résultats similaires en termes de recherche d’images. Par rapport à un algorithme LSH, notre algorithme nécessite beaucoup moins de mémoire. Par exemple, dans [KSH04] et [FS07], les auteurs utilisent 20 tables de hachage. Si l’on considère qu’un point est enregistré dans une table de hachage par 2 entiers 32 bits (un index et un checksum), un nuage d’un million de descripteurs nécessite $10^6 \times 2 \times 4 \times 20 = 160Mo$. Pour le même nuage, notre algorithme n’utilise qu’une table et consomme $10^6 \times 2 \times 4 = 8Mo$ en mémoire. En pratique, ce gain est très important puisque notre algorithme permet de travailler en mémoire vive pour de très gros nuages alors que le LSH impose très rapidement de stocker les index sur le disque dur.

A noter que dans l’algorithme testé, une fois les descripteurs filtrés par le hachage, la fonction distance utilisée pour vérifier que les points sont de vrais voisins est la fonction linéaire où le boucle de calcul a été complètement déroulée

(voir algorithme 2). En utilisant la distance partielle, présentée en 2.6.3, nous pourrions obtenir un léger gain.

Afin de comparer avec notre implémentation de LSH, nous montrons les performances de cet algorithme dans le tableau 4.3. Contrairement à notre hachage, avec le LSH, aucun paramétrage n’obtient le même rappel que la recherche linéaire. Pour des rappels inférieurs, les temps de recherche des plus proches voisins sont similaires à ceux obtenus avec le hachage proposé. Par exemple pour $m, k = 20, 100$, le LSH obtient un rappel de 0.986 avec un temps de recherche des voisins de 920ms. Le hachage proposé obtient un rappel similaire (0.987) pour le couple $n, k = 12, 7$ avec un temps de 406ms. Au vu des différences possibles d’implémentation, nous pouvons affirmer que les temps d’exécution sont similaires à performances égales. Le principal avantage de notre méthode est alors la faible consommation mémoire.

Résultats sur DB_{32k} Afin de confirmer que notre algorithme peut aussi être utilisé sur des bases de taille moyenne, nous mesurons ses performances sur la base DB_{32k} . Nous utilisons uniquement le couple $n, k = 12, 10$, qui obtient un bon couple vitesse/rappel sur la petite base (200 est utilisé comme seuil de distances). Nous obtenons le même rappel et la même précision que sur la base DB_{4k} . Concernant le rappel, il est normal d’obtenir le même résultat car la taille de la base n’affecte pas les voisins SIFT retrouvés. Nous pourrions toutefois nous attendre à ce qu’en rajoutant des images, la précision diminue, ce qui n’est pas le cas. Cela confirme que la méthode de vérification affine permet de ne conserver que les images qui sont vraiment similaires. Au niveau des performances en temps, le temps de recherche des voisins SIFT est de 187ms et le temps de vérification affine de 1.600ms. On voit que ces temps augmentent peu par rapport aux temps obtenus pour la petite base (respectivement 62 ms et 1.493ms).

4.6 Tests sur une grande base

Dans ce paragraphe, nous présentons une implémentation dédiée aux très grandes bases.

Pour notre base de 510.000 images, si l’on fixe un maximum de 256 SIFT par image, il faut alors $510.000 \times 256 \times 144 = 18.8Go$. Une telle quantité de données ne tient pas dans la RAM d’une machine classique. Avec l’algorithme tel que décrit dans les paragraphes précédents, cela impose de charger depuis le disque les descripteurs SIFT, à la demande, pour calculer la distance euclidienne et vérifier si ce sont bien des voisins du point requête. Nous proposons donc d’ignorer cette étape. C’est-à-dire que nous ne filtrons pas les points en fonction de leur distance au point requête. Tous les points de la base appartenant à la case de la table de hachage du point requête sont déclarés voisins. L’inconvénient est que nous allons accepter des points qui ne sont pas de vrais voisins (au sens de la distance euclidienne inférieure à un seuil). En contrepartie, nous pouvons ne pas charger en mémoire les descripteurs SIFT et l’algorithme est donc plus rapide.

n	k	$\binom{n}{k}$	R	P	TNN (ms)	TRAN(ms)	INCLINAISON15.15	CROP90	EMBOSS	MEDIAN3	INCLINAISON15	CROP75	ROTCROP1050	SCALE12
6	2	15	0.994	1	4324	1885	9	4	2					
6	3	20	0.991	1	998	1823	15	6	2					
6	4	15	0.985	1	357	1696	23	14	3					
6	5	6	0.977	1	114	1673	33	22	6					
8	2	28	0.995	1	5846	1926	8	3	2					
8	3	56	0.994	1	1599	1899	11	3	2					
8	4	70	0.992	1	692	1859	14	6	2					
8	5	56	0.985	1	246	1792	24	9	3					
8	6	28	0.980	1	129	1627	31	15	5	1				
8	7	8	0.969	1	69	1406	39	28	14	1		1		
10	6	210	0.988	1	267	1798	17	10	4					
10	7	120	0.983	1	144	1720	28	13	4					
10	8	45	0.973	1	94	1603	38	21	10	1	1			
10	9	10	0.959	1	38	1602	44	32	19	1	6	2	2	1
12	4	495	0.995	1	1604	2046	10	3	1					
12	5	792	0.994	1	801	2010	12	3	1					
12	6	924	0.990	1	587	1951	16	6	3					
12	7	792	0.987	1	406	1876	20	10	4					
12	8	495	0.984	1	265	1779	25	13	4					
12	9	220	0.975	1	155	1647	36	22	8		1			
12	10	66	0.967	1	62	1493	44	28	14		1	1		
14	6	3003	0.993	1	1466	2003	13	4	2					
14	7	3432	0.991	1	1443	2033	16	6	3					
14	8	3003	0.988	1	1246	1970	18	9	4					
14	9	2002	0.984	1	962	1884	25	14	6					
14	10	1001	0.975	1	423	1709	34	22	8		1			
14	11	364	0.970	1	184	1636	41	24	12	1	1	1		
14	12	91	0.958	1	66	1424	47	33	21	1	6	2		

TABLE 4.2 – Résultats de la recherche d’images similaires en utilisant HASHDIM comme algorithme de recherche de voisins dans l’algorithme SIFT+NN+AFF, sur la base DB_{4000} . n est le nombre de dimensions utilisées pour calculer les multiples clés de hachage d’un point requête. k est le nombre de dimensions utilisées pour calculer la clé de hachage unique d’un point de la base. Les titres de colonnes R et P signifient respectivement Rappel et Precision, TNN est le temps par image requête pour retrouver les voisins de ses descripteurs, TRAN est le temps par image requête pour effectuer la vérification affine. Enfin, pour chacune des dernières colonnes, le nombre indiqué est le nombre d’images manquées parmi les 50 à retrouver pour une transformation donnée. Pour une meilleure lisibilité, les cases vides ont pour valeur 0. Pour toutes les transformations qui n’apparaissent pas dans ce tableau, nous retrouvons toujours toutes les 50 images.

m	k	R	P	TNN (ms)	TRAN(ms)	<i>INCLINAISON15_15</i>	CROP90	EMBOSS	MEDIAN3	INCLINAISON15	CROP75	ROTCROP1050	SCALE12
20	60	0.994	1	9623	2132	10	5	2					
20	80	0.992	1	2601	2077	12	7	2					
20	100	0.986	1	920	1965	23	10	3					
20	120	0.983	1	646	1910	29	9	4	1				
20	140	0.978	1	508	1808	36	17	7	1				
20	160	0.974	1	426	1731	35	20	10	2	1			
20	180	0.963	1	316	1632	42	27	17	2	7	1	0	1
20	200	0.957	1	272	1580	44	29	19	2	12	1	1	3

TABLE 4.3 – Résultats de la recherche d’images similaires en utilisant le LSH1 sur la base DB_{4000} . m est le nombre de tables de hachage utilisées. k est le nombre d’hyperplans aléatoires utilisés pour générer chacune des fonctions de hachage. Les titres de colonnes R et P signifient respectivement Rappel et Precision, TNN est le temps par image requête pour retrouver les voisins de ses descripteurs, TRAN est le temps par image requête pour effectuer la vérification affine. Enfin, pour chacune des dernières colonnes, le nombre indiqué est le nombre d’images manquées parmi les 50 à retrouver pour une transformation donnée. Pour une meilleure lisibilité, les cases vides ont pour valeur 0. Pour toutes les transformations qui n’apparaissent pas dans ce tableau, nous retrouvons toujours toutes les 50 images.

File_Id	tx	ty	nb descriptors	fileName
1
2
3

TABLE 4.4 – Table listant les fichiers en base.

$SIFT_1$				$SIFT_2$...
$x_1^1(16\text{bits})$	$y_1^1(16\text{bits})$	$\theta_1^1(16\text{bits})$	$sc_1^1(16\text{bits})$	$x_2^1(16\text{bits})$	$y_2^1(16\text{bits})$	$\theta_2^1(16\text{bits})$	$sc_2^1(16\text{bits})$...
x_1^2	y_1^2	θ_1^2	sc_1^2	x_2^2	y_2^2	θ_2^2	sc_2^2	...

TABLE 4.5 – Table des descripteurs.

Pour ces tests, nous fixons un maximum de 256 SIFT par image. Sur une machine à processeur Intel QuadCore Xeon 2.66GHz, en traitant plusieurs images en parallèle, le temps moyen de calcul des descripteurs est de 0.34s par image. Le temps total sur les 510.000 images est d'environ 48 heures. Cela génère 128 millions de points, qui occupent 18Go sur le disque. On voit bien qu'il est impossible sur une machine 32 bits de charger tous ces points en mémoire vive.

Notre implémentation utilise plusieurs tables pour stocker le système d'indexation. Une première table liste les images présentes en base. La table 4.4 illustre son contenu. Ensuite, nous enregistrons dans une table tous les points d'intérêts des descripteurs SIFT de la base. C'est-à-dire que comme notre méthode n'utilise pas le descripteur lui-même (le vecteur à 128 dimensions), mais seulement sa position, son orientation et son échelle, seuls ces 4 paramètres sont enregistrés. Pour optimiser l'espace mémoire nécessaire, ces 4 valeurs sont codées sur un total de 64 bits (16 bits pour chaque champ). La table 4.5 illustre cette représentation. Pour notre base, cette table occupe 1.02 Go. Enfin, nous enregistrons aussi la table de hachage dans un fichier. Chaque élément de la table est une structure de 64 bits. Les 24 premiers bits enregistrent l'id de l'image auquel fait référence cet élément. Les 8 bits suivants codent l'index du descripteur parmi les descripteurs de cette image. Enfin, les 32 derniers bits permettent de stocker le checksum de cet élément. Cette table occupe aussi 1Go pour la base de 510.000 images. La construction de cette indexation (avec $n, k = 10, 12$) prend environ deux heures sur notre machine.

Avec cette implémentation, il faut une minute pour que le moteur de recherche charge ces tables. Cela requiert environ 2Go de RAM. Nous pouvons donc exécuter des recherches sans effectuer aucun accès disque.

4.6.1 Résultats

Nous reprenons le protocole de tests décrit précédemment (dans lequel nous effectuons 50 requêtes et devons retrouver les 53 déformations pour chaque requête). Le temps moyen de recherche des descripteurs voisins est de 248ms. Le temps de vérification affine est de 445ms par image. Nous obtenons un rappel de 0.966 et une précision de 0.970.

Il faut comparer ces valeurs à celles obtenues pour le test où nous vérifions la distance euclidienne entre les descripteurs. Dans la table 4.2, nous avons utilisé un seuil de 200. Nous obtenions, sur une base de 4000 images, pour $n, k = 12, 10$ un rappel de 0.967, une précision de 1, des temps de 62ms pour la recherche r-NN et 1493ms pour la vérification affine.

Concernant la recherche r-NN, le temps est multiplié par 7 alors que la taille de la base est multipliée par plus de 120. Ceci est dû au fait que pour cette grande base, aucun calcul de distance euclidienne n'est effectué. Concernant le temps de vérification affine, pour effectuer ce test sur une grande base, nous avons optimisé l'implémentation de la vérification affine et il n'est donc pas possible de comparer les résultats avec ceux de 4.2.

Sur la très large base, le rappel est quasiment identique. Par contre, la précision est inférieure. Il y a deux raisons à cela. D'abord, en ne vérifiant pas la distance, nous acceptons plus de fausses correspondances, et donc nous retrouvons des images alors qu'il ne le faudrait pas. Ensuite, la base est beaucoup plus grande, et la probabilité qu'il y ait des images réellement similaires aux images requêtes est plus élevée. En vérifiant manuellement, nous nous apercevons que cette seconde raison est la source principale des erreurs. En téléchargeant aléatoirement des images sur Internet, certaines l'ont été plusieurs fois. Si c'est le cas pour une image utilisée comme requête dans notre test, nous allons retrouver les 53 images modifiées, mais aussi une copie de la requête téléchargée sur un autre site. En pratique, il s'agit de la plupart des cas. Il faudrait supprimer ces images de la base pour obtenir une meilleure mesure de précision. Il serait possible d'utiliser les résultats de la recherche pour éliminer ces doublons.

4.7 Résultats pour l'application d'analyse de linéaires

L'une des applications qui motivait nos travaux était de retrouver les produits présents sur une image de linéaire. Pour évaluer notre algorithme, nous avons besoin de deux éléments : une vérité terrain et une métrique qui nous donne le score d'un résultat par rapport à cette vérité terrain.

Dans nos travaux, nous n'avons pas développé d'algorithme de calcul de métrique pour ce problème. De plus, il s'agit en soi d'un problème complexe car en plus de devoir vérifier que les produits retrouvés sont réellement ceux sur l'image, il faut aussi juger de la qualité de leurs positions estimées. Si un produit est trouvé, à une position proche du vrai produit, mais légèrement décalé, s'agit-il d'une erreur ou est-ce tolérable ? Sur ce point, la méthode d'évaluation reste à définir.

Toutefois, pour estimer les performances de notre algorithme, nous avons effectué une évaluation manuelle de celui-ci. Dans le paragraphe suivant, nous présentons cette évaluation. Ensuite, nous l'appliquons à notre algorithme sur plusieurs images de linéaire.

Critère	Valeur	Remarque
NbToFind	45	Dans la base, nous ne disposons pas des tranches des produits.
NbFound	43	
NbOK	41	
NbOther	1	En D, le produit trouvé n'est pas le bon.
NbFalse	1	En C, un produit est trouvé alors qu'il n'y en a pas.
NbMissed	3	2 produits manqués en A et 1 en B.

TABLE 4.6 – Exemple d'évaluation sur le résultat de la figure 4.13.

4.7.1 Protocole d'évaluation

L'intérêt principal de cette évaluation est d'estimer le gain en temps qu'apportent les algorithmes de recherche des plus proches voisins. Au niveau qualitatif, notre méthode est perfectible. Nous pourrions par exemple détecter les étagères. Il serait aussi utile d'introduire des contraintes métier : par exemple, deux produits identiques ne peuvent pas être séparés par un autre produit. Pour ces raisons, les résultats qualitatifs sont à prendre avec précaution. Mais les temps de recherche des plus proches voisins donnent une bonne idée du temps de traitement nécessaire à notre algorithme.

Après l'analyse d'un linéaire, notre algorithme retourne la liste des produits trouvés sur ce linéaire ainsi que la transformation affine entre l'image de la base et sa position sur le linéaire. Pour chaque image de linéaire analysée, nous utilisons les critères suivants :

- NbToFind : Nombre total de produits à trouver sur le planogramme.
- NbFound : Nombre total de produits retournés par l'algorithme (qui peuvent être incorrects).
- NbOK : Nombre de produits trouvés par notre algorithme à leur position correcte.
- NbOther : Nombre de produits du linéaire pour lesquels un produit dans la base a bien été trouvé, mais ce n'est pas le bon.
- NbFalse : Nombre de produits trouvés à des positions du linéaire où il n'y a aucun produit.
- NbMissed : Nombre de produits manqués par l'algorithme. C'est-à-dire qu'à la position d'un produit sur le linéaire, on ne trouve rien.

A partir de cette liste, il est difficile d'obtenir une métrique simple. Nous allons donc simplement, pour certains couples de paramètres, mesurer ces valeurs manuellement.

Sur les figures 4.13, nous montrons une photo de linéaire ainsi que le résultat de notre algorithme. Les critères ci-dessus associés à ce résultat sont expliqués dans le tableau 4.6.

4.7.2 Données pour l'évaluation

Pour l'évaluation, nous utilisons 3 photos de linéaire : celle de la figure 4.13 ainsi que les deux de la figure 4.14. Pour le linéaire de fromages, la base est constituée de 68 imageries de produits. La base pour le linéaire Barilla contient



(a)



(b)

FIGURE 4.13 – (a) : Photo d'un linéaire. (b) : Résultats de notre algorithme sur ce linéaire. Sur cette seconde image, les imagettes sont des images de la base de recherche que nous avons copiées à leur positions respectives trouvées (selon leurs matrices affines estimées). Voir la table 4.6 pour les explications liées aux lettres A, B, C, D.

40 images et celle pour le linéaire Biscuits contient 1173 produits.

4.7.3 Résultats

Pour chacun des linéaires, nous lançons l’application d’analyse avec plusieurs algorithmes de recherche des plus proches voisins. Les résultats sont présentés dans la table 4.7. Sur cette table, la colonne qui nous importe le plus est celle du temps de recherche des plus proches voisins (noté Temps NN). Nous cherchons un algorithme qui soit le plus rapide possible tout en ne dégradant pas trop les résultats par rapport à une recherche des plus proches voisins exacts. Les mesures concernant l’analyse qualitative du linéaire sont à prendre avec précaution car l’algorithme est très largement perfectible. Concernant la partie Temps RANSAC, cette valeur est également indicative. L’algorithme n’est pas optimisé, et nous n’avons pas précisément cherché les meilleures valeurs des paramètres (nombre de jets du RANSAC, nombre de paires de points minimum pour chercher une matrice...).

Le premier constat intéressant est qu’avec des algorithmes optimisés de recherche des plus proches voisins, nous obtenons des temps qui sont tout à fait acceptables pour un utilisateur (quelques secondes). L’algorithme le plus rapide est celui qui utilise une segmentation K-means des points avec $K=1600$ et pour lequel le GPU est utilisé (pour attribuer la classe d’un point). Tout en étant beaucoup plus rapide que l’algorithme de recherche linéaire avec distance partielle (285 fois plus rapide pour le linéaire “Biscuit”), nous ne manquons que quelques produits (on pourrait alors imaginer une seconde passe plus précise uniquement aux endroits où aucun produit n’a été trouvé).

Un élément également très important est le temps nécessaire pour construire cette indexation. En effet, on peut imaginer que l’application va permettre à l’utilisateur de filtrer les produits dans lesquels faire la recherche selon des mots-clés. Par exemple, travaillant sur le linéaire Barilla avec une base d’images contenant toutes les références de pâtes, l’utilisateur a tout intérêt à limiter ses recherches aux produits Barilla. Dans ce cadre, la construction de l’indexation doit être la plus rapide possible. Dans la table 4.8, nous présentons les temps de construction en fonction de l’algorithme utilisé, pour les trois bases considérées. Pour les algorithmes linéaires, il n’y a pas d’indexation et le temps n’est pas précisé (pour la version GPU, le temps de transfert des points de la mémoire centrale à la mémoire GPU est négligeable). L’algorithme HK-GPU avec $k=1600$ est une option qui reste parmi les plus intéressantes. Notre hachage avec $(n, k) = (12, 2)$ ou bien le LSH sont à peine plus rapides dans certains cas.

En conclusion, si la machine utilisée dispose d’un GPU, la meilleure solution semble être l’algorithme par segmentation K-means. Toutefois, il est intéressant de considérer une machine sans GPU, notamment pour pouvoir utiliser l’application sur une machine portable, par exemple directement en magasin. Dans ce cas, l’algorithme par segmentation K-means devient nettement plus lent. Notre hachage avec $n, k = 12, 2$ ou le LSH offrent les meilleures performances. Pour les départager, il faudrait une analyse quantitative plus précise. Toutefois, l’avantage principal de notre algorithme reste sa faible consommation mémoire.



(a) Linéaire Biscuits



(b) Linéaire Barilla

FIGURE 4.14 – Deux photos de linéaire utilisées pour l'évaluation.

Algo NN	Linéaire	Temps NN(s)	Temps RANSAC(s)	Temps Spatial(s)	NbToFind	NbFound	NbOK	NbOther	NbFalse	NbMissed
Linéaire, Boucle Déroulée	Fromages	55.2	5.3	1.4	45	38	37	0	1	7
	Barilla	91.4	24	5.2	36	45	31	1	10	1
	Biscuits	1324.1	57	5.6	81	65	58	7	1	16
Distance Partielle	Fromages	19.8	5.3	1.4	45	38	37	0	1	7
	Barilla	25.3	24	5.2	36	45	31	1	10	1
	Biscuits	743.4	57	5.6	81	65	58	7	1	16
GPU	Fromages	2.5	5.3	1.4	45	38	37	0	1	7
	Barilla	4.0	24	5.2	36	45	31	1	10	1
	Biscuits	52.2	57	5.6	81	65	58	7	1	16
HASHDIM, 12,2	Fromages	2.1	2.3	1.3	45	36	36	0	0	9
	Barilla	3.8	8.2	5.2	36	45	33	3	8	0
	Biscuits	64.5	26.8	4.9	81	64	56	6	2	17
HASHDIM, 12,6	Fromages	1.9	0.9	1.2	45	33	33	0	0	12
	Barilla	3.5	6.6	5.2	36	45	30	4	8	0
	Biscuits	10.3	10.6	4.0	81	54	50	4	0	23
HK-GPU, k=1600	Fromages	0.9	1.4	1.3	45	35	35	0	0	10
	Barilla	1.1	10.4	5.3	36	48	33	2	12	0
	Biscuits	2.6	31.3	4.5	81	61	53	7	1	19
HK-CPU, k=1600 CPU	Fromages	38.1	1.4	1.3	45	35	35	0	0	10
	Barilla	93.3	10.4	5.3	36	48	33	2	12	0
	Biscuits	94.8	31.3	4.5	81	61	53	7	1	19
HK-GPU, k=4096	Fromages	2.2	0.8	1.3	45	37	37	0	0	8
	Barilla	2.4	5.2	5.6	36	48	35	1	11	0
	Biscuits	3.2	21.2	4.2	81	56	48	7	0	21
LSH opti, m,k=20,80	Fromages	2.1	2.9	1.3	45	36	36	0	0	9
	Barilla	3.8	19.6	5.2	36	44	31	2	9	0
	Biscuits	27.6	47.4	4.3	81	57	47	7	1	22

TABLE 4.7 – Evaluation de l’application d’analyse de linéaire, pour plusieurs algorithmes de recherche des plus proches voisins. HASHDIM signifie que notre hachage est utilisé. HK signifie une segmentation par K-means. Pour cette méthode, HK-CPU signifie que seul le CPU est utilisé alors que HK-GPU utilise le GPU pour calculer à quelle classe appartient un point.

Algo NN	Linéaire	Nb prods in DB	Nb descr in DB	Temps Build(ms)
HASHDIM, n,k=12,2	Fromages	68	28976	927
	Barilla	40	19535	613
	Biscuits	1173	261288	7794
HASHDIM, n,k=12,6	Fromages	68	28976	5076
	Barilla	40	19535	3696
	Biscuits	1173	261288	43958
HK-GPU, k=1600	Fromages	68	28976	1474
	Barilla	40	19535	539
	Biscuits	1173	261288	4837
HK-CPU, k=1600	Fromages	68	28976	62000
	Barilla	40	19535	42028
	Biscuits	1173	261288	562282
HK-GPU, k=4096	Fromages	68	28976	2893
	Barilla	40	19535	1261
	Biscuits	1173	261288	11512
LSH opti, m,k=20,80	Fromages	68	28976	698
	Barilla	40	19535	459
	Biscuits	1173	261288	5660

TABLE 4.8 – Temps de construction de la structure d’indexation en fonction de l’algorithme utilisé, pour trois bases d’images différentes.

4.8 Conclusion

Dans ce chapitre, nous avons deux objectifs. Le premier était de tester s’il était possible de modifier les fonctions de hachage du LSH1 pour en améliorer les performances sur des points SIFT. Le second était de proposer un algorithme de recherche r-NN, aussi rapide que le LSH, mais consommant beaucoup moins de mémoire.

Concernant le premier objectif, nous avons introduit une modification de l’algorithme LSH en adaptant le choix des hyperplans aux données SIFT. Dans nos tests, c’est cet algorithme qui obtient les meilleures performances. Toutefois, cet algorithme, même s’il réduit le nombre de tables de hachages nécessaire par rapport au LSH normal, nécessite toujours une grande quantité de mémoire.

Pour réduire ce besoin de mémoire qui peut devenir critique pour des très grands nuages de points, nous avons proposé un nouvel algorithme de hachage que nous avons appelé HASHDIM. Cet algorithme, légèrement moins performant que le LSH modifié, est meilleur que les autres algorithmes r-NN testés et consomme beaucoup moins de mémoire que le LSH.

Dans le cadre de recherche d’images similaires, en utilisant l’algorithme SIFT+NN+AFF, nous avons montré que les performances du LSH et de HASHDIM sont très similaires en temps d’exécution et en qualité de résultats. Encore une fois, l’avantage principal de l’algorithme HASHDIM est sa faible consommation mémoire. Cet élément est particulièrement important si le nuage de points devient très grand et que la quantité de mémoire centrale n’est plus suffisante pour le LSH.

Nous avons aussi proposé une implémentation de notre algorithme pour une grande base de 510.000 images. Pour faire tenir l’indexation en mémoire centrale, nous avons modifié l’algorithme de requête de HASHDIM. C’est-à-

dire que l'algorithme ne calcule aucune distance euclidienne entre descripteurs SIFT et n'a donc pas besoin de conserver en mémoire ces descripteurs à 128 dimensions. Cet algorithme permet de lancer des requêtes dans une base de 510.000 images en moins d'une seconde.

Une possibilité pour accélérer encore cet algorithme (qui ne calcule pas de distances euclidiennes) et traiter des bases plus grandes serait d'appliquer la vérification affine à une petite liste d'images potentiellement similaires à la requête. Par exemple, il est possible de ne considérer que les n images qui ont le plus de correspondances avec l'image requête. Coupler ces deux choix (pas de calcul de distances euclidiennes et sélection d'une "short-list") est en fait une méthode de recherche par "Bag-Of-Features". Dans le chapitre suivant, nous reprenons cette idée plus en détails et montrons comment utiliser HASHDIM avec le formalisme de la recherche par "Bag-Of-Features".

4.9 Appendice

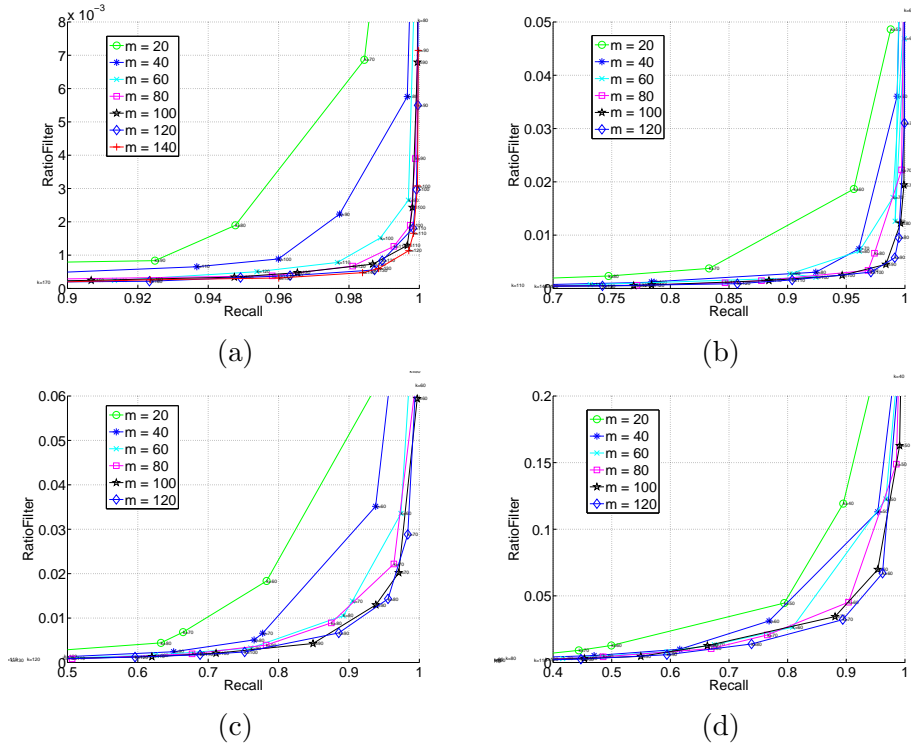


FIGURE 4.15 – Résultats du hachage LSH1 pour des seuils de (a) 150, (b) :200, (c) 250, (d) 300. Le paramètre m est le nombre de tables utilisées. Sur une courbe, le nombre d'hyperplans utilisés k varie.

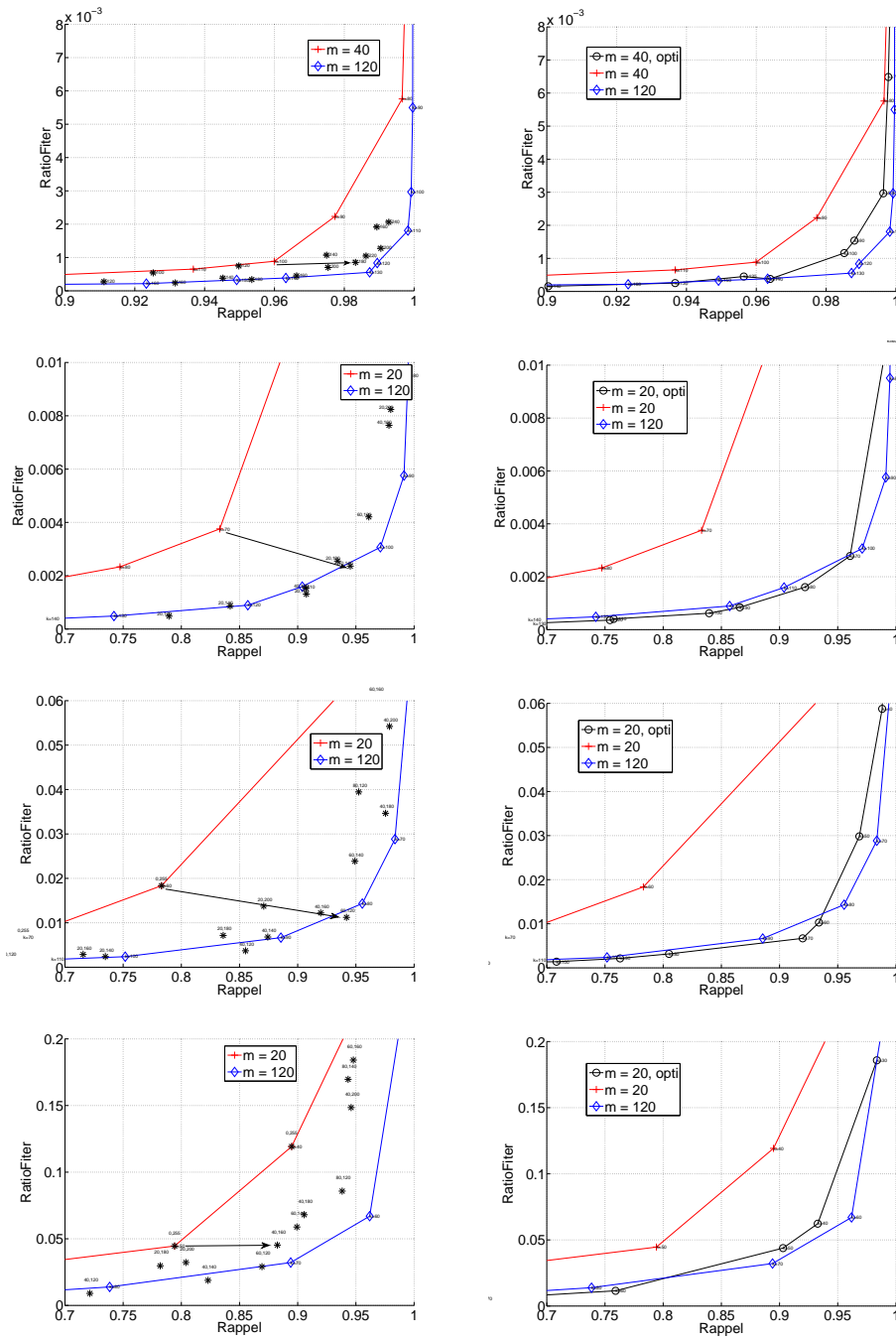
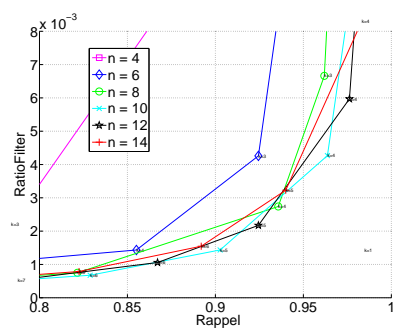
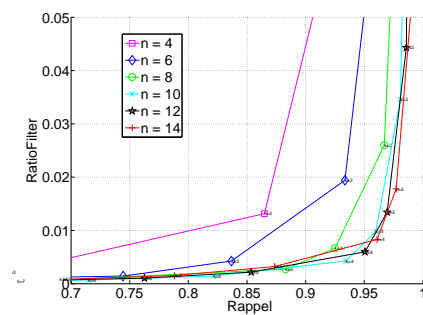


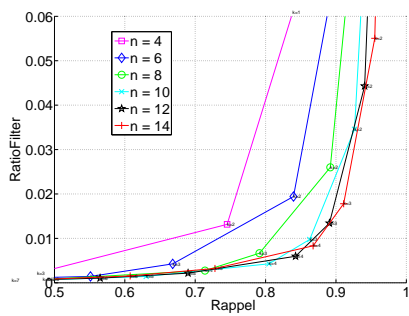
FIGURE 4.16 – Résultats du hachage LSH1 et LSH-Opti pour des seuils de (a) 150, (b) :200, (c) 250, (d) 300. Sur la colonne de gauche, les deux courbes correspondent à des résultats du LSH1. Chaque point correspond aux performances d'un paramétrage S_{min}, S_{max} de LSH-Opti. Sur la colonne de droite, nous reprenons les deux courbes du LSH1 et nous rajoutons une courbe des performances du LSH-Opti pour le paramétrage S_{min}, S_{max} que nous avons choisi (i.e. correspondant à la flèche de la figure de la colonne de gauche).



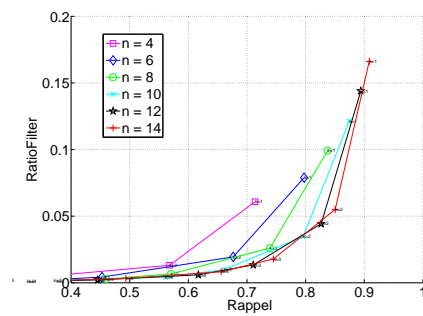
(a)



(b)

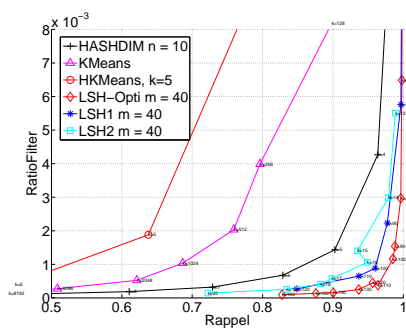


(c)

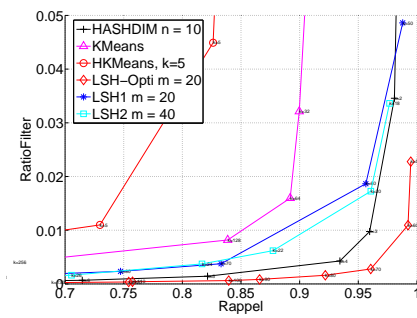


(d)

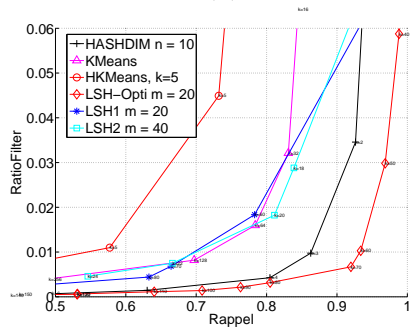
FIGURE 4.17 – Résultats du hachage HASHDIM pour des seuils de (a) 150, (b) :200, (c) 250, (d) 300.



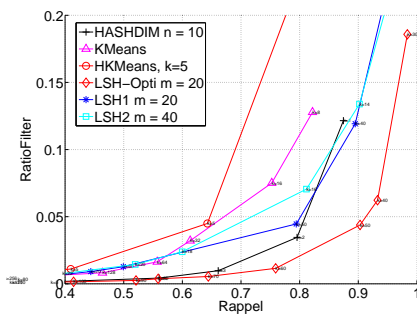
(a)



(b)



(c)



(d)

FIGURE 4.18 – Comparaison de l’algorithme HASHDIM avec d’autres algorithmes r-NN, pour un seuil de : (a) 150 et (b) 200, (c) 250 et (d) 300.

Chapitre 5

Utilisation du hachage HASHDIM dans une recherche Bag-Of-Features

Dans le chapitre précédent, nous avons proposé et utilisé un nouvel algorithme r-NN pour faire de la recherche d'images similaires dans le cadre de l'algorithme SIFT+NN+AFF. Nous avons vu que pour une base de 510.000 images, il était nécessaire d'adapter HASHDIM et de ne pas calculer de distance euclidienne. Il s'agit là d'une illustration des travaux de Jegou et al [JDS08] montrant qu'il est possible d'utiliser n'importe quel algorithme de recherche des plus proches voisins pour faire de la recherche d'images similaires par la méthode de Bag-Of-Features, introduite dans les travaux de Sivic et al. [SZ03].

Dans ce chapitre, nous reprenons ces deux articles et montrons comment intégrer l'algorithme HASHDIM dans une recherche Bag-Of-Features. Ensuite, nous comparons les résultats de cette approche avec les méthodes plus classiques de Bag-Of-Features, pour lesquelles l'algorithme de recherche de voisins utilise un apprentissage K-means. Nous montrons que pour le cas de la recherche d'images similaires, HASHDIM est plus efficace que ces vocabulaires à base de K-means. Dans cette partie, notre contribution est d'intégrer l'algorithme HASHDIM dans une recherche par Bag-Of-Features et de comparer les résultats à d'autres vocabulaires plus classiques (Auclair et al. [ACV09]).

Sommaire

5.1	Approche Bag-Of-Features	124
5.2	Lien entre vocabulaire visuel et algorithme de recherche des plus proches voisins	125
5.3	Utilisation de notre hachage HASHDIM dans une recherche BOF	126
5.4	Taille du vocabulaire	128
5.5	Choix de la fonction de coût	128
5.6	Impact des paramètres (n, k)	130
5.7	Comparaison avec des vocabulaires K-means	131
5.8	Test sur la base de 510.000 images	134
5.9	Tests sur la base Nister	136

5.10 Conclusion	139
----------------------------------	------------

5.1 Approche Bag-Of-Features

Le principe initial de cette méthode, notée BOF dans les paragraphes suivants, est décrit dans un article intitulé “Video Google” de Sivic et al. [SZ03]. Les auteurs s’inspirent d’une méthode d’indexation de documents textuels utilisant les vecteurs de fréquences des mots (i.e. un modèle de Salton [SM86]). Dans ce cadre, chaque document est décrit par un unique vecteur qui a pour dimension le nombre de mots dans le vocabulaire utilisé. La i^{eme} composante de ce vecteur est la fréquence du mot i du vocabulaire dans ce document (par souci de clarté, il s’agit ici d’un modèle simplifié). Lorsque l’on souhaite retrouver les documents proches d’un document requête, le vecteur de fréquence de ce document est calculé et les documents de la base qui ont les vecteurs de fréquence les plus proches sont retournés. On remarque qu’il s’agit d’un algorithme qui fournit une mesure de similarité entre un document requête et ceux d’une base. En général un tel algorithme est utilisé pour sélectionner les k documents les plus similaires à la requête.

En pratique, la recherche n’est pas faite linéairement mais des fichiers inversés (connus sous le nom de “inverted files”) sont utilisés. Chaque fichier inversé correspond à un mot du vocabulaire et contient les références vers tous les documents de la base qui contiennent ce mot. Lors d’une recherche de document similaire, on parcourt uniquement les inverted files correspondant aux mots du document requête. Les scores de chaque document retrouvé sont alors calculés selon une fonction de coût qui peut prendre en compte divers paramètres. L’utilisation des fichiers inversés est très efficace car les vecteurs de fréquence contiennent majoritairement des zéros.

Dans [SZ03], les auteurs présentent une façon de définir un vocabulaire visuel à partir de descripteurs locaux d’images. Pour cela, des descripteurs SIFT sont extraits d’une base d’images et sont ensuite clusterisés par un K-means. Les K centres de classes ainsi obtenus définissent un vocabulaire de K mots. Pour calculer le vecteur de fréquences d’une image, on associe à chaque descripteur SIFT le mot le plus proche. Pour retrouver rapidement les vecteurs de fréquences les plus proches d’un vecteur requête dans la base, des fichiers inversés sont utilisés.

Pour chaque image, son vecteur de fréquences est ainsi défini :

$$(f_1, \dots, f_k) \tag{5.1}$$

où f_i est la fréquence des SIFT dont le mot le plus proche est le mot numéro i . C’est-à-dire que si on note h le nombre de vecteurs SIFT de cette image :

$$f_i = \frac{n_i}{h} \tag{5.2}$$

où n_i est le nombre de SIFT de cette image qui sont associés au mot i .

Pour obtenir de meilleurs résultats, il faut pondérer les mots selon leur fréquence dans la base d’images. Pour cela, le schéma tf-idf (term-frequency inverse-document-frequency de [SM86]) est utilisé. C’est-à-dire que les coefficients du vecteur de fréquence sont définis par :

$$w_i = f_i \times \log \frac{N}{N_i} = \frac{n_i}{h} \times \log \frac{N}{N_i} \quad (5.3)$$

où N est le nombre d'occurrences de mots dans la base et N_i est le nombre d'occurrences du mot i dans la base. À noter que ceci n'est pas exactement un schéma tf-idf classique. Il faudrait pour cela choisir $w_i = f_i \times \log \frac{N^{images}}{N_i^{images}}$ avec N^{images} le nombre d'images dans la base et N_i^{images} le nombre d'images dans la base qui contiennent le mot i . Nous n'avons pas testé cette seconde approche car ce n'est pas le schéma classiquement utilisé pour la recherche d'images similaires.

Cette méthode est utilisée sur de très grandes bases pour choisir une liste d'images potentiellement proches de la requête. Toutefois, cet algorithme n'obtient pas une très bonne précision. Pour cela, il est nécessaire de filtrer cette liste, par exemple, en cherchant une relation affine avec l'image requête. Pour l'étape de vérification affine, il est nécessaire d'avoir des correspondances de points. Pour cela, chaque descripteur de l'image requête est considéré en correspondance avec les descripteurs du fichier inversé du mot visuel auquel il appartient. En couplant les avantages de ces deux algorithmes (rapidité sur des très grandes bases pour la recherche BOF et bonne précision pour la vérification affine), on peut faire de la recherche d'images similaires sur de très grandes bases d'images.

5.2 Lien entre vocabulaire visuel et algorithme de recherche des plus proches voisins

Jegou et al. [JDS08] ont montré que l'utilisation d'un vocabulaire correspond à un algorithme de recherche des plus proches voisins. Dans les paragraphes suivants, nous reprenons leur explication et montrons comment utiliser l'algorithme HASHDIM dans une recherche par Bag-Of-Features.

Dans BOF, le score de ressemblance entre une image requête et la j^{eme} image de la base peut être exprimé sous la forme d'un produit scalaire entre deux vecteurs de fréquence. Il peut aussi être exprimé par :

$$s_j = \frac{1}{m_j n} \sum_{i=1}^n \sum_{k=1}^{m_j} f(x_i, p_k^j) \quad (5.4)$$

où m_j est le nombre de descripteurs dans la j^{eme} image, n est le nombre de descripteurs dans l'image requête et $f(x, y)$ est une fonction de coût qui exprime la ressemblance entre deux descripteurs (par soucis de clarté, nous avons ici uniquement pris le terme inverse-frequency, et non le terme complet tf-idf).

Dans BOF, l'algorithme utilise un quantificateur pour exprimer f . Un quantificateur est de la forme :

$$q : R^d \mapsto [1, K] \\ x \mapsto q(x)$$

Il transforme un descripteur (e.g. un SIFT dans un espace à 128 dimensions) en un entier. Avec le vocabulaire par K-means, le quantificateur associé à un descripteur x le centre de classe qui est le plus proche de x (l'index du centre est choisi comme quantificateur). Si deux descripteurs x et y sont proches, il y a, avec une forte probabilité, $q(x) = q(y)$. La fonction de coût entre deux descripteurs est alors :

$$f_q(x, y) = \delta_{q(x), q(y)} \quad (5.5)$$

où $\delta_{a,b}$ est le symbole de Kronecker qui vaut 1 si $a = b$ et 0 sinon.

Le score s'exprime alors par :

$$s_j = \frac{1}{m_j n} \sum_{i=1}^n \sum_{k=1}^{m_j} \delta_{q(x_i), q(p_k^j)} \quad (5.6)$$

Nous avons ici repris l'expression de l'algorithme BOF tel que présenté classiquement. Mais il est tout à fait possible de changer la fonction f utilisée. Par exemple, on pourra prendre f issu d'un algorithme de recherche r-NN, alors de la forme :

$$f_\epsilon(x, y) = \begin{cases} 1 & \text{si } d(x, y) < \epsilon \\ 0 & \text{sinon} \end{cases} \quad (5.7)$$

On pourrait aussi utiliser un algorithme k-NN :

$$f_{k-NN}(x, y) = \begin{cases} 1 & \text{si } x \text{ est un k-NN de } y \\ 0 & \text{sinon} \end{cases} \quad (5.8)$$

Dans notre cas, nous allons utiliser notre méthode de hachage HASHDIM pour définir une fonction f_ϵ .

5.3 Utilisation de notre hachage HASHDIM dans une recherche BOF

Avec notre hachage, nous assignons simplement un mot à chaque case de la table de hachage. L'algorithme pour calculer les scores de similarité entre une image requête et les images de la base est très simple. Il est décrit par l'algorithme 4. Dans cet algorithme, nous utilisons la fonction de coût :

$$f_{hash1}(x, y) = \frac{1}{n} \frac{1}{n_j} \times \delta'_{hash(x), hash(y)} \times \log(N/N_c)^2 \quad (5.9)$$

où x est un descripteur de l'image requête, y un descripteur de la base (appartenant à l'image j), n le nombre de descripteurs de l'image requête, n_j le nombre de descripteurs de l'image j , N le nombre total de descripteurs dans la base, N_c le nombre de descripteurs de la base qui sont hachés dans la table en $hash(y)$ et où la fonction δ' est très similaire au symbole de Kronecker mais adaptée pour notre cas. Le terme $\frac{1}{n} \frac{1}{n_j} \times \delta'_{hash(x), hash(y)}$ correspond à la partie term-frequency du schéma tf-idf alors que la partie $\log(N/N_c)^2$ correspond au

terme inverse-document-frequency (le carré vient du fait que cette pondération existe pour le descripteur de l'image ainsi que pour le descripteur de la base).

Dans la formule ci-dessus, y est un point de la base et est donc haché à une unique case de la table. Inversement, x est un point requête et il est donc haché dans plusieurs cases de la table (i.e. $hash(x)$ est une liste de clés alors que $hash(y)$ est une clé unique). Nous définissons alors :

$$\delta'_{hash(x),hash(y)} = \begin{cases} 1 & \text{si } hash(y) \in hash(x) \\ 0 & \text{sinon} \end{cases} \quad (5.10)$$

A des fins de tests, nous définissons d'autres fonctions du même style :

$$f_{hash2}(x, y) = \frac{1}{n} \frac{1}{n_j} \times \delta'_{hash(x),hash(y)} \quad (5.11)$$

où seul le terme term-frequency est utilisé (et pas le inverse-document-frequency) et :

$$f_{hash3}(x, y) = \delta'_{hash(x),hash(y)} \quad (5.12)$$

où aucun terme de la pondération tf-idf n'est utilisé.

Nous souhaitons confirmer que c'est la fonction f_{hash1} qui utilise le schéma tf-idf qui donne les meilleurs résultats. De plus, dans nos tests, le nombre de descripteurs par image est borné, et ce maximum est généralement atteint. Cela signifie qu'il y a de fortes chances que les fonctions f_{hash2} et f_{hash3} donnent des résultats très similaires.

Input: L : liste des descripteurs de l'image requête
Input: H : table de hachage des descripteurs de la base
Input: N : nombre de descripteurs dans la base

```

1 foreach point  $q \in L$  do
2   C = keys(q);
3   foreach  $c \in C$  do
4     V = H[c];
5      $N_c$  = taille de H[c];
6     foreach  $p \in V$  do
7       j = image de p;
8        $s_j + = \frac{1}{n} \frac{1}{n_j} \log(N/N_c)^2$ ;
9     end
10  end
11 end

```

Algorithm 4: Algorithme de recherche d'images similaires par BOF en utilisant notre hachage HASHDIM. Les images les plus similaires sont celles obtenant les scores les plus élevés.

Dans les fonctions ci-dessus, notre hachage est utilisé comme un quantificateur. On peut toutefois aussi tester la méthode par Bag-Of-Features en utilisant notre hachage simplement pour une recherche des voisins r-NN. Nous définissons alors la fonction :

$$\delta_{hash(x),hash(y)}^T = \begin{cases} 1 & \text{si } hash(y) \in hash(x) \text{ et } d(x,y) < T \\ 0 & \text{sinon} \end{cases} \quad (5.13)$$

Et les trois fonctions similaires sont générées :

$$f_{hash1}^T(x,y) = \frac{1}{n} \frac{1}{n_j} \times \log(N/N_c)^2 \times \delta_{hash(x),hash(y)}^T \quad (5.14)$$

$$f_{hash2}^T(x,y) = \frac{1}{n} \frac{1}{n_j} \times \delta_{hash(x),hash(y)}^T \quad (5.15)$$

$$f_{hash3}^T(x,y) = \delta_{hash(x),hash(y)}^T \quad (5.16)$$

5.4 Taille du vocabulaire

Dans [NS06], les auteurs montrent qu'une taille de vocabulaire importante donne de meilleurs résultats. Ils testent des vocabulaires de 10^4 et 10^6 mots. Dans notre approche, le nombre de mots possibles pour un couple de paramètres n, k est : $\binom{128}{k}$ (que ce soit pour un point requête ou un point de la base, une clé de hachage est calculée pour un vecteur de dimension k). Pour $k = 8$, le vocabulaire contient 1.4×10^{12} mots. En pratique, une majorité des mots ne sont jamais atteints. Les figures 5.1.a et 5.1.b montrent la fréquence des mots. Nous remarquons qu'une faible proportion représente des mots qui sont très fréquents. Ces mots correspondent à des descripteurs locaux qui sont très fréquents et donc peu distinctifs. De la même manière que dans les travaux de Sivic et al. [SZ03], nous supprimons un pourcentage des mots en supprimant les plus fréquents.

Dans un premier temps, nous testons quel pourcentage α_{DEL} de mots supprimés donne les meilleurs scores de rappel. Pour mesurer ce rappel, nous utilisons le protocole de test décrit en 2.7, en demandant à l'algorithme de retourner les 53 images les plus proches de l'image requête. Le rappel est le pourcentage d'images qui sont des vrais voisins parmi ces 53. Dans ce cadre, la précision est égale au rappel (et n'est donc pas indiquée). Le tableau 5.2 montre les résultats obtenus pour différentes valeurs de α_{DEL} en utilisant les deux fonctions f_{hash1}^T (le test est réalisé avec $T = 200$) et f_{hash1} , et avec plusieurs couples de paramètres (n, k) . Pour chaque fonction, les meilleurs scores de rappel sont obtenus pour $\alpha_{DEL} = 1$. Nous supprimerons donc 1% du vocabulaire (en choisissant les mots les plus fréquents) dans les expériences suivantes.

Nous remarquons de plus qu'en réduisant la taille du vocabulaire, le temps de calcul est largement diminué. Par exemple, on passe de 158ms par recherche (pour f_{hash1}^{200} et $(n, k) = (8, 6)$) à 40ms.

5.5 Choix de la fonction de coût

Dans ce paragraphe, nous comparons les fonctions de coût proposées. Les résultats pour certains couples de paramètres sont présentés dans le tableau

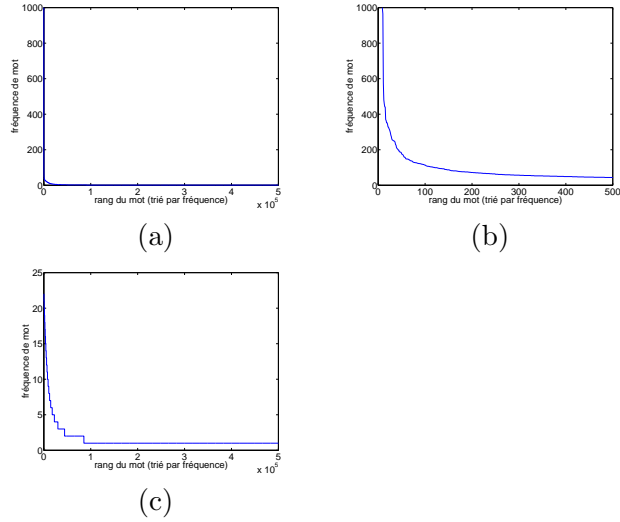


TABLE 5.1 – Fréquences des mots dans la base DB_{4000} avec $n, k = 10, 8$. Sur la figure (a), nous avons représenté tous les mots. Sur la figure (b), nous avons seulement zoomé sur une zone particulière de la figure (a). Sur la figure (c), 1% des mots sont supprimés (les plus fréquents).

	n	k	$\alpha_{DEL} = 0$	$\alpha_{DEL} = 1$	$\alpha_{DEL} = 2$	$\alpha_{DEL} = 3$
f_{hash1}	4	4	0.901/176ms	0.965/28ms	0.951/25ms	0.929/24ms
	8	6	0.955/145ms	0.966/47ms	0.964/45ms	0.956/45ms
	10	8	0.966/115ms	0.969/49ms	0.967/47ms	0.960/43ms
f_{hash1}^{200}	4	4	0.970/193ms	0.977/27ms	0.973/25ms	0.964/24ms
	8	6	0.978/158ms	0.979/40ms	0.977/37ms	0.975/35ms
	10	8	0.975/114ms	0.975/46ms	0.975/42ms	0.972/40ms

TABLE 5.2 – Rappel obtenu en fonction du pourcentage de mots supprimés du vocabulaire, en utilisant DB_{4000} . L'algorithme retourne 53 images et ne fait pas de vérification affine.

f	R	f	R	f	R
f_{hash1}	0.955	f_{hash2}	0.955	f_{hash3}	0.950
f_{hash1}^{150}	0.980	f_{hash2}^{150}	0.979	f_{hash3}^{150}	0.979
f_{hash1}^{200}	0.978	f_{hash2}^{200}	0.978	f_{hash3}^{200}	0.976
f_{hash1}^{250}	0.971	f_{hash2}^{250}	0.971	f_{hash3}^{250}	0.970

TABLE 5.3 – Rappel obtenu pour différentes fonctions de coût (en utilisant $n, k = 8, 6$).

5.3. Nous remarquons que, de manière générale, le choix de la fonction a peu d’impact sur le rappel (mais pour une même fonction, le seuil de distance est important). Nous choisissons toutefois d’utiliser les fonctions f_{hash1} et f_{hash1}^{150} qui donnent des rappels très légèrement supérieurs aux autres fonctions de coût (dans les tests suivants, nous testerons aussi f_{hash1}^{200} car ses scores de rappel sont seulement très légèrement inférieurs à ceux de f_{hash1}^{150}). Nous choisissons de conserver la fonction f_{hash1} même si son score de rappel est inférieur car elle ne calcule aucune distance entre SIFT. En effet, la fonction f_{hash1}^{150} vérifie que deux descripteurs ont une distance inférieure à 150 et impose donc de conserver en mémoire ces descripteurs. Inversement, la fonction f_{hash1} ne calcule aucune distance, et il est alors possible de ne pas garder en mémoire le descripteur SIFT à 128 dimensions. C’est là un gain non négligeable pour travailler sur de grandes bases d’images.

Après avoir choisi la taille du vocabulaire à utiliser et les fonctions de coût, nous étudions l’impact des paramètres n, k sur le résultat dans le paragraphe suivant.

5.6 Impact des paramètres (n, k)

Pour ces tests, nous avons enlevé 1% des mots les plus fréquents du vocabulaire et utilisé les fonctions de coût f_{hash1} , f_{hash1}^{150} et f_{hash1}^{200} . Les valeurs de rappel sont présentées dans la table 5.4 pour différents couples (n, k) .

Nous constatons que f_{hash1} n’obtient pas des scores de rappel aussi performants que les deux autres fonctions. Toutefois, comme précisé dans le paragraphe précédent, cette fonction est intéressante car elle ne nécessite pas de conserver en mémoire les descripteurs SIFT. Nous allons donc analyser les résultats d’une part pour f_{hash1} et d’autre part pour les deux autres fonctions de coût : f_{hash1}^{150} et f_{hash1}^{200} .

Concernant f_{hash1} , nous avons mis en gras dans le tableau de résultats les scores de rappel les plus élevés (ceux dépassant 0.96). Le meilleur rappel de 0.969 est obtenu pour le couple $(n, k) = (12, 10)$ (le même rappel est obtenu pour $(n, k) = (10, 8)$ mais en plus de temps). Ce rappel signifie que sur les 2650 images recherchées (50 requêtes avec pour chacune 53 images à retrouver), l’algorithme a retourné 2568 images correctes et en a manquées 82. Le tableau 5.5 montre les images manquées par transformation. Comme dans la partie 4.5.1 où nous utilisons notre hachage couplé à une vérification affine, ce sont les transformations *INCLINAISON15_15*, *CROP90* et *EMBOSS* qui sont de loin les plus difficiles. Sur les 82 images manquées, 78 sont dues à ces trois

transformations. En dehors de ces trois transformations, l’algorithme manque seulement 2 images dues à *INCLINAISON15*, une image de *CROP75* et une image de *CONTRAST_DECR_3*.

A elle seule, la transformation *INCLINAISON15.15* génère plus de la moitié des images manquées. Une des raisons possibles est que pour une telle transformation, les gradients sont décalés entre les cases des histogrammes du descripteur SIFT. C’est-à-dire qu’une dimension sur laquelle il n’y avait pas de gradient peut, après transformation, en avoir beaucoup. La clé de hachage générée par notre algorithme sera alors très différente. Nous n’apportons là aucune preuve, mais c’est une direction à étudier.

Nous remarquons aussi que les rappels obtenus pour les deux autres fonctions de coût (f_{hash1}^{150} et f_{hash1}^{200}) sont plus élevés. Cela paraît logique car pour ces fonctions, nous vérifions que deux points sont voisins pour incrémenter le score d’une image, alors que cette vérification n’est pas faite pour f_{hash1} . Dans le tableau 5.4, nous avons mis en gras les rappels supérieurs à 0.98 pour ces deux fonctions. Le meilleur rappel de 0.984 est obtenu pour f_{hash1}^{200} avec $(n, k) = (14, 6)$. Toutefois, pour ce score, le temps de recherche est de 1077ms par image. Un meilleur compromis est obtenu en utilisant f_{hash1}^{200} avec $(n, k) = (2, 2)$. Ce couple obtient un rappel de 0.983 juste inférieur au meilleur rappel mais avec un temps de recherche de 91ms par image. Dans le tableau 5.5, on peut voir que ce sont les mêmes trois transformations qui posent problème (*INCLINAISON15.15*, *CROP90* et *EMBOSS*). En dehors de ces transformations, très peu d’images sont manquées.

5.7 Comparaison avec des vocabulaires K-means

Dans ce test, nous souhaitons comparer les résultats de la recherche BOF pour plusieurs algorithmes de recherche de voisins utilisés pour définir le vocabulaire (notre algorithme HASHDIM et ceux à base de K-means). Nous utilisons les mêmes algorithmes que ceux utilisés dans la comparaison d’algorithmes r-NN. Le premier est celui utilisant un K-means classique, décrit dans les travaux de Sivic et al. [SZ03]. Le second est celui utilisant un K-means hiérarchique décrit dans les travaux de Nister et al. [NS06]. Les résultats sont présentés dans le tableau 5.6.

Le premier constat est que pour les vocabulaires à base de K-means, il est important d’utiliser un vocabulaire de grande taille. Pour l’algorithme KMeans, le meilleur rappel est de 0.943, obtenu pour un vocabulaire de 64.000 mots. Toutefois, nous n’avons pas testé avec plus de mots car les temps pour générer le vocabulaire (calcul du K-means) est déjà très important. D’autre part, pour le K-means, nous avons appris le vocabulaire sur une base différente (et plus petite) que la base DB_{32k} . La raison est que si le nuage de points est trop grand, l’étape de K-means est vraiment trop longue. Nous privilégions donc l’algorithme HKMeans qui utilise un K-means hiérarchique.

Pour l’algorithme HKMeans, l’algorithme de génération du vocabulaire est beaucoup plus rapide et il devient possible de générer le vocabulaire à partir du nuage de tests (celui de la base DB_{32k}). Nous avons aussi testé cet algorithme

Params		$hash_1$		$hash_1^{150}$		$hash_1^{200}$	
n	k	R	Time(ms)	R	Time(ms)	R	Time(ms)
2	2	0.943	99	0.979	159	0.983	91
4	2	0.736	364	0.978	735	0.974	384
4	4	0.965	30	0.977	27	0.977	27
6	2	0.687	683	0.977	726	0.976	731
6	4	0.957	67	0.979	46	0.976	51
6	6	0.949	24	0.964	25	0.963	25
8	2	0.338	1044	0.978	1133	0.978	1137
8	4	0.949	106	0.979	92	0.979	90
8	6	0.966	63	0.979	40	0.979	40
8	8	0.928	23	0.952	25	0.943	25
10	2	0.210	1532	0.977	1624	0.978	1631
10	4	0.941	200	0.981	168	0.980	173
10	6	0.967	124	0.980	108	0.981	102
10	8	0.969	64	0.980	44	0.975	46
10	10	0.901	23	0.924	24	0.915	24
12	2	0.157	2062	0.977	2216	0.981	2201
12	4	0.931	335	0.982	310	0.982	301
12	6	0.963	486	0.980	305	0.982	333
12	8	0.967	198	0.981	198	0.980	196
12	10	0.969	53	0.977	52	0.972	52
12	12	0.885	23	0.899	24	0.894	24
14	2	0.118	2655	0.977	2836	0.981	2840
14	4	0.915	589	0.982	569	0.982	576
14	6	0.963	1052	0.981	1174	0.984	1077
14	8	0.965	1081	0.983	1148	0.982	1185
14	10	0.969	398	0.977	376	0.977	408
14	12	0.967	68	0.973	65	0.970	63
14	14	0.873585	23	0.876	24	0.874	24

TABLE 5.4 – Résultats de la recherche d’images similaires en utilisant notre hachage couplé à une méthode par Bag-Of-Features, sur la base DB_{4000} . L’algorithme retourne 53 images.

fonction	algo	R	manquées	<i>INCLINAISON15_15</i>	<i>CROP90</i>	<i>EMBOSS</i>	<i>MEDIAN3</i>	<i>INCLINAISON15</i>	<i>CROP75</i>	<i>CONTRAST_DECR_3</i>	<i>ROT_CROP_10_50</i>	<i>CROP_50</i>	<i>ROT_CROP_45_50</i>	<i>INTENSITY_150</i>	<i>CROP_10</i>	<i>CROP_25</i>
f_{hash1}	n,k=12,10	0.969	82	47	19	12		2	1	1						
f_{hash1}^{200}	n,k=2,2	0.983	45	28	8	7	1			1						
f_{hash1}^{200}	n,k=14,6	0.984	41	24	7	8			2	1						
f_{hash1}^{150}	n,k=10,8	0.980	52	38	6	7				1						

TABLE 5.5 – Images manquées par une recherche Bag-Of-Features utilisant notre hachage HASHDIM sur une base de 4000 images. Seuls les couples de paramètres les plus efficaces sont présentés ici.

algo	Recall	Time (ms)
Our hashing, n,k=10,8	0.974	18
KMeans,k=4000	0.900	275
KMeans,k=16000	0.927	310
KMeans,k=32000	0.935	385
KMeans,k=64000	0.943	593
HKMeans, k,L=10,4	0.844	567
HKMeans, k,L=10,5	0.952	107
HKMeans, k,L=10,6	0.972	15
HKMeans, k,L=12,6	0.961	24
HKMeans-other, k,L=10,4	0.840	592
HKMeans-other, k,L=10,5	0.918	92
HKMeans-other, k,L=10,6	0.948	44
HKMeans-other, k,L=12,6	0.952	37

TABLE 5.6 – Scores obtenus par une recherche BOF pour plusieurs algorithmes r-NN utilisés. L’algorithme retourne les 53 images les plus similaires à la requête (sans faire de vérification affine). Les scores sont moyennés sur 50 requêtes. La base utilisée est DB_{32k} . KMeans utilise un K-means standard (méthode de [SZ03]), calculé sur un jeu d’images qui n’est pas le jeu de recherche DB_{32k} . HKMeans est la version hiérarchique du K-means, de Nister et al. [NS06] calculé sur la base DB_{32k} . HKMeans-other est la même méthode mais le vocabulaire est calculé sur une autre base. Pour les algorithmes K-means, l’assignation d’un mot à un descripteur est fait par une recherche linéaire exacte. Enfin, pour HKMeans et HKMeans-other, le score à plat de Nister et al. (décrit dans [NS06]) est utilisé.

si le vocabulaire est calculé sur une autre base d'images (méthode HKMeans-other). Les meilleurs résultats sont obtenus quand la base de recherche a été utilisée pour apprendre le vocabulaire. Dans l'algorithme HKMeans, le nombre de mots est égal au nombre de feuilles de l'arbre utilisé, c'est-à-dire k^L (k est le nombre de divisions à chaque noeud, et L la profondeur de l'arbre). Avec HKMeans, le meilleur score (rappel de 0.972) est obtenu pour un vocabulaire de un million de mots ($k = 10$ et $L = 6$) appris sur la base de recherche. Il est intéressant de noter que sur ce test, l'algorithme HASHDIM obtient un rappel quasi-identique 0.974, avec des temps de recherche similaires (respectivement 18ms et 15ms). Mais comparé à l'algorithme HKMeans-other, notre algorithme est plus performant. Ce dernier élément le rend attractif si la base évolue. Dans ce cas, pour être efficace avec l'algorithme HKMeans, il faudrait ré-apprendre régulièrement le vocabulaire.

En conclusion, nous pouvons affirmer que pour la recherche d'images similaires, notre algorithme est plus performant qu'un vocabulaire généré par un K-means (sauf si celui-ci est calculé sur la base de recherche). Il obtient un meilleur rappel tout en étant plus rapide. Un autre avantage est qu'il ne nécessite aucune longue étape d'apprentissage. En effet, le temps nécessaire pour effectuer un K-means à 32.000 classes sur un nuage de un million de points est de quelques heures.

Enfin, dans le tableau 5.7, nous avons repris les résultats du tableau 5.5 (recherche BOF sur la base DB_{4k}) afin d'analyser les résultats par transformation. Nous y avons ajouté les résultats d'une recherche BOF qui utilise un vocabulaire KMeans. Alors que pour notre algorithme, la transformation la plus difficile est l'inclinaison, pour le vocabulaire KMeans, il s'agit de la transformation de recadrage de 90% (i.e. *CROP_90*).

5.8 Test sur la base de 510.000 images

Nous appliquons le protocole précédent sur la base de 510.000 images. Comme la base est nettement plus grande, il est nécessaire d'appliquer la vérification affine à plus d'images. C'est-à-dire que le score BOF permet de sélectionner une liste d'images parmi les 510.000 et ensuite, la vérification affine permet de conserver uniquement celles similaires à la requête. Les résultats sont présentés dans le tableau 5.8. A noter que pour ce test, l'algorithme de vérification affine a été ré-implémenté. Notamment, lorsque l'on a peu de paires de points, il est plus rapide de tester exhaustivement toutes les possibilités que de procéder par jets aléatoires.

Sur la base de 32.000 images, nous obtenions un rappel de 0.974 avec HASHDIM, en retournant 53 images. Sur la base de 520.000 images, ce rappel est de 0.743 si l'on ne teste (i.e. avec une recherche de vérification affine) que 53 images. Il est donc nécessaire de tester plus d'images. Par exemple, lorsqu'on applique la vérification affine aux 8000 images sélectionnées par le score BOF, le rappel monte à 0.965.

Dans les paragraphes précédents, nous avons testé l'algorithme HASHDIM sur une base d'images que nous avons construite. Dans le paragraphe suivant,

fonction	algo	R	manquées	INCLINAISON15_15	CROP90	EMBOSS	MEDIAN3	INCLINAISON15	CROP75	CONTRAST_DECR_3	ROT_CROP_10_50	CROP_50	ROT_CROP_45_50	INTENSITY_150	CROP_10	CROP_25
f_{hash1}	n,k=12,10	0.969	82	47	19	12		2	1	1						
f_{hash1}^{200}	n,k=2,2	0.983	45	28	8	7	1			1						
f_{hash1}^{200}	n,k=14,6	0.984	41	24	7	8			2	1						
f_{hash1}^{150}	n,k=10,8	0.980	52	38	6	7				1						
f_{hash1}	<i>KMeans</i> ,k=4000	0.978	59	6	31	7			7		5	2	1			
f_{hash1}	<i>KMeans</i> ,k=16000	0.993	19		16	2			1							
f_{hash1}	<i>KMeans – other</i> ,k=4000	0.956	126	28	38	27	1		13		10	4	2	1	1	1
f_{hash1}	<i>KMeans – other</i> ,k=16000	0.967	88	26	34	14			6	2	2	2	1			1
f_{hash1}	<i>KMeans – other</i> ,k=32000	0.969	80	29	31	13			4		1	1	1			
f_{hash1}	<i>KMeans – other</i> ,k=64000	0.972	73	28	26	14			2		1	1	1			

TABLE 5.7 – Images manquées par une recherche Bag-Of-Features utilisant notre hachage et un vocabulaire K-means, sur une base de 4000 images. Seuls les couples de paramètres les plus efficaces sont présentés ici. (*KMeans*) : le vocabulaire est calculé sur la base de tests. (*KMeans – other*) : le vocabulaire est calculé sur une autre base.

Nb Images Testées	R	P	T-BOF(ms)	T-AFF(ms)
53	0.743	0.980	265	11
1000	0.940	0.973	268	24
2000	0.954	0.971	267	33
4000	0.961	0.971	269	53
8000	0.965	0.971	268	85
16000	0.966	0.971	270	137

TABLE 5.8 – Recherche d’images similaires par Bag-Of-Features utilisant notre hachage, sur une base de 510.000 images. La première ligne est un rappel du score obtenu sans Bag-Of-Features, c’est-à-dire en appliquant une vérification affine à toutes les images ayant des correspondances.

nous appliquons la recherche Bag-Of-Features avec l'algorithme HASHDIM sur une base publique.

5.9 Tests sur la base Nister

Dans la littérature des méthodes par Bag-Of-Features, une base très utilisée est celle de Nister, présentée dans l'article [NS06] et téléchargeable sur le site Internet des auteurs. Nous présentons ici nos résultats sur cette base.

5.9.1 Base d'images

Cette base consiste en 10200 images de 2550 objets, chacun étant pris en photo depuis quatre points de vue différents. La figure 5.9 montre les 12 photos de 3 objets de cette base. Quand on utilise une des images de la base comme image requête, l'algorithme doit idéalement retrouver les 4 images du même objet. La mesure utilisée dans la littérature est le nombre d'images du même objet retrouvées parmi les 4 images obtenant les meilleurs scores. A noter que dans notre classification des problèmes de recherche d'images (présentée en 1.2.1), cette base fait partie de la recherche d'images de mêmes scènes, alors que jusqu'à présent, nous n'avions testé nos algorithmes que sur le problème de la recherche d'images similaires.

Dans nos tests, nous utilisons les descripteurs fournis avec la base d'images. Nous pourrions utiliser notre algorithme SIFT mais les différences de performances entre notre algorithme et celui de [NS06] pourraient alors provenir de deux sources : les descripteurs utilisés et l'algorithme de recherche d'images similaires. En utilisant les descripteurs fournis, nous ne comparons que l'algorithme de recherche d'images similaires. Le détecteur de zones d'intérêt utilisé est MSER (pour Maximally Stable Extremal Regions, introduit dans [MCMP02]). Les zones d'intérêt extraites sont décrites par un descripteur SIFT à 128 dimensions. L'avantage d'utiliser les MSER est que la zone d'intérêt est invariante par changement affine. Or, comme les objets sont pris de points de vue différents, ce point est fondamental pour obtenir de bons scores sur cette base.

5.9.2 Protocole de test

Pour pouvoir comparer nos résultats avec ceux de [NS06], nous utilisons la même mesure de performances. Celle-ci consiste à lancer l'algorithme de recherche d'images similaires pour chaque image du jeu de données. L'algorithme est configuré pour retourner les 4 images les plus proches (selon le score BOF, sans vérification affine). S'il ne se trompe pas, parmi les 4 images, on retrouve l'image identique à l'image requête, ainsi que les 3 autres images du groupe. Le score est le nombre d'images du groupe parmi les 4 images retournées. Un score de 1 signifie en général qu'on a simplement retrouvé l'image requête dans la base. Un score de 4 signifie qu'on a retrouvé les 4 images du groupe dans les 4 premières positions. Le score final est la moyenne de ce score sur toutes les requêtes lancées.



TABLE 5.9 – Trois groupes d’images de la base Nister

			$\alpha_{DEL} = 0$	$\alpha_{DEL} = 0.001$	$\alpha_{DEL} = 0.01$	$\alpha_{DEL} = 0.1$	$\alpha_{DEL} = 1$	$\alpha_{DEL} = 2$
f_{hash1}	4	4	1.03	1.13	2.67	2.65	2.51	2.41
	8	6	1.11	1.79	2.92	2.83	2.65	2.58
	14	11	2.51	2.79	2.99	2.95	2.84	2.80
f_{hash1}^{400}	4	4	2.83	2.81	3.07	2.98	2.90	2.87
	8	6	2.26	3.19	3.22	3.10	2.93	2.85
	14	11	2.98	3.15	3.15	3.11	3.01	2.97

TABLE 5.10 – Scores obtenus en fonction du pourcentage de mots les plus fréquents supprimés du vocabulaire. La base utilisée contient 600 images. Le score est moyenné sur 600 requêtes. Le meilleur score est 3.07, obtenu pour $n, k = 14, 10$.

Dans un premier temps, nous travaillons sur une base de taille réduite afin de pouvoir tester rapidement un grand nombre de paramètres. Nous choisissons aléatoirement 150 groupes de 4 images dans la base de Nister (i.e. 600 images).

5.9.3 Suppression des mots les plus fréquents

Nous cherchons d’abord le nombre de mots du vocabulaire à supprimer. Le tableau 5.10 montre les résultats obtenus. On voit que les meilleurs scores sont obtenus pour $\alpha_{DEL} = 0.01$. Cela correspond par exemple pour le couple $n, k = 14, 11$ à supprimer 36 mots des 368907 qui sont atteints par le nuage de points. Dans la suite des tests sur la base Nister, nous utiliserons toujours $\alpha_{DEL} = 0.01$. A noter que cette valeur est très différente de celle obtenue pour des SIFT non calculés sur des détecteurs MSER (test 5.4). C’est un point qui reste à explorer.

n \ k	5	6	7	8	9	10	11	12	13	14	15
8	2.57	2.91	2.83	2.59							
9	2.35	2.93	2.95	2.86	2.55						
10	1.86	2.84	2.92	2.96	2.74	2.53					
11		2.58	2.73	2.99	2.94	2.78	2.48				
12			2.38	2.92	3.02	2.93	2.75	2.46			
13				2.76	2.93	2.99	2.90	2.71	2.42		
14					2.74	3.07	2.99	2.85	2.68	2.39	
15						3.02	3.02	2.95	2.81	2.64	2.32

TABLE 5.11 – Scores obtenus en fonction des paramètres n et k . La base contient 600 images et le score est moyenné sur 600 requêtes.

Algorithm	Score
HKMeans, $k, L=10, 6$	3.16
AKMeans, $k=1M$	3.45
Our hashing	2.82

TABLE 5.12 – Résultats de plusieurs algorithmes sur la base de Nister [NS06]. Pour le KMeans à plat de Sivic et al., nous avons pris le score dans les travaux de Philbin et al. [PCI+07]. Il s’agit en fait d’une modification de l’algorithme de Sivic et al., mais qui donne des scores très similaires de manière plus rapide. Cette méthode est notée AKMeans. Pour HKMeans, nous avons pris le score dans l’article de Nister et al. [NS06].

5.9.4 Choix du couple n, k

Toujours sur cette même base de 600 images, nous cherchons le couple n, k qui donne le meilleur score. Le tableau 5.11 montre les résultats obtenus pour certains couples de paramètres, en utilisant la fonction f_{hash1} . Le meilleur score est 3.06, obtenu pour le couple $n, k = 14, 10$.

5.9.5 Comparaison avec des vocabulaires K-means

Pour comparer nos résultats avec ceux de la littérature, nous avons repris les résultats d’autres algorithmes dans les articles décrivant ces méthodes. Les résultats sont mis dans le tableau 5.12.

Sur cette base, notre algorithme est moins performant que ceux utilisant un vocabulaire K-means. Une piste d’explication peut être que notre hachage ne semble pas adapté lorsque les descripteurs sont calculés entre des images de points de vue différents. Les tests sur la base de recherche d’images similaires (partie 5.5) ont montré que la transformation de loin la plus difficile pour notre algorithme était l’inclinaison (transformation qui simule un changement de point de vue). Or, dans cette base Nister, tous les objets sont pris selon des points de vue largement différents. Même si le descripteur utilise une région d’intérêt MSER, qui cherche à être invariante par transformation affine, cela n’est peut-être pas suffisant pour rendre l’algorithme HASHDIM performant

sur ces points.

Une autre raison possible de baisse de performance est que l'utilisation d'un détecteur de région d'intérêt affine invariant va concentrer les descripteurs dans l'espace. La raison est que toutes les déformations affines d'un détail (e.g. toutes les ellipses obtenues par déformation affine d'un cercle) vont générer le même descripteur. Dans ce cas, le tri des dimensions nécessaire à notre hachage est peut-être beaucoup plus sensible aux petites variations du descripteur, et donc l'algorithme HASHDIM moins efficace. Une perspective de nos travaux est d'explorer plus précisément ce point.

5.10 Conclusion

Dans ce chapitre, nous souhaitons montrer que l'algorithme HASHDIM, proposé au chapitre précédent s'intégrait dans une recherche par Bag-Of-Features. Et bien sûr, un second objectif était d'en comparer les résultats aux algorithmes Bag-Of-Features plus classiques pour lesquels un vocabulaire est construit par apprentissage K-means et de conclure sur les avantages d'utiliser HASHDIM pour faire de la recherche d'images similaires par Bag-Of-Features.

Dans la première partie de ce chapitre, nous avons donc montré que notre hachage s'adaptait très simplement aux méthodes dites de Bag-Of-Features. Dans le cadre de la recherche d'images similaires, nous avons appliqué le protocole de test pour la recherche d'images similaires défini en 2.7. Sur ces données, notre algorithme obtient un meilleur rappel et est plus rapide qu'un algorithme basé sur un vocabulaire K-means. Le seul cas où le vocabulaire K-means est meilleur est le cas où il est calculé sur la base sur laquelle est effectuée la recherche, ce qui n'est pas possible dans certains cas, par exemple si la base évolue régulièrement ou si elle est trop grande. De plus, un avantage de notre algorithme est qu'il ne nécessite aucun apprentissage. Inversement, l'étape de calcul des K classes du K-means s'avère très longue pour des grands nuages de points.

L'un des points faibles de notre hachage est qu'il n'est pas très robuste aux changements de points de vue. Avec cette transformation, deux descripteurs d'une même région ont moins de chances d'aboutir à des clés de hachage identiques. Expérimentalement, nous constatons cela sur le test de recherche d'images similaires, où la transformation d'inclinaison est largement la plus difficile. De même, sur la base Nister, où les objets sont pris de points de vue différents (i.e., il s'agit de recherche d'images de même scène), notre algorithme est moins performant que celui utilisant un vocabulaire K-means. Pour ce type de données, HASHDIM ne semble pas tout à fait adapté. Toutefois, cette recherche d'images de même scène n'était pas un des objectifs principaux de nos travaux.

Chapitre 6

Conclusion et Perspectives

Dans cette première partie de thèse, notre objectif global était d'étudier les performances des algorithmes de recherche des plus proches voisins, pour les descripteurs locaux d'images, et d'en proposer de nouveaux, plus performants. Cette performance se mesure en terme de rapidité d'exécution, mais aussi, comme nous l'avons vu en terme de mémoire consommée. En effet, un algorithme très rapide, mais nécessitant beaucoup de mémoire sera potentiellement très ralenti si la machine ne dispose pas d'assez de RAM. Ces travaux étaient motivés par deux applications : celle classique de la recherche d'images similaires et celle de l'analyse d'images de linéaires de supermarchés. Dans ce cadre, nous nous sommes restreint aux nuages de points SIFT car ces descripteurs locaux d'images sont très efficaces.

Pour répondre à ces objectifs, dans un premier temps, nous avons étudié l'algorithme le plus simple : la recherche r-NN linéaire. Nous avons montré que l'utilisation de distances partielles permettait un gain important (jusqu'à 5 fois plus rapide). Nous avons aussi montré qu'une implémentation sur GPU peut être jusqu'à cent fois plus rapide qu'une version CPU (même si des gains 30 fois plus rapides sont plus communs). Pour diminuer le temps de recherche des voisins SIFT et la taille mémoire nécessaire, nous avons aussi proposé une méthode de sélection des SIFT qui permet de diminuer fortement le nombre de descripteurs par image tout en conservant un rappel quasi identique en terme de recherche d'images. Malgré ces gains intéressants, ces premières méthodes ne sont pas suffisantes pour traiter des grandes bases d'images. Nous avons donc orienté nos travaux vers des structures d'indexation dédiées au problème de la recherche des plus proches voisins.

Nous avons alors proposé une modification de l'algorithme LSH pour accélérer la recherche r-NN sur des points SIFT. Cet algorithme est performant, mais comme toutes les méthodes LSH, il souffre d'une forte consommation mémoire. Nous avons donc proposé d'autres fonctions de hachage, dans le but d'obtenir des performances similaires au LSH, mais demandant beaucoup moins de mémoire. Ces fonctions de hachage sont basées sur un tri, pour chaque point, des dimensions de l'espace. L'algorithme r-NN qui utilise ces fonctions, appelé HASHDIM, est presque aussi rapide que le LSH modifié, mais consomme beaucoup moins de mémoire.

Enfin, dans un dernier chapitre, nous avons intégré HASHDIM dans une recherche d’images similaires par Bag-Of-Features. Sur notre protocole de test, cet algorithme est plus performant que ceux utilisant des vocabulaires K-means. Toutefois, il semble que les fonctions de hachage de HASHDIM ne soient pas adaptées à certains types de données (e.g. descripteurs SIFT calculés sur des régions invariantes par transformation affine).

Finalement, pour l’application de recherche d’images similaires, l’utilisation de HASHDIM dans une recherche par Bag-Of-Features apparaît selon nos tests comme la meilleure option. Toutefois, dans nos tests sur l’application d’analyse de linéaires, nous avons montré que coupler une recherche de voisins par segmentation K-Means avec une attribution de classe faite par GPU était la meilleure option (sur ce point, il serait important de tester l’approche de Philbin et al. [PCI+07] dans laquelle l’attribution de classe se fait par “randomized kd-trees”). Mais si la machine ne dispose pas de GPU, pour l’application d’analyse de linéaires de supermarché, la conclusion n’est pas aussi nette et il est difficile de s’exprimer clairement pour une méthode. D’autre part, lorsque l’on décide de “coupler” des algorithmes (e.g. K-Means et vérification rapide de distance dans Jegou et al. [JDS08]), la liste des possibilités d’algorithmes augmente très rapidement. L’objectif étant d’utiliser un premier algorithme pour éliminer, dans une première passe, les points vraiment éloignés du point requête. Et ensuite, un second algorithme est utilisé pour ne retrouver, parmi ces points proches, que ceux qui sont réellement des voisins. Il devient alors difficile de mesurer, de manière exhaustive, les performances de tous les algorithmes.

Au niveau applicatif, ces travaux ont abouti à la mise au point de deux logiciels. Le premier est un moteur de recherche d’images similaires performant. Celui-ci retrouve des images similaires dans une base de 500.000 images en moins de 300ms. Son interface est simplement constituée d’une page web qui permet de charger une image et d’appeler le moteur de recherche avec cette image comme requête. Le second logiciel permet d’analyser une image de linéaire de supermarché et de reconstruire son planogramme. L’interface développée permet par exemple de lancer cette analyse automatique, mais aussi de corriger les résultats, de fusionner plusieurs étagères analysées, etc.

Perspectives

Dans nos travaux futurs, l’un des premiers points qui nous paraît important est de compléter les tests comparatifs que nous avons effectués dans cette thèse. D’abord, concernant la recherche linéaire, il est nécessaire de tester une implémentation qui utilise la multiplication matricielle (voir partie 2.6.2) avec une librairie optimisée (de type BLAS, par exemple ATLAS [ATL]). Il sera aussi nécessaire de rajouter la méthode décrite dans les travaux de Jegou et al. [JDS08], qui couple un K-means (avec un nombre pas trop élevé de classes) et une méthode rapide pour éliminer les points non voisins. Enfin, une méthode importante à rajouter est celle dite de “randomized trees” (voir [AAG96, LF06, SAH08], par exemple utilisé dans Philbin et al. [PCI+07] pour attribuer la classe d’un point dans une segmentation K-means), ou celle proche, de KD-Forest, introduite dans Valle et al. [VCPF08a] (méthode utilisant plu-

sieurs KD-Tree, chacun étant généré à partir de la projection du nuage de points sur un sous-espace de dimension réduite).

D’autre part, il serait intéressant de tester l’algorithme HASHDIM couplé à une méthode Bag-Of-Features dans le cadre de la recherche de vidéos similaires. Cette application est proche de la recherche d’images similaires et notre algorithme est donc tout à fait adapté. Cela permettrait par exemple de le tester sur le jeu de données publique TRECVID.

Un autre axe qui nous paraît important est d’essayer de construire un algorithme d’association de descripteurs SIFT optimal selon un certain sens. Par exemple, en utilisant des transformations d’images synthétiques, ou réelles (et connues), il est possible de classer toutes les paires de SIFT entre ces images en deux groupes : inliers (paires qui respectent la transformation) et outliers (toutes les autres paires). Cette classification sert alors de vérité terrain. Pour une paire de SIFT, un classifieur proposé fournit sa classe (i.e. inlier ou outlier). Il est alors possible d’estimer le score d’un tel classifieur en utilisant la vérité terrain. Une telle construction devrait permettre d’améliorer les scores de recherche d’images par “Bag-Of-Features”. La difficulté est d’apprendre ce classifieur au vu de la grande quantité de données.

Enfin, à un niveau plus applicatif, l’application d’analyse de linéaire soulève des problèmes particuliers. L’un d’eux est de choisir le bon produit lorsque plusieurs images sont très similaires. Par exemple, pour certains produits, seule une petite région du packaging va changer de couleur, ou encore, par exemple sur certaines crèmes fraîches, seule l’indication du taux de matière grasse change. Bien que l’approche basée uniquement sur des correspondances de SIFT soit une bonne première étape, il reste à proposer une meilleure fonction de score de similarité (entre un produit de la photo de linéaire et un produit de la base) afin d’aboutir à une application plus robuste, et demandant moins de vérification manuelle.

Contributions

Nous résumons ici les contributions de nos travaux dans cette première partie de thèse :

- un algorithme qui utilise les descripteurs locaux pour construire un planogramme à partir d’une image de linéaire de supermarché.
- une méthode linéaire de recherche de plus proches voisins qui utilise une distance partielle sur les SIFT.
- une implémentation de la recherche linéaire des plus proches voisins sur GPU (processeur de la carte graphique, fortement parallélisable).
- une méthode de sélection des descripteurs SIFT par image, dans le but de réduire la taille des nuages de points à traiter.
- une modification de l’algorithme de recherche des plus proches voisins LSH [GIM99] pour l’adapter aux descripteurs SIFT.
- un algorithme original de recherche des plus proches voisins basé sur des fonctions de hachage (algorithme HASHDIM).
- une étude comparative des performances de HASHDIM et des principaux algorithmes de recherche des plus proches voisins sur des descripteurs

SIFT.

- l'intégration de HASHDIM dans une recherche d'images par Bag-Of-Features et la comparaison des résultats obtenus avec ceux de la méthode Bag-Of-Features basée sur d'autres algorithmes de recherche des plus proches voisins.

Deuxième partie

**Contributions à la
reconstruction 3D multi-vues**

Chapitre 7

Reconstruction 3D : Introduction

Sommaire

7.1	Reconstruction 3D multi-vues	146
7.2	Histoire de la formation des images	146
7.3	Histoire de la reconstruction 3D	147
7.4	Problématique	149

7.1 Reconstruction 3D multi-vues

Dans la seconde partie de cette thèse, nous nous intéressons à la reconstruction 3D d'un objet à partir de plusieurs images. Cet énoncé regroupe en fait deux problèmes duaux : soit l'objet à reconstruire est immobile et les images sont prises à partir de points de vue différents soit l'objet est mobile et les photos sont enregistrées depuis un point de vue fixe. Certaines étapes des algorithmes de reconstruction peuvent différer entre les deux situations, mais globalement, il s'agit du même problème. On regroupe les algorithmes pour reconstruire une représentation 3D de l'objet à partir d'images sous le nom de reconstruction 3D multi-vues. Pour l'instant, nous restons volontairement imprécis sur le choix de la représentation souhaitée. Cela peut être un nuage de points, une surface triangulée, une grille de voxels ou un modèle 3D par exemple.

La reconstruction 3D a toujours été un sujet très étudié en vision par ordinateur. Ceci s'explique par le fait que si cette étape est correctement effectuée, elle ouvre de nombreuses possibilités d'interprétation en aval. Il peut s'agir de mesures de dimensions, d'analyses de comportements... De manière générale, comme le monde réel est en 3 dimensions, son analyse est plus simple si l'on travaille en 3 dimensions. Par exemple, pour un système de vidéo-surveillance, les décisions à prendre seront plus simples si l'on est capable de représenter la scène en 3D et de représenter les déplacements depuis une vue du dessus. En ne disposant que d'une vue quelconque de la scène, il est plus difficile de définir si une personne a pénétré dans une zone délimitée ou si elle est cachée par une autre personne.

7.2 Histoire de la formation des images

Dans le domaine de la reconstruction 3D, on cherche à retrouver une représentation du monde 3D à partir de plusieurs de ses projections dans des plans (les images). Pour aborder correctement ce problème, il est nécessaire de bien comprendre le problème inverse, c'est-à-dire : la formation des images comme projection d'un monde 3D.

L'histoire de cette compréhension est liée à l'histoire de la peinture. Avant de maîtriser la projection perspective, on trouvait généralement des erreurs grossières sur les représentations. Par exemple, dans l'image 7.1 datant du 14^{ème} siècle, les règles de perspective ne sont pas respectées, notamment pour le côté gauche de la tour. De même sur l'image 7.2, on a tracé les lignes parallèles qui devraient s'intersecter en un unique point de fuite, ce qui n'est pas le cas. Les lignes du bas sont même complètement parallèles. A partir du 15^{ème} siècle et du début de la Renaissance, les règles de perspective sont généralement respectées dans les peintures. Filippo Brunelleschi fut le premier à introduire des règles pour respecter la perspective en peinture, ensuite utilisées par Donatello notamment. Un exemple célèbre d'utilisation de ces règles est illustré sur la figure 7.3 sur laquelle nous avons tracé les lignes parallèles qui convergent bien en un unique point de fuite. Ces règles furent regroupées dans un traité sur les lois de la perspective par Leon Battista Alberti en 1435, sous le nom de *Della*



FIGURE 7.1 – Image extraite de Kaufmann Haggadah, 14^{ème} siècle, The Jews : A Treasury of Art and Literature. NY : Levin Assoc. 1992

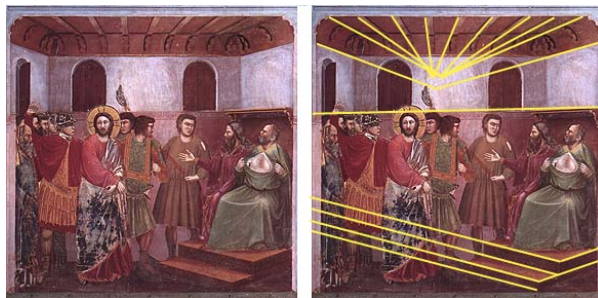


FIGURE 7.2 – Jesus Devant le Caif, Giotto (1305).

Pictura. Une des règles explique par exemple comment dessiner le pavage d'une place (on parle alors de Albertian Grid). Ce traité explique aussi comment, en visualisant une scène par un petit trou au sommet d'une pyramide avec un morceau de verre formant sa base, on peut la dessiner correctement sur le verre. Albrech Dürer fabriqua quelques machines sur ce principe. L'image 7.4 illustre une variante où l'on utilise une grille au lieu d'un morceau de verre.

7.3 Histoire de la reconstruction 3D

Les premiers travaux sur la géométrie de deux vues sont ceux du mathématicien allemand Kruppa, en 1913. Le premier algorithme de reconstruction 3D d'un nuage de points à partir de deux images de ces points est attribué à Longuet-Higgins [Lon81], basé sur la contrainte épipolaire. Ces premiers travaux sur la reconstruction 3D à partir de deux images sont regroupés dans [Fau93]. Les travaux suivants se sont ensuite concentrés sur comment reconstruire aussi des lignes, comment utiliser plus de deux vues simultanément ou bien encore, quelle quantité minimale de donnée est nécessaire pour une reconstruction 3D. Les résultats majeurs de ces travaux sont regroupés dans [HZ04], [FLP01] ou [MSKS04].

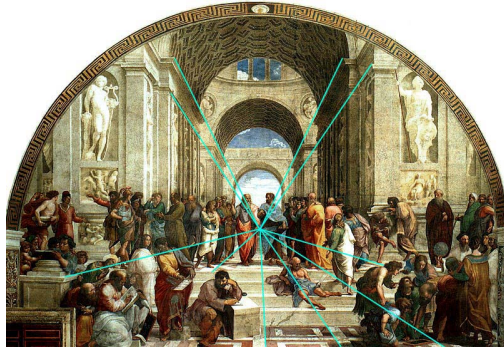


FIGURE 7.3 – L'école d'Athènes, Raphaël (1518).



FIGURE 7.4 – Albrecht Dürer (1471-1528) Dessinateur travaillant sur une vue en perspective de femme, 1525. Etching. The Metropolitan Museum of Art, New York.

7.4 Problématique

Dans nos travaux, nous avons étudié les deux problématiques suivantes :

- Comment utiliser les algorithmes classiques de reconstruction 3D pour des voitures ? Ces algorithmes traitent bien de la reconstruction d'objets dont les surfaces sont lambertiennes et texturées mais sont plus difficiles à mettre en oeuvre pour des surfaces avec peu de texture, réfléchissantes ou transparentes comme on en trouve sur les voitures. Une difficulté supplémentaire est que certaines parties sont mobiles par rapport à l'objet (les roues). Pour ces raisons, nous étudierons si les algorithmes classiques sont efficaces sur de telles surfaces et comment il faut les adapter pour reconstruire des voitures.
- Pour la reconstruction 3D, quel est l'apport des descripteurs SIFT présentés dans la première partie de cette thèse ? Leurs performances permettent-elles d'envisager d'autres algorithmes que les précédents descripteurs locaux, moins efficaces, rendaient impossibles ? Et si c'est le cas, quel est l'avantage de ces approches ?

Dans le cadre de la reconstruction 3D de voitures, la motivation est industrielle. On souhaite reconstruire en 3D des voitures pour faire de l'identification de véhicule. L'une des raisons de vouloir identifier des voitures est le contrôle des entrées-sorties de parking. Un cas de vol typique est détaillé ici. Un voleur rentre dans un parking avec une voiture sans valeur. A l'entrée, l'automate lui fournit un ticket et lit le numéro de la plaque d'immatriculation qu'il associe au ticket. Une fois dans le parking, le voleur échange les plaques d'immatriculation entre la voiture avec laquelle il est entré et la voiture à voler. Il se présente alors à la barrière de sortie du parking. L'automate lit le numéro de plaque qui correspond bien à celui écrit sur le ticket du voleur. La barrière s'ouvre, la voiture est volée. Si l'on dispose d'un système qui enregistre en plus la forme de la voiture, lors du contrôle de sortie, il faut se représenter avec le même ticket et la même voiture. On peut alors empêcher tous les vols du type décrit.

Une autre application potentielle est le péage automatique sur autoroute. Le prix du péage dépend en effet du véhicule (camionnette, poids lourd, caravane, simple voiture...). On souhaite développer un système qui lit les plaques d'immatriculation et reconnaît le type de véhicules pour pouvoir ensuite facturer. On peut alors imaginer disposer les caméras sur une barrière métallique au dessus des voies, les véhicules n'ayant pas à s'arrêter.

Dans nos travaux, nous n'avons pas cherché à répondre directement à l'une ou l'autre de ces demandes industrielles. Nous cherchons plutôt à obtenir la meilleure reconstruction possible en utilisant des méthodes de reconstruction 3D. Au niveau industriel, il s'agit de travaux pour une étude très en amont d'une possible réalisation. Nous avons orienté nos travaux vers la problématique plus générale : comment reconstruire en 3D une voiture à partir d'une acquisition vidéo ?

Dans le chapitre suivant, nous faisons un état de l'art de la reconstruction 3D. Ensuite, nous proposons des méthodes de reconstruction de voitures qui utilisent des algorithmes classiques. Puis, nous étudions une méthode par déformation de surface, dans le but d'améliorer la précision des reconstructions

obtenues.

Chapitre 8

Etat de l'art de la reconstruction 3D

Sommaire

8.1	Systèmes actifs	152
8.2	Introduction aux systèmes passifs	154
8.3	Méthodes par approximation de nuages de points .	162
8.4	Méthodes par analyse de l'espace 3D	164
8.5	Méthodes de reconstruction cohérente avec les images	166
8.6	Autres approches par critère de photo-cohérence local	176
8.7	Conclusion	177

Dans ce chapitre, nous présentons un état de l'art de la reconstruction 3D. Nous restons volontairement générique en ne spécifiant pas une méthode de reconstruction (par images, laser, une seule caméra, multi-caméras...). Le sujet de nos travaux concerne la reconstruction 3D à partir d'images mais pour situer ce sujet dans le contexte plus général de la reconstruction 3D, nous présentons aussi des méthodes qui n'utilisent pas l'image comme acquisition. Dans un premier temps, nous introduisons ici les grandes familles de méthodes qui permettent de retrouver une information sur la géométrie 3D d'une scène à partir d'une acquisition d'informations. Nous proposons de classer les systèmes selon qu'ils sont actifs ou passifs. Les systèmes actifs sont caractérisés par une émission d'un signal nécessaire pour obtenir une information 3D. A l'inverse, les systèmes passifs ne font que recevoir un signal de la scène. Les systèmes actifs sont décrits en section 8.1. Tous les paragraphes suivants sont dédiés à des méthodes passives.

8.1 Systèmes actifs

8.1.1 Acquisition laser

Dans un système par acquisition laser, un signal est émis par le laser, réfléchi par la scène. Le système d'acquisition mesure ce signal réfléchi. A partir de la différence de phase, on peut déduire le temps de parcours (Time Of Flight), et avec une hypothèse sur la vitesse du signal, on estime la distance entre le point où se fait l'émission et l'objet. En général, le laser est monté sur un système de balayage. C'est-à-dire que le laser va émettre dans un intervalle de directions possibles. On obtient alors un nuage de points 3D. Pour avoir un nuage complet (tous les côtés de la scène), le laser est déplacé et de nouveaux nuages de points sont acquis depuis ces positions. Suit une étape de recalage où les nuages de points sont mélangés pour n'en former qu'un. La figure 8.1 montre un système d'acquisition laser de la société Trimble.

Un des inconvénients de ces systèmes est leur coût. De plus, comparative-ment à une acquisition par une simple caméra, ce système est délicat à installer (le système est notamment plus volumineux, difficile à intégrer...). L'acquisition laser est traditionnellement utilisée en topographie, suivi de chantier (mines, tunnel...), ou en mesure d'évolution des structures (e.g. des ponts), archéologie, etc.

8.1.2 Caméras actives Time-Of-Flight

Ces caméras récentes sont basées sur la mesure du temps de parcours d'un signal. Le capteur de la caméra est entouré d'émetteurs (e.g. LEDs qui émettent en infrarouge). Ces signaux sont émis puis réfléchis par les objets de la scène. Au moment où ces signaux sont mesurés sur le capteur, on peut mesurer leurs temps de parcours et en déduire la profondeur de la scène. La figure 8.2 montre un exemple de ce type de caméras. A noter que ces caméras sont généralement appelées caméras TOF (pour Time Of Flight). Elles ont été testées pour des



FIGURE 8.1 – Exemple de système d’acquisition laser (modèle Trimble Gx).



FIGURE 8.2 – Exemple de caméra TOF. Modèle de la société PMD Technologies GmbH.

applications variées : robotique [Hus07, Per06], biométrie [LOB], suivi de personnes sur vidéo [BSA06]...

Pour notre problème, nous avons eu la possibilité de manipuler pendant quelques jours la caméra de la société PMD Technologies GmbH. Toutefois, un problème connu de ces caméras est leur sensibilité aux conditions extérieures ainsi qu’au type de surface. Dans notre cas, sur les surfaces de voitures, les mesures étaient beaucoup trop bruitées pour être interprétables.

Toutefois, ce domaine des caméras 3D TOF est très actif. On remarque par exemple la tenue d’un workshop conjointement à la conférence CVPR 2008 intitulé : Time of Flight Camera based Computer Vision.

8.1.3 Utilisation de lumière structurée

Dans cette approche un signal structuré est projeté sur l’objet. Par exemple, on va projeter des lignes ou un damier sur l’objet. En parallèle, la scène est visualisée par une caméra. En fonction de la déformation de la structure projetée, par triangulation, on peut retrouver la scène en 3D. Par exemple, dans [BBGK06], Bronstein et al. utilisent ce principe pour faire de la reconstruction de visage.

8.2 Introduction aux systèmes passifs

Les systèmes passifs que nous présentons sont tous des systèmes d’acquisition d’images. L’avantage de cette catégorie est que le matériel nécessaire est très modeste : une caméra ou un appareil photo grand public est suffisant dans une grande majorité des cas. On peut utiliser une seule image, plusieurs images prises à un même instant par plusieurs caméras, plusieurs images prises à une même position de caméra mais à des instants différents (i.e. une vidéo à partir d’une caméra statique) ou encore plusieurs images depuis une caméra mobile.

Lorsque l’on ne dispose que d’une seule image, le choix des méthodes de reconstruction est très limité et impose de très fortes hypothèses. On parle alors de Shape-From-Shading. Nous présentons cette première méthode dans le paragraphe 8.2.1. Mais toutes les autres méthodes de reconstruction 3D, plusieurs images sont utilisées. Cela implique avant tout de modéliser la vision par caméra. Nous introduisons dans le paragraphe suivant les grandes lignes de la vision par caméra, nécessaires à la bonne compréhension des sections suivantes dans lesquelles des méthodes de reconstruction à partir de plusieurs images sont présentées. Le problème de la calibration (i.e. détermination des paramètres) des caméras (autant interne qu’externe) étant considéré comme résolu pour la grande majorité des cas, nous n’en présentons pas ici tous les détails. Seuls les éléments nécessaires à une bonne compréhension de la suite du document sont traités. Pour plus de précision, on pourra consulter les livres classiques sur le sujet [HZ04, MSKS04].

Après avoir présenté ces notions de vision 3D, nous présenterons différentes méthodes qui reposent sur ces étapes pour aboutir à une représentation 3D des objets d’une scène.

8.2.1 Shape From Shading

On peut traduire ce terme “Shape From Shading” par forme à partir d’ombrage. Toutefois, le nom anglais nous paraît plus intuitif et est très utilisé, même dans la littérature française. Cette méthode utilise le niveau de gris d’une seule image pour en déduire la normale locale à la surface. La modélisation considère que la surface doit être parfaitement Lambertienne (i.e. émet ou réfléchit un rayonnement de la même manière quelle que soit la direction). La figure 8.3 illustre le principe de cette méthode. On trouve dans [DFS04, Koz98, ZTCS99] des revues des différentes méthodes existantes et dans [PCF06] une approche récente. La méthode de Shape From Shading peut par exemple être utilisée pour obtenir une carte d’élévation de la surface lunaire à partir d’une photo. On trouve aussi des cas d’utilisation sur des photos de visage 8.4. Du fait des conditions restrictives d’utilisation de cette approche, elle ne paraît pas du tout adaptée pour traiter le cas d’une surface de voiture, très fortement réfléchissante et avec des parties transparentes.

Pour gérer plusieurs images dans une méthode de reconstruction, il est nécessaire d’établir les relations géométriques entre les caméras et donc de modéliser la vision 3D. C’est ce que nous présentons dans les paragraphes suivants.

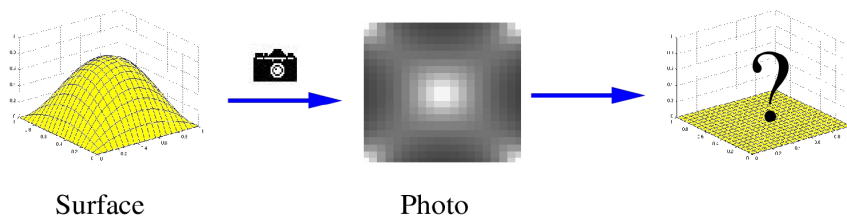


FIGURE 8.3 – Principe du shape from shading. A partir de l'image centrale, on souhaite reconstruire la forme initiale (image de gauche).

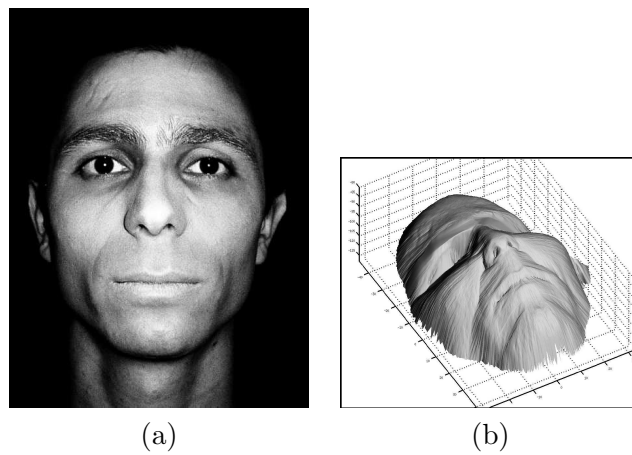


FIGURE 8.4 – Exemple de résultats issus du shape from shading. (Images extraites de [PCF06]).

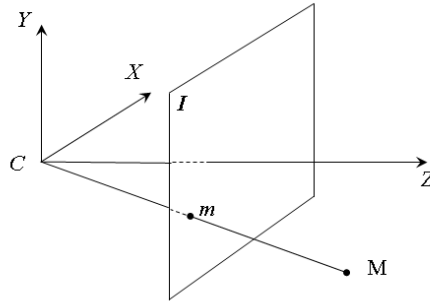


FIGURE 8.5 – Modèle de caméra pinhole. C est le centre optique. Le point 3D M est projeté dans le plan image I au point m .

8.2.2 Vision 3D

Le modèle de caméra généralement utilisé en vision par ordinateur est celui du trou d'aiguille (pin-hole). Une caméra est associée à un repère de projection. Par convention l'axe X de ce repère est l'axe horizontal dans le plan focal, vers la gauche. L'axe Y est l'axe vertical dans le plan focal. Et l'axe Z pointe vers l'avant, suivant l'axe optique. Ce repère est centré au centre optique. La figure 8.5 illustre le principe de la caméra pin-hole.

Lorsque l'on projette un point du monde 3D dans le plan image de cette caméra, la première étape est d'obtenir les coordonnées 3D de ce point dans le repère de la caméra. Ensuite, la caméra projette ce point dans le plan image. Ces deux étapes bien distinctes correspondent aux deux ensembles de paramètres de la caméra. Les paramètres qui donnent la position et l'orientation du repère de la caméra dans le repère canonique (appelé aussi repère monde) sont appelés ses paramètres externes. Les paramètres qui permettent de projeter un point depuis le repère 3D de la caméra dans le plan image s'appellent les paramètres internes de la caméra.

Ces deux étapes se modélisent simplement : soit M la représentation homogène d'un point 3D de coordonnées non homogènes $M' = (X, Y, Z)$ dans le repère monde :

$$M = (X, Y, Z, 1)$$

Ce point est projeté dans le plan image au point $m = (x, y, w)$:

$$m = K \times P \times M \tag{8.1}$$

La matrice 3x3 K correspond aux paramètres intrinsèques (internes) de la caméra. La matrice 3x4 P correspond aux paramètres externes. Il s'agit en fait de la matrice de changement de repère entre le repère canonique et le repère de la caméra. A noter qu'on obtient un point 2D en coordonnées homogènes, le point de coordonnées non homogènes est $x' = (x/w, y/w)$.

Calibration interne

La calibration interne est l'étape de détermination des paramètres de la matrice de projection K pour une caméra. La méthode classique consiste à prendre des images d'un objet dont on connaît le motif (e.g. un échiquier dont on connaît la taille des cases). La matrice de calibration est de la forme :

$$K = \begin{bmatrix} f_x & 0 & O_x \\ & f_y & O_y \\ & & 1 \end{bmatrix} \quad (8.2)$$

Dans cette matrice, f_x est la focale en x et f_y la focale en y (pour une caméra de bonne qualité, on aura $f_x = f_y$). Le point (O_x, O_y) représente le point central de l'image (intersection de l'axe optique et du plan focal). Pour une caméra idéale, $O_x = O_y = 0$, c'est-à-dire que l'axe optique intersecte exactement le plan focal au milieu de l'image, ce qui est faux en pratique, spécialement pour des caméras bon marché. A noter que le paramètre de "skew" a été négligé ici (hypothèse valable pour une caméra moderne standard).

Il existe plusieurs moyens d'obtenir ces paramètres de la caméra. Le plus classique est de prendre plusieurs photos d'un objet dont les coordonnées 3D sont connues dans le repère local (celui de l'objet). Ensuite, on va estimer pour chaque photo la calibration externe (la position de la caméra) et l'on va calculer les paramètres internes qui minimisent l'erreur de rétroprojection.

Dans notre cas, nous avons utilisé un logiciel (GML calibration toolbox) pour obtenir la calibration à partir d'un répertoire contenant les photos d'un échiquier de dimensions connues.

La modélisation de caméra présentée ci-dessus ne prend pas en compte la distorsion radiale. Cette distorsion a tendance à rendre courbes les lignes photographiées. Ce phénomène est particulièrement visible avec un objectif grand angle. La modélisation généralement utilisée pour ce problème est :

$$x' = x(1 + a_1 r^2 + a_2 r^4) \quad (8.3)$$

$$y' = y(1 + a_1 r^2 + a_2 r^4) \quad (8.4)$$

où (x, y) sont les coordonnées du point avec la distorsion, (x', y') les coordonnées corrigées, et $r^2 = x^2 + y^2$ et a_1 et a_2 deux nouveaux paramètres de la calibration interne de la caméra. L'approche communément utilisée pour calculer ces paramètres est celle dite de Tsai [Tsa86]. On peut aussi chercher à estimer un modèle de distorsion radiale en détectant dans les images les courbes qui devraient être des droites, et optimiser les paramètres pour les rectifier en des droites (voir [DF95]).

Calibration externe

Nous présentons désormais comment retrouver les positions (translations et rotations) des caméras. Seuls les éléments qui permettent de comprendre la méthodologie pour deux vues sont présentés.

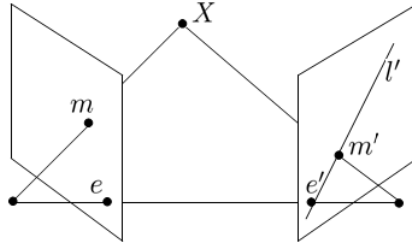


FIGURE 8.6 – m et m' sont les deux projections d'un même point 3D X dans deux plans images de deux caméras. e et e' sont les deux épipoles. l' est la droite épipolaire associée au point m dans la seconde caméra.

Matrice fondamentale Soit m un point sur la première image qui correspond au point m' sur la deuxième image (voir figure 8.6). Ces deux points sont reliés par :

$$m'^T F m = 0 \quad (8.5)$$

F est une matrice 3×3 appelée matrice fondamentale. Elle contient toute l'information sur la calibration de la caméra et sur la position des deux caméras (paramètres internes et externes). Une caractéristique utilisée par la suite est qu'elle est de rang 2.

Un point m sur la première image appartient forcément à la droite Fm sur la seconde image (Fm représente la droite en notation tensorielle). Une telle droite est appelée une droite épipolaire.

Epipoles On appelle épipole la projection d'un centre optique d'une caméra dans le plan focal d'une autre caméra. Les épipoles sont les noyaux à droite et à gauche de F .

Matrice essentielle Si l'on connaît la calibration, on peut définir pour chaque point m le point $u = K^{-1}m$. Alors on a la relation :

$$u'^T E u = 0 \quad (8.6)$$

la matrice E est appelée matrice essentielle. Si l'on connaît F , on peut l'obtenir par :

$$E = K^T F K \quad (8.7)$$

Elle contient l'information concernant les deux positions des caméras utilisées. Soit T la translation entre les deux positions des caméras et R la rotation, on a :

$$E = \widehat{T} R \quad (8.8)$$

où \widehat{T} est la matrice antisymétrique construite à partir du vecteur translation $T = (T_x, T_y, T_z)$ de la caméra :

$$\widehat{T} = \begin{bmatrix} 0 & -T_z & T_y \\ T_z & 0 & -T_x \\ -T_y & T_x & 0 \end{bmatrix} \quad (8.9)$$

L'objectif est d'utiliser les points suivis en 2D pour estimer E et ensuite déduire R et T .

Calcul de F Soit m et m' deux projetés d'un même point 3D M sur deux images distinctes. Ils sont reliés par F :

$$m'^T F m = 0 \quad (8.10)$$

La contrainte ci-dessus peut aussi être traduite par un système linéaire :

$$A f = 0 \quad (8.11)$$

où f est un vecteur colonne d'inconnues, contenant les 9 éléments de la matrice F . Chaque ligne de la matrice A traduit la contrainte imposée par un couple de points (m, m') .

On remarque que l'on ne peut obtenir la solution (i.e. la matrice F) qu'à un facteur d'échelle près (si F est solution de 8.10 alors λF l'est aussi).

A partir de 8 points idéaux (non bruités), on peut donc résoudre le système linéaire et obtenir F à un facteur d'échelle près.

Algorithme à 8 points A cause du bruit, on va utiliser 8 points pour minimiser $\|A f\|$ sous la contrainte $\|f\| = 1$. La solution est le vecteur singulier correspondant à la plus petite valeur singulière obtenue par décomposition SVD de A . Le vecteur f obtenu forme une matrice F_{tmp} qui n'est pas forcément de rang 2. Il faut la projeter sur l'espace des matrices fondamentales (de rang 2). Pour cela, on calcule sa décomposition SVD : $F_{tmp} = U \times \text{diag}\{\sigma_1, \sigma_2, \sigma_3\} \times V^T$ et on obtient :

$$F = U \times \text{diag}\{\sigma_1, \sigma_2, 0\} \times V^T \quad (8.12)$$

où $\text{diag}\{a, b, c\}$ est une matrice diagonale dont les valeurs sur la diagonale sont a, b et c .

Il a été montré que l'algorithme était très instable. Cette instabilité rend l'algorithme presque inutilisable si, pour un point 2D (x, y) , on utilise sa représentation homogène $(x, y, 1)$. Il est prouvé dans [Har97] que l'algorithme est beaucoup plus stable en centrant les points (e.g. en prenant l'origine au centre de l'image au lieu du coin en haut à gauche) et en utilisant une coordonnée homogène égale à la moitié de la taille de l'image (au lieu de 1).

Algorithme à 7 points Si la matrice A a 7 lignes, le noyau de A est de dimension 2, généré par F_1 et F_2 (i.e. $F = F_1 + \alpha F_2$). De plus, on sait que F est de rang 2, donc $\det(F) = 0$ ce qui amène un polynôme cubique en α :

$$\det(F_1 + \alpha F_2) = 0 \quad (8.13)$$

En résolvant, on aboutit à une ou trois matrices fondamentales possibles.

Estimation robuste de F en utilisant un algorithme RANSAC Le problème des algorithmes précédents est que le calcul est très instable si certaines paires de points ne sont pas de vraies correspondances ou sont des correspondances imprécises (on parle d'outliers). On peut utiliser l'algorithme à 8 points avec beaucoup de points (8 est un minimum), mais cela n'évite pas que, si certains de ces points sont mauvais, le résultat sur F sera très instable. Pour rendre l'algorithme robuste, on va utiliser un algorithme de type RANSAC (voir [FB81]).

Cette classe d'algorithmes va tester un très grand nombre de solutions possibles, chaque solution utilisant le plus petit nombre de points possibles. Dans notre cas, on va tirer au hasard 7 couples de points, calculer F correspondant à ces 7 points, et tester si tous les autres couples de points correspondent à cette matrice F . Les points pour lesquels $m'^T F m$ est petit sont dits inliers, les autres outliers. Au final, on garde la solution qui a le plus de points inliers.

En fait, dans l'algorithme RANSAC, tester, pour chaque couple de points, $m'^T F m$ a peu de sens. Il s'agit d'une erreur algébrique qui n'a pas de signification physique. On préférerait tester l'erreur de rétroprojection. Toutefois, il faut pour cela calculer pour chaque matrice F possible la reconstruction, ce qui est gourmand en calcul. On préférera utiliser la distance de Sampson qui est une approximation à l'ordre 1 de l'erreur de rétroprojection. On trouvera plus d'informations sur cette distance page 165 de [MSKS04] ou page 32 de [Tor02].

Optimisation non linéaire L'algorithme précédent fournit la matrice F obtenue à partir de 7 points, qui satisfait le plus grand nombre de couples de points. Toutefois, on peut affiner le calcul. Une première solution rapide est d'utiliser l'algorithme 8 points en n'utilisant que les inliers.

Comme pour l'étape utilisant l'algorithme RANSAC décrite dans le paragraphe précédent, on peut améliorer le résultat en cherchant à minimiser l'erreur de rétroprojection, mais cela reste gourmand en ressources. On cherchera plutôt à minimiser la distance de Sampson avec une optimisation non linéaire. On trouvera des détails dans [TM97] et [TZ00].

Calcul des positions des caméras à partir de E Les étapes précédentes ont abouti à une estimation de la matrice E (via la connaissance de F et K). On va utiliser la relation : $E = \hat{T}R$ pour calculer les positions des caméras. L'exercice revient à factoriser E pour obtenir R et T .

On note $E = U\Lambda V^T$ la décomposition SVD de E . On définit W et Z telles que :

$$W = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ et } Z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (8.14)$$

On obtient alors pour la translation : $\hat{T} = UZU^T$.

Pour la rotation, il y a deux solutions possibles :

$$R_1 = UWV^T \text{ et } R_2 = UW^T V^T \quad (8.15)$$

On fixe la première caméra à $T = (0, 0, 0)$ et $R = I_d$. Ce qui donne pour cette caméra la matrice de projection $P = K[I_d|0]$. Pour la seconde caméra, on a quatre possibilités :

$$P'_0 = K[R_1|T] \text{ ou } P'_1 = K[R_1|-T] \text{ ou } P'_2 = K[R_2|T] \text{ ou } P'_3 = K[R_2|-T] \quad (8.16)$$

Pour savoir quelle position est correcte pour la seconde caméra, on va reconstruire les points 3D pour chacune des 4 possibilités. Ensuite, on ne gardera que celle pour laquelle tous les points sont devant la caméra. A noter que cette étape de reconstruction du nuage de points 3D fait partie intégrante de la calibration externe : le résultat final des étapes présentées ici consiste en la connaissance des calibrations externes des caméras ainsi qu'un nuage de points.

Reconstruction des points 3D Grâce aux algorithmes précédents, on a abouti aux deux matrices de projections P_0 et P_1 pour les deux positions de caméras. Pour chaque point 2D de chaque caméra, on peut donc tracer la droite dans l'espace qui relie le centre optique de la caméra au point dans le plan focal. Quand on fait cela sur les deux caméras pour un même point 3D, les deux droites s'intersectent dans l'espace exactement à la position du point 3D. Toutefois, cela n'est vrai que pour des points non bruités. En pratique, les droites ne s'intersecteront pas, il faut alors décider de la position du point 3D qui minimise l'erreur de rétroprojection.

On trouve dans Hartley et al. [HS97] une étude comparative très complète de ce problème. Nous exposons ici la solution qui est utilisée actuellement dans notre implémentation :

On a, pour une caméra : $m = PM$ avec $m = w(u, v, 1)^\top$. On note p_i les éléments de la ligne i de la matrice P (p_i est représenté sous forme d'un vecteur colonne). On peut alors réécrire l'équation $m = PM$ par :

$$wu = p_1^\top M, \quad wv = p_2^\top M, \quad w = p_3^\top M \quad (8.17)$$

En éliminant w de ces équations, on arrive au système :

$$up_3^\top M = p_1^\top M \quad (8.18)$$

$$up_3^\top M = p_2^\top M \quad (8.19)$$

Pour deux vues, on peut donc créer un système de quatre équations du type $AM = 0$ permettant de trouver les 4 coordonnées de M . On peut minimiser AM sous la contrainte $\|M\| = 1$. La solution est alors le vecteur propre de norme unitaire correspondant à la plus petite valeur propre de la matrice $A^\top A$. On trouve ce vecteur propre par une décomposition SVD de la matrice A .

Dans [PGV+04], les auteurs utilisent ces techniques pour obtenir des nuages de points 3D à partir d'une acquisition vidéo d'un objet immobile, faite avec une caméra mobile. Les auteurs n'utilisent pas d'étape distincte de calibration interne mais utilisent la reconstruction pour calibrer la caméra (on parle alors d'autocalibration). Ils commencent par obtenir le nuage de points ainsi que les positions des caméras pour deux vues. Ensuite, ils ajoutent des vues de manière itérative. A chaque vue ajoutée, les correspondances de points avec une vue

déjà utilisée permettent de calculer la position de la caméra de la nouvelle vue. Les nouvelles positions des points 2D permettent d'affiner les positions 3D des points, et de nouveaux points 3D sont rajoutés au nuage. Enfin, une étape de “bundle adjustment” permet de minimiser les erreurs sur les positions des caméras ainsi que sur les positions des points 3D. Ces étapes présentées dans [PGV⁺04] sont une application typique du livre [HZ04].

Dans les paragraphes précédents, nous avons présenté les étapes d'un algorithme classique de calibration externe (et donc aussi de reconstruction d'un nuage de points 3D). Le résultat final est la connaissance des calibrations des caméras ainsi qu'un nuage de points 3D qui appartiennent à la surface de l'objet étudié. Pour certaines applications, ce résultat est déjà intéressant, il peut par exemple permettre de calculer une boîte englobante de l'objet. Pour aller plus loin, il est souvent souhaitable d'obtenir une surface proche de la surface de l'objet, et non pas un nuage de points 3D. La première classe de méthode pour faire cela est donc naturellement celle qui tente d'approcher le nuage de points 3D obtenu ici par une surface. Nous la présentons dans le paragraphe suivant (partie 8.3). Ensuite, nous présenterons d'autres méthodes passives de reconstruction 3D. Dans toutes ces parties, nous considérons que les étapes de calibration interne et externe sont des problèmes résolus. Il s'agit alors uniquement de savoir comment aboutir à un objet 3D à partir des images.

8.3 Méthodes par approximation de nuages de points

Nous avons vu dans le paragraphe précédent que l'étape de calibration externe aboutit, en plus des positions des caméras, à un nuage de points 3D. Une façon de reconstruire l'objet est donc d'interpoler ou d'approximer ces points par une ou plusieurs surfaces. Nous présentons ici quelques-unes des méthodes possibles.

8.3.1 Méthodes 2D et demi

Delaunay 2D

On appelle méthode 2D et demi la reconstruction d'une surface 3D où celle-ci est une surface d'élévation (i.e. une fonction $f(x, y)$ définie sur un plan). Une des méthodes les plus anciennes dans cette catégorie est celle de [Har93]. Les points 3D sont projetés dans un plan et ensuite triangulés selon une triangulation de Delaunay. Le maillage 3D est construit en utilisant les relations de voisinage trouvés en 2D, mais entre les points 3D. Cette approche souffre de plusieurs problèmes. Tout d'abord, il s'agit d'une interpolation donc la surface passe exactement par tous les points 3D. Cela pose problème si certains points sont erronés (points dits outliers). Ensuite, le plan de projection influe fortement sur la reconstruction. Si, par exemple, la normale à la surface est quasi-orthogonale à la normale du plan de projection, la surface reconstruite sera très mauvaise.

Surface spline d'approximation

On peut aussi citer d'autres méthodes comme l'approximation par surfaces splines. Si on appelle S la surface reconstruite et $\{p_i\}$ l'ensemble des points à approcher, on cherche la surface qui minimise une énergie du type :

$$E(S) = \sum_{i=1}^n d(p_i, S)^2 \quad (8.20)$$

L'une des difficultés de cette approche est qu'il faut également trouver la meilleure paramétrisation $t_i = (u_i, v_i)$ de p_i (i.e., celle qui minimise la distance entre p_i et S). Il s'agit donc d'une double minimisation qui est effectuée de manière alternative. C'est-à-dire que l'on trouve une paramétrisation t_i pour les points à approcher, et ensuite, on trouve S qui minimise l'erreur ci-dessus, et on répète cette opération jusqu'à convergence.

On appelle "foot point" de p_i le point de la surface $S(t_i)$. Dans l'idéal, $S(t_i)$ est le point de la surface le plus proche de p_i . Trouver ce t_i pour un point p_i est un problème en soi (étape aussi appelée "foot point computation"). Dans [WPL06, Flo], trouver t_i pour un point p_i se fait en deux étapes. On commence avec une solution initiale simplement trouvée, par exemple en échantillonnant la surface et en prenant l'échantillon le plus proche de p_i . Dans [WPL06], les auteurs divisent l'espace en une grille régulière dans laquelle ils mettent un index vers le point de la surface le plus proche dont la normale passe par cette case. Une fois cette position initiale trouvée, on utilise une minimisation type Newton pour affiner le résultat (détails donnés dans [Flo]).

La seconde étape consiste, en utilisant la paramétrisation donnée à trouver les points de contrôle de la surface spline. Pour cela, on minimise la fonction $E()$ en fonction des points de contrôle. En pratique, un terme de lissage est rajouté. Il faut aussi pouvoir dériver la fonction distance par rapport aux points de contrôle du spline. Mais l'expression de la distance entre un point à approcher et la surface spline vaut

$$d(p_i, S) = \min_{p \in S} \| (p - p_i) \| \quad (8.21)$$

et n'est pas dérivable par rapport à $S(t)$. On va donc minimiser une approximation de cette fonction. La première idée est celle de [HSW89], où l'on cherche à minimiser la distance entre un point de donnée p_i et son foot point $p(t_i)$:

$$d(p_i, S) = \| p(t_i) - p_i \| \quad (8.22)$$

Une autre estimation de cette erreur est celle de [BI98], où la distance est projetée sur la normale à la surface au foot point. Enfin, une autre solution est proposée dans [PH03], qui approche mieux la fonction distance souhaitée (en utilisant une approximation à l'ordre 2). Dans [WPL06], les auteurs comparent ces trois méthodes dans le cadre d'approximation de courbes. Dans tous les cas compliqués, la dernière solution est la plus adaptée. Dans les cas simples, la méthode de [HSW89] donne aussi de bons résultats, plus rapidement. Et la seconde méthode [BI98], est parfois instable.

En pratique, il faut aussi s’assurer qu’il n’y ait pas d’auto-intersection (“wiggles”) de la surface. Pour ça, on rajoute une énergie de lissage dans l’énergie minimisée :

$$E(S) = E_d(S) + \lambda E_s(S) \quad (8.23)$$

avec :

$$E_d(S) = \sum_{i=1}^n d(p_i, S)^2 \quad (8.24)$$

et $E_s(S)$ l’énergie dite “thin plate energy” :

$$E_s(S) = \int_0^1 \int_0^1 \left(\left(\frac{\partial^2 s}{\partial u^2} \right)^2 + 2 \left(\frac{\partial^2 s}{\partial u \partial v} \right)^2 + \left(\frac{\partial^2 s}{\partial v^2} \right)^2 \right) dudv \quad (8.25)$$

On trouve dans [Flo98] des détails sur la façon d’incorporer cette régularisation dans le système d’équations.

8.3.2 Approximation de surface 3D

Une autre approche est d’utiliser un algorithme de maillage de nuages de points 3D basé sur une tétrahédrisation de Delaunay ([DG03, Ame99]. La librairie CGAL (Computational Geometry Algorithms Library, [CGA]) propose une implémentation de cette tétrahédrisation. Toutefois, ce type d’algorithme est gourmand en temps de calcul.

Toutes ces approches pour construire une surface 3D à partir du nuage de points souffrent du fait que le nuage de points 3D peut contenir des outliers, qu’il y a des régions sans points et sont difficiles à mettre en oeuvre (notamment si la topologie de l’objet est complexe).

8.4 Méthodes par analyse de l’espace 3D

Une autre approche pour aboutir à une représentation 3D de l’objet est d’analyser la scène 3D qui contient cet objet, en utilisant les images et les calibrations externes. C’est-à-dire qu’on va chercher à savoir quelles régions sont à l’intérieur de l’objet et lesquelles sont à l’extérieur. La première méthode de cette catégorie (Shape From Silhouette, partie 8.4.1) utilise les silhouettes dans chaque image pour effectuer cette segmentation de l’espace. La seconde méthode que nous présentons (Space Carving, partie 8.4.2) utilise une discrétisation de l’espace en voxels et itère un processus de décision pour conserver uniquement les voxels correspondant à l’objet photographié.

8.4.1 Shape From Silhouette

Cette méthode utilise comme donnée de départ des images prises à un même instant par plusieurs caméras statiques d’une même scène. Sur chaque image, la silhouette de l’objet étudié est extraite. Ensuite, une reconstruction 3D de l’objet étudié est calculée à partir de ces silhouettes. Une condition d’utilisation est qu’on étudie un (ou plusieurs) objets dont on peut extraire la silhouette sur

chacune des images fournies par les caméras. On peut regrouper les méthodes de reconstruction à partir de silhouette en deux catégories : les approches volumétriques et les approches surfaciques.

Dans une approche volumétrique, l'espace est discrétisé en une grille de voxels. La forme 3D peut être vue comme l'ensemble des voxels qui sont projetés à l'intérieur des silhouettes sur les différentes caméras. Cette approche est consommatrice en calcul à cause de la discrétisation de l'espace. De plus, avec les premières approches, la décision de dire si un voxel appartient ou non à la forme 3D est délicate en pratique car les silhouettes ne sont pas exactes. A noter que la reconstruction 3D est appelée l'enveloppe visuelle (ou visual hull [Lau94]). On trouve dans [SCMS01] une revue de ces méthodes volumétriques. Cette enveloppe visuelle est par exemple utilisée dans les travaux de Hernandez et al. [ES04] pour initialiser une surface déformable.

Dans une approche surfacique, la forme 3D est reconstruite comme un polyèdre. Ce polyèdre est obtenu par l'intersection des projections 3D des silhouettes dans l'espace. Cette étape de reconstruction de polyèdre est réputée complexe mais aboutit à des résultats plus précis que les méthodes volumétriques. On trouve dans [CG00] une étude de ces méthodes. Dans [MAF⁺04], les auteurs utilisent cette méthode sur la plate-forme GrImage de l'INRIA de Grenoble.

Ces deux approches reposent sur une extraction précise de la silhouette. Dans les applications qui utilisent ces méthodes, la scène est généralement filmée avec un fond facile à enlever (e.g. un fond vert). Si cette première étape comporte des erreurs, il sera d'autant plus difficile d'obtenir une reconstruction 3D correcte.

Dans notre cas avec un environnement réel et un objet réfléchissant, avec des transparences, cette étape d'extraction serait compliquée à mettre en oeuvre. De plus, à tout instant, il faut disposer de plusieurs images de la scène, ce qui complexifie le système d'acquisition. Dans notre cas, ces contraintes nous paraissent difficiles à surmonter pour appliquer dans de bonnes conditions une méthode de Shape From Silhouette.

8.4.2 Méthodes de Space Carving

Au lieu d'utiliser les silhouettes pour séparer les voxels entre intérieur et extérieur comme dans le Shape From Silhouette, certains auteurs ont proposé de vérifier la cohérence de la couleur entre plusieurs images [SD99]. C'est-à-dire qu'un voxel appartient à la surface de l'objet si sa couleur est cohérente entre ses différentes projections sur les images des caméras. La méthode initialement la plus connue sur ce principe s'appelle space carving [KS00]. En partant d'une grille où tous les voxels sont marqués comme opaques, de manière itérative, on marque certains pixels comme transparents et d'autres comme appartenant à la surface de l'objet (avec connaissance de leurs couleurs). En marquant certains voxels transparents, on peut estimer la couleur de nouveaux voxels qui étaient cachés derrière ces voxels opaques, et ainsi de suite. D'autres auteurs ont ensuite amélioré les résultats en modifiant l'une des étapes de l'algorithme, mais le principe reste identique [BDC01, YPW03]. Le problème majeur de cette

approche est que si des voxels sont marqués transparents par erreur, les voxels derrière ceux-ci vont certainement aussi être marqués transparents. C'est-à-dire que l'erreur se propage facilement. Pour contrer cela, les auteurs utilisent un algorithme très conservateur dans l'élimination des voxels. Mais c'est aussi pour cette raison que les objets 3D obtenus sont généralement plus gros que le vrai objet. D'autre part, cette méthode n'est pas adaptée si l'objet contient de grandes régions de couleur uniforme, ce qui est le cas pour les voitures (e.g. le capot).

8.5 Méthodes de reconstruction cohérente avec les images

Dans cette partie, nous présentons les méthodes qui à partir des images et des calibrations externes des caméras, cherchent à obtenir une représentation 3D, cohérente avec les intensités de tous les pixels des images. La plus simple de ces méthodes est celle qui à partir de deux images calibrées, cherche à obtenir une carte des profondeurs. On parle alors de stéréovision (partie 8.5.1). Dans les paragraphes suivants, nous présenterons des méthodes qui cherchent à minimiser une erreur basée sur un critère local de photo-cohérence. Nous classons ces méthodes selon le type d'outil mathématique utilisé (surface déformable, Level-Set puis Graph-Cut). Enfin, dans une dernière partie (section 8.6), nous présenterons des approches qui utilisent aussi un critère de photo-cohérence local, mais sans rentrer dans les cadres décrits précédemment.

8.5.1 Stéréovision

En stéréovision, on dispose de deux images d'une même scène pour aboutir à une reconstruction 3D. On fait une distinction entre stéréovision dense et stéréovision à base de points (i.e. où le résultat est un nuage de points 3D). Dans le premier cas, le résultat est une connaissance 3D sur un domaine complet. En général, on construit une carte de profondeur associée à une image (i.e. pour chaque pixel, on connaît la profondeur à laquelle se trouve l'objet 3D). On parle aussi de carte de disparité. Dans ce cas, pour chaque pixel de coordonnées (x, y) d'une image, on cherche la position (x', y') qui lui correspond dans la seconde image. En général, on cherche à transformer les images pour avoir $y' = y$. C'est à dire qu'un même détail sur les deux images est projeté à la même ordonnée sur les deux images. On parle alors d'images rectifiées (voir figure 8.7). Cette rectification est possible en utilisant la géométrie épipolaire (voir [MSKS04]). On note alors $d(x)$ la disparité en (x, y) telle que :

$$x' = x + d(x) \text{ et } y' = y \quad (8.26)$$

On trouve dans [SSZ01] un état de l'art des méthodes de stéréovision dense. Cet article dispose d'un site Internet associé (vision.middlebury.edu/stereo/) qui permet d'évaluer comparativement les algorithmes présentés dans l'article ainsi que ceux apparus depuis sa parution. Dans notre cas, nous ne nous intéressons pas particulièrement à ces méthodes car elles reposent sur un couple

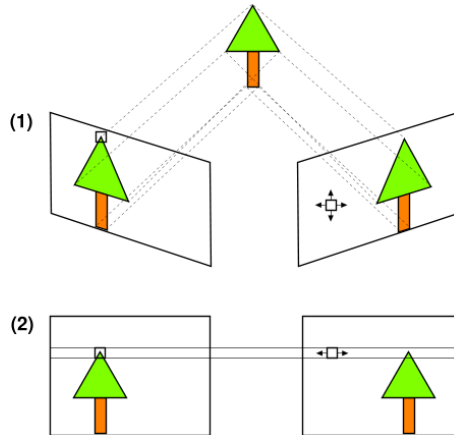


FIGURE 8.7 – Rectification d’images. (1) Images non rectifiées. Pour trouver sur l’image de droite le pixel correspondant au pixel du sommet de l’arbre sur l’image de gauche, l’espace de recherche est à deux dimensions. (2) Images rectifiées. Pour trouver le pixel du sommet du sapin sur l’image de droite, il suffit de parcourir la ligne de même ordonnée que ce pixel dans l’image de gauche. L’espace de recherche est à une seule dimension.

d’images. Nous disposons plutôt d’une vidéo et donc d’une multitude d’images de l’objet dont on cherche une reconstruction 3D. Dans les paragraphes suivants, nous allons donc plutôt nous intéresser aux méthodes dites de reconstruction 3D multi-vues qui cherchent à maximiser un score de photo-cohérence (ou inversement, à minimiser une erreur de photo-cohérence). Dans le paragraphe suivant, nous montrons le principe de ces méthodes. Ensuite, nous expliquons plusieurs algorithmes qui reposent sur ce principe mais utilisent des outils mathématiques différents.

8.5.2 Généricité des méthodes basées sur la couleur

Dans le Space Carving, on utilise un critère de photo-cohérence pour distinguer les voxels proches de la surface de l’objet de ceux extérieurs à l’objet. Une manière de formaliser cette approche est de voir la mesure de photo-cohérence comme une erreur locale et de chercher la surface 3D qui va minimiser la somme de ces erreurs.

Le critère local de photo-cohérence peut varier. Classiquement, il s’agit de la Normalized Cross Correlation (NCC). Mais la SSD (Sum of Squared Differences) peut aussi être utilisée. Ensuite, un algorithme va chercher à obtenir la surface de l’objet 3D qui minimise les erreurs mesurées par le critère local de photo-cohérence.

Dans les paragraphes suivants, nous introduisons plusieurs algorithmes basés sur ce principe en les classant selon l’outil de minimisation utilisé : surface déformable, Level-Set ou bien Graph-Cut. A noter que l’article [SCD+06] propose une méthodologie d’évaluation comparative de ces algorithmes. Sur le site associé (vision.middlebury.edu/mview/), on trouve les évaluations d’un

grand nombre d’algorithmes possibles sur deux objets 3D dont on connaît la vérité terrain (surface 3D mesurée par une acquisition laser).

Dans les paragraphes suivants, nous présentons les méthodes par déformation de surface, puis nous faisons quelques rappels sur les Level-Set et Graph-Cut ainsi que la façon dont ces outils sont utilisés pour faire de la reconstruction 3D multi-vues.

8.5.3 Méthodes par déformation de surface

Initialement, les contours actifs (ou Snakes) ont été proposés dans les travaux de Kass et al. [KWT88]. L’idée centrale est de faire évoluer un contour pour trouver les frontières d’un objet dans une image. L’évolution du contour correspond à la minimisation d’une énergie construite à partir d’un terme lié à l’image et d’un terme interne (e.g. un terme de lissage). Ces approches ont été utilisées pour le problème de la stéréovision ([KWT88, WTK87]) ou encore pour segmenter des images obtenues par résonance magnétique [CC93]. Nous présentons ici les travaux de Hernandez et al. [ES04] qui utilisent ce principe pour reconstruire une surface 3D qui minimise une erreur basée sur une mesure de photo-cohérence et sur le respect des silhouettes de l’objet.

Dans leurs travaux, la surface finale est obtenue par déformation de surface. Cette déformation est guidée par trois forces. La première est liée à la texture (force de photo-cohérence), la seconde à la silhouette et enfin, une dernière force sert de lissage. À noter que dans leurs travaux, l’acquisition est faite par une caméra fixe et c’est l’objet qui est mobile (placé sur une table tournante). Mais ce point ne fait aucune différence particulière avec une acquisition où l’objet est immobile et la caméra mobile.

La surface initiale utilisée est l’enveloppe visuelle. La force liée à la texture va attirer la surface pour faire augmenter un critère local de NCC entre les vues. Pour cela, pour chaque pixel de chaque image, on parcourt la demi-droite 3D correspondante, et on cherche la profondeur du point de cette demi-droite qui maximise la NCC avec les autres images. Une fois trouvée, on enregistre cette valeur de NCC dans une grille 3D, à la position du point 3D trouvé. Toutefois, ce volume ne peut pas être utilisé pour guider la surface déformable. On pourrait utiliser un gradient sur ce volume. Mais, ce gradient ne serait représentatif que proche de la surface. Les auteurs proposent d’utiliser un gradient vector flow (noté GVF) [XMPM98]. Cela permet d’attirer la surface même lorsqu’elle est loin des zones de forte NCC. La force de silhouette, similaire à celle de [MWTN04], dépend de la distance entre un point 3D retroprojeté et la silhouette sur une image. De plus, une force interne est rajoutée pour régulariser la surface triangulée. Cette force interne attire un sommet du mesh vers le centre de son voisinage (sommets qui lui sont liés par une arête, on parle de “1-ring”).

Cette méthode obtient de très bons résultats mais elle est très lente. Le principal inconvénient est qu’il est nécessaire de calculer la NCC sur tous les points d’une grille 3D.

En outre, il est possible de gérer les changements de topologie dans une approche par surface déformable (e.g. [DYQS04]). Mais cela amène en général une implémentation délicate pour la gestion de la surface triangulée. Dans nos

travaux, nous considérons que la topologie des objets est connue.

Une autre approche pour obtenir une surface 3D par minimisation d’une erreur basée sur un critère de photo-cohérence est d’utiliser les approches par Level-Set.

8.5.4 Level-Set

La notion de Level-Set a été introduite par Osher et Sethian dans [OS88]. On trouve dans [Set99] une explication détaillée du principe. Nous en présentons ici les grandes lignes afin de comprendre les algorithmes de reconstruction 3D qui l’utilisent.

Nous présentons le cas d’un Level-Set utilisé pour suivre l’évolution d’une interface 2D (i.e. une courbe), mais le principe est identique pour une interface 3D (i.e. une surface). Cela peut par exemple être le cas si l’on souhaite modéliser l’évolution de l’interface glace-eau pour un glaçon posé dans de l’eau.

Traditionnellement, pour suivre une courbe, on étudie les déplacements de ses points de contrôle soumis à des forces. On parle alors d’approche Lagrangienne. Les problèmes principaux de cette approche sont les possibles recouvrements (voir figure 8.8) et la difficulté à gérer les changements de topologie. La méthode par Level-Set permet de s’affranchir de ces problèmes en suivant une courbe implicite. On parle alors d’approche Eulérienne. La courbe fermée recherchée Γ est définie comme le niveau 0 d’une fonction φ définie sur le domaine 2D où se trouve la courbe :

$$\Gamma = \{(x, y) \mid \varphi(x, y) = 0\} \quad (8.27)$$

Au lieu de bouger les points de la courbe, on fait évoluer la fonction φ selon l’équation aux dérivées partielles (de Hamilton Jacobi) :

$$\frac{\partial \varphi}{\partial t} + F|\nabla \varphi| = 0 \quad (8.28)$$

où F est la force appliquée au contour (dans la direction orthogonale à celui-ci).

La figure 8.9 montre une courbe ainsi que la surface illustrant la fonction φ associée.

En pratique, l’espace d’étude est discrétisé selon une grille régulière. L’algorithme de suivi est itératif. A chaque étape, la force F en chaque point est calculée (pour une évolution simple, elle peut aussi être constante). Les dérivées spatiales de φ sont calculées pour exprimer $\nabla \varphi$. Enfin, en fonction d’un pas de la variable t choisi, on fait évoluer φ en chaque point de la grille. Après convergence, on extrait la courbe Γ comme le niveau 0 de la fonction φ (par exemple en utilisant un algorithme dit “marching squares”).

Pour une interface 3D, le principe est identique. La grille est une grille à 3 dimensions. L’interface cherchée est une surface. Pour extraire cette surface comme niveau 0 de la fonction φ , on peut utiliser un algorithme de “marching cube” [WEL87] ou de “marching tetrahedra” [TPG99]. En 3D, le temps nécessaire pour mettre à jour tous les points de la grille peut être long. Certains auteurs ont proposé de ne mettre à jour qu’une bande étroite des points

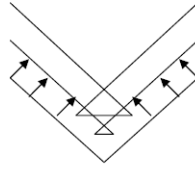


FIGURE 8.8 – Exemple de recouvrement lors de l'évolution d'une interface.

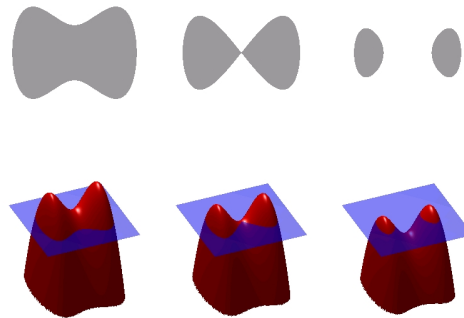


FIGURE 8.9 – Exemple de suivi de contour avec la fonction φ associée.

de la grille proche de l'interface à l'itération précédente [AS95] afin de réduire le temps de calcul nécessaire.

Il est à noter que l'approche Lagrangienne correspondante en 3D est le suivi d'une surface triangulée où l'on déplace chaque sommet en fonction des forces auxquelles il est soumis. Il est alors difficile, comme en 2D, d'éviter les recouvrements au fil de l'évolution de la surface, ainsi que de gérer les changements de topologie.

Après avoir introduit l'outil Level-Set dans ce paragraphe, nous présentons maintenant son utilisation dans le cadre de la reconstruction 3D.

8.5.5 Reconstruction de surface par Level-Set

Les méthodes par Level-Set pour la stéréovision ont été introduites dans Faugeras et al. [FK98]. Les auteurs commencent par introduire le principe pour la stéréovision avant de l'étendre à une reconstruction à n vues. On reprend ici le principe de la stéréovision pour comprendre leur méthode. On cherche la surface d'élévation f paramétrée par (x, y) (i.e. $z = f(x, y)$). Un point $M(x, y)$ de cette surface est projeté sur les deux images I_1 et I_2 aux points $m_1(x, y)$ et $m_2(x, y)$. On peut alors définir l'erreur de la surface f comme :

$$E(f) = \int \int [I_1(m_1(x, y)) - I_2(m_2(x, y))]^2 dx dy \quad (8.29)$$

On cherche alors la surface f qui minimise cette erreur. Pour cela, on utilise

l'équation d'Euler Lagrange associée :

$$(I_1 - I_2)(\nabla I_1 \times \frac{\partial m_1}{\partial f} - \nabla I_2 \times \frac{\partial m_2}{\partial f}) = 0 \quad (8.30)$$

Une approche classique pour résoudre ce problème est de considérer la fonction $f(x, y, t)$ aussi dépendante du temps, et de résoudre l'équation aux dérivées partielles :

$$\frac{\partial f}{\partial t} = \varphi(f) \quad (8.31)$$

où $\varphi(f)$ est égal au premier membre de l'équation d'Euler Lagrange ci-dessus. La seconde approche consiste à utiliser un Level-Set pour trouver la surface f qui minimise l'erreur définie. C'est cette seconde méthode que les auteurs utilisent. La surface est alors le niveau 0 d'une fonction définie sur une grille 3D.

Dans la suite de l'article, les auteurs présentent une façon d'étendre cette méthode à une véritable surface 3D qui n'est pas seulement une fonction d'élevation $f(x, y)$. Ils montrent des résultats mais la base publique de tests de [SCD⁺06] n'existant pas à l'époque, il est difficile de comparer ces résultats à d'autres. Un intérêt majeur de cette méthode est qu'elle gère les changements de topologie au cours de l'évolution. En pratique, les auteurs n'utilisent pas l'erreur très simple définie ci-dessus mais plutôt une intégrale où l'erreur locale est une cross-corrélation (et pas une simple différence d'intensité).

Dans Pons et al. [PKF07], les auteurs reprennent ce principe en l'étendant à une nouvelle erreur qu'ils appellent erreur globale. Reprenons l'exemple ci-dessus avec deux images. On note Π_i la matrice de projection de la caméra i . On note Π_i^{-1} la rétroprojection d'un pixel sur la surface approchée de l'objet S . Avec les mêmes notations que ci-dessus, l'erreur d'une surface S , pour le couple de caméras (1, 2) est définie par :

$$E_{1,2}(S) = \int \int [I_2(x, y) - I_1(\Pi_1(\Pi_2^{-1}(x, y)))]^2 dx dy \quad (8.32)$$

où l'intégration se fait ici sur les pixels de l'image I_2 . Pour n vues, on peut définir l'erreur globale d'une surface S comme :

$$E(S) = \frac{1}{n(n-1)} \sum_{i=1}^n \sum_{j=1, j \neq i}^n E_{i,j}(S) \quad (8.33)$$

Les auteurs présentent une méthode par Level-Set pour minimiser cette erreur. L'avantage de cette méthode est que le calcul de l'erreur locale se fait entre deux images d'un même plan focal. Dans l'équation ci-dessus, l'erreur est calculée entre l'image I_2 et l'image virtuelle $\Pi_2(\Pi_1^{-1}(I_1))$. On peut alors utiliser des fenêtres pour calculer les erreurs locales sans problème de distorsion. De plus, il n'y a pas de problème de calcul de visibilité (la visibilité est automatiquement gérée lors du calcul des images virtuelles de l'objet).

Les problèmes de ces approches par Level-Set restent que le minimum de l'erreur obtenu est un minimum local, et qu'il dépend de l'initialisation. Pour

lutter contre les minimums locaux, on peut utiliser une approche hiérarchique : la grille utilisée est d'abord à grande maille, puis on réduit la taille de ces mailles. De plus, le temps de calcul est souvent problématique pour les approches par Level-Set. Dans Labatut et al. [LKP06], les auteurs proposent une implémentation sur processeur graphique (GPU) pour accélérer un algorithme de stéréovision multi-vues par Level-Set. Cette accélération leur permet d'avoir une reconstruction en quelques secondes sur le jeu de test publique de [SCD+06].

Dans le paragraphe suivant, nous étudions les méthodes par Graph-Cut qui permettent aussi d'obtenir une surface 3D par minimisation d'une erreur basée sur un critère local de photo-cohérence. Nous présentons d'abord l'outil mathématique Graph-Cut avant de présenter les méthodes de reconstruction 3D qui l'utilisent.

8.5.6 Graph-Cut

L'optimisation par Graph-Cut est devenue une méthode très utilisée pour minimiser une énergie, notamment à travers les travaux de Boykov et Kolmogorov. Les Graph-Cut ont été utilisés dans des domaines variés de vision par ordinateur comme la restauration d'images [BVZ01, BVZ98], la stéréovision [KKZ03], la segmentation d'images [Boy03], la reconstruction 3D multi-vues [KZ02] ou bien encore l'imagerie médicale [BJ00]. Dans ces problèmes, on cherche à donner un label à chaque pixel d'une image (où chaque voxel d'un volume dans le cas 3D). Assigner la meilleure configuration de labels aux pixels est un problème d'optimisation. La méthode par Graph-Cut permet, en définissant un graphe sur les pixels et en cherchant la coupe minimale (équivalent au flot maximal), de minimiser une énergie. L'idée de base est de définir le graphe en fonction de l'énergie de telle manière que la coupe minimale sur le graphe corresponde à un minimum de cette énergie. L'intérêt est que l'on dispose d'un algorithme efficace pour chercher cette coupe minimale. Un point fort de cette méthode est que sous certaines conditions, le minimum trouvé est un minimum global (plus précisément, on connaît une borne maximale de la différence entre le minimum global et celui trouvé).

Les problèmes où l'on a deux labels correspondent à des applications de segmentation. On souhaite par exemple séparer un personnage du fond d'une image. Dans le cas de la reconstruction 3D, on peut vouloir séparer les voxels entre ceux intérieurs à l'objet et ceux extérieurs à l'objet. Le cas de la stéréovision représente un exemple à n labels où ceux-ci correspondent aux n valeurs de disparité possibles.

Lien avec les champs de Markov aléatoires

Ces problèmes peuvent être modélisés comme des champs de Markov aléatoires (notés MRF pour Markov Random Fields). Un champ de Markov aléatoire est représenté par trois ensembles : \mathcal{S} est l'ensemble des sites étudiés (e.g. l'ensemble des pixels pour une image), N est un système de voisinage sur ces sites et F un ensemble de variables aléatoires associées aux sites. Pour le système de voisinage $N = \{N_i | i \in \mathcal{S}\}$, chaque N_i est un ensemble de sites de

\mathcal{S} qui sont voisins du site i . Le champ aléatoire $F = \{F_i | i \in \mathcal{S}\}$ est constitué de variables aléatoires F_i qui prennent leurs valeurs notées f_i dans un ensemble de labels possibles : $L = l_1, \dots, l_n$. Un ensemble de labels f_i pour chaque site est appelé une configuration. La probabilité d'une configuration particulière $P(F = f)$ doit satisfaire la propriété de Markov pour que F soit un champ de Markov :

$$\forall i \in \mathcal{S}, P(f_i | f_{\mathcal{S}-i}) = P(f_i | f_{N_i}) \quad (8.34)$$

où $f_{\mathcal{S}-i}$ est l'ensemble des labels pris aux sites de \mathcal{S} sauf au site i . Cela revient à dire que la configuration en un site ne dépend que de la configuration du voisinage de ce site. On cherche alors la configuration f basée sur les observations D qui maximise la vraisemblance $P(D|f)$. En utilisant le théorème de Bayes, cette vraisemblance s'exprime comme une énergie $E(f)$ et le maximum a posteriori (MAP) de f maximise cette énergie.

Ce type de modélisation, relativement fréquent, aboutit à une énergie difficile à minimiser en pratique (fonction non convexe dans un espace à beaucoup de dimensions). La méthode la plus connue aboutissant à un minimum proche du minimum global est celle du recuit simulé [SD84]. Le problème majeur de cette méthode est son temps de calcul qui peut être très long. Les Graph-Cut apparaissent alors comme une alternative intéressante pour résoudre les problèmes de MRF en étant beaucoup plus rapides qu'un recuit simulé.

Modèles d'énergie

Le premier modèle d'énergie minimisable par Graph-Cut est du type (voir [KZ04]) :

$$E(f) = \sum_{p \in \mathcal{S}} D_p(f_p) + \sum_{(p,q) \in N} V_{p,q}(f_p, f_q) \quad (8.35)$$

où on a noté ici \mathcal{S} l'ensemble des pixels (i.e. ensemble des sites pour une image), f est l'étiquetage (mapping de \mathcal{S} sur l'ensemble des labels L), N est le voisinage utilisé. $D_p(f_p)$ est une mesure du coût pour assigner le label f_p au pixel p . $V_{p,q}(f_p, f_q)$ mesure le coût pour assigner les labels f_p et f_q aux deux pixels p et q voisins. Ce coût est utilisé comme terme de lissage. Il s'agit là d'une énergie qui dérive d'une modélisation par MRF.

En pratique, la minimisation n'est possible que pour certaines formes de fonctions $V_{p,q}$. La première forme est appelée modèle de Potts :

$$V_{p,q}(f_p, f_q) = T[f_p \neq f_q] \quad (8.36)$$

où la fonction T est un indicateur qui retourne 1 si son argument est vrai et 0 s'il est faux. On peut aussi utiliser pour V une fonction convexe de $|f_p - f_q|$, ou également des fonctions dites "robustes" (du type de celles utilisées pour les M-estimateurs, voir appendice 6 de [HZ04]).

La méthode par Graph-Cut permet de trouver un minimum global dans le cas où l'on a deux labels. Pour plus de labels, des méthodes permettent

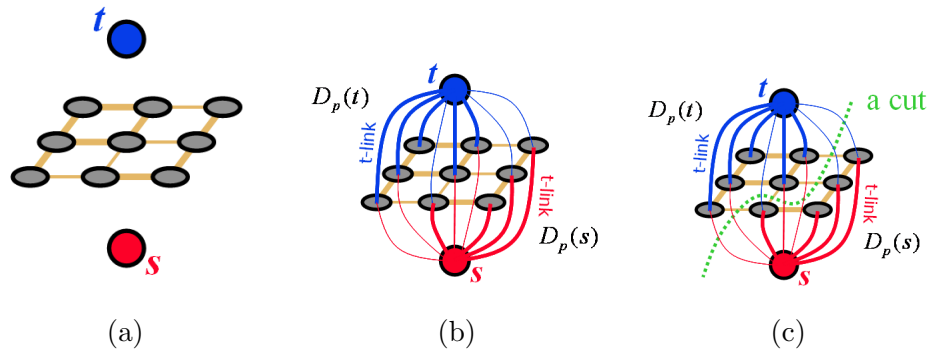


FIGURE 8.10 – (a) : Tous les noeuds du graphe ainsi que les arêtes entre les noeuds correspondant aux sites ($n - links$). (b) On a rajouté les arêtes avec les noeuds terminaux ($t - links$). (c) : Un exemple de coupe.

d’approcher le minimum global (e.g. α -expansion [KZ04]) en exécutant des minimisations itératives sur deux labels.

Minimisation

L’idée sous-jacente au Graph-Cut est d’utiliser une recherche de coupe minimale sur un graphe pour trouver le minimum d’une énergie. La première étape consiste à construire un graphe lié à l’énergie étudiée. Pour cela, considérons le cas où l’on n’a que deux labels.

Un graphe dirigé pondéré est noté $G = (E, V)$ où V est l’ensemble des noeuds et E l’ensemble des arêtes qui les connectent. On construit un noeud pour chaque site (e.g. pixel, voxel, ...). On rajoute ensuite des noeuds spéciaux appelés terminaux. L’un d’eux est appelé source, noté s et l’autre le puits, noté t . Ces deux noeuds correspondent aux deux labels. Pour chaque paire de noeuds correspondant à deux sites voisins p et q , on ajoute deux arêtes (i.e. une dans chaque direction) dont les coûts correspondent au terme $V_{p,q}$. Ces arêtes sont notées $n - links$. On crée aussi des arêtes entre chaque noeud de site et chaque noeud terminal, appelées $t - links$. Le coût d’un $t - link$ entre un noeud associé au site p et un terminal associé au label l correspond au coût d’assignation du label l au site p . Il est donc lié au terme D_p de l’énergie.

L’étape suivante consiste à trouver une coupe minimale dans le flux entre le noeud source et le noeud puits. Cette coupe segmente les noeuds en deux ensembles : ceux rattachés à la source et ceux rattachés au puits. Le résultat de la minimisation de l’énergie étudiée correspond à cette labellisation. La figure 8.10 montre deux étapes de la construction du graphe ainsi qu’une coupe.

Après avoir introduit l’outil Graph-Cut, nous pouvons revenir à son emploi dans le cadre de la reconstruction 3D multi-vues.

8.5.7 Reconstruction de surface par Graph-Cut

Dans une partie de la littérature, on trouve des articles où les auteurs utilisent un Graph-Cut pour faire de la stéréovision ([KZ02, BVZ01]). Ensuite, les cartes de profondeur obtenues sont mélangées pour ne créer qu'une unique représentation 3D (voir 8.6.1). Toutefois, le Graph-Cut n'est pas appliqué directement sur une représentation volumique.

Dans Vogiatzis et al. [VTC05], les auteurs proposent de définir un graphe sur les voxels qui permet de minimiser une énergie comprenant un terme de photo-cohérence et un terme de voisinage. Les noeuds de ce graphe sont les centres des voxels. Le coût d'une arête est un terme de photo-cohérence calculé au point 3D situé au point milieu des deux sommets de l'arête. Le calcul de ce terme nécessite la connaissance d'une surface approchée pour exprimer la visibilité d'un point 3D sur les images (les auteurs utilisent l'enveloppe visuelle). Chaque noeud est également connecté au noeud source selon une énergie similaire à une force ballon (introduite dans [Coh91]). Il est à noter que seuls les voxels contenus dans une écorce (épaisseur autour d'une surface) autour de l'enveloppe visuelle sont considérés dans ce graphe. Les noeuds des voxels sur les bords de cette écorce sont liés aux noeuds source et puits avec une énergie infinie. Ainsi, la coupe minimale trouvée correspond à une surface comprise dans l'écorce.

L'avantage de cet algorithme est d'utiliser une représentation volumétrique en utilisant un Graph-Cut. La modélisation est élégante car il s'agit d'un MRF avec un terme de photo-cohérence et un 6-voisinage entre les voxels. L'inconvénient est qu'il est nécessaire d'avoir une surface initiale pour calculer localement la photo-cohérence. Enfin, malgré l'utilisation de Graph-Cut, cet algorithme reste relativement lent (40 minutes sur certains jeux de données de [SCD+06]).

Dans Tran et al. [TD06], les auteurs modifient cet algorithme en rajoutant des contraintes pour forcer la surface à passer par certains points. Cela permet de mieux retrouver les protrusions ainsi que les concavités.

Dans Hornung et al. [HK06], les auteurs proposent une autre façon de construire le graphe sur les voxels. De plus, ils se basent sur une voxelisation hiérarchique de l'espace. C'est-à-dire qu'on va chercher la surface à un niveau peu précis, puis s'en servir pour améliorer la surface au niveau un peu plus précis. On part d'une surface grossière S_k (eg, l'enveloppe visuelle). Ensuite, au niveau k courant, on définit une écorce autour de S_k . Sur les voxels contenant l'écorce, on va calculer une mesure de photo-cohérence. Les auteurs utilisent une mesure des variances de couleurs, normalisée par échantillon, mais précisent que la NCC ou même une simple variance des couleurs (comme dans [SD99]) peut suffir. Un gros avantage de cet algorithme est que la photo-cohérence n'est calculée que sur l'écorce, et pas sur tout le volume. Cela est non négligeable car en général, c'est l'étape qui prend le plus de temps.

Ensuite, la surface qui minimise une énergie sur l'écorce est obtenue en cherchant une coupe minimale sur le graphe. Cette énergie est composée d'un terme de photo-cohérence et d'un terme cherchant à minimiser l'aire de cette surface. Pour cela, un graphe est défini sur les voxels. Ce graphe assigne un sommet à chaque face de voxel. Les auteurs ajoutent ensuite 12 arêtes dans

chaque voxel (formant ainsi un octaèdre dans chaque cube des voxels). Un poids représentant la photo-cohérence est donné à chaque arête. On donne aussi à chaque arête un poids constant pour favoriser les surfaces planes. Après la minimisation, on peut passer au stade $k + 1$ pour affiner la surface dans une écorce plus fine.

Les résultats sont comparables aux meilleurs algorithmes, mais grâce à la méthode hiérarchique et à l'écorce, il est plus rapide que les autres algorithmes par Graph-Cut sur les voxels.

8.6 Autres approches par critère de photo-cohérence local

Dans cette dernière partie, nous présentons deux approches qui utilisent un critère de photo-cohérence local, mais sans minimiser une énergie globale.

8.6.1 Méthode par mélange de carte de profondeur

De nombreux algorithmes (notamment ceux de stéréovision) permettent de calculer une carte de profondeur pour une vue (i.e. pour chaque pixel, on connaît la profondeur à laquelle se trouve l'objet). Avec n vues, on peut par exemple obtenir n cartes de profondeur. Le problème devient alors de fusionner ces informations pour obtenir une unique surface 3D. Dans [GCS06], les auteurs obtiennent de bons résultats avec cette approche (voir vision.middlebury.edu/mview/ lié au comparatif de [SCD⁺06]).

Pour obtenir la carte de profondeur de chaque vue, leur méthode est plutôt gourmande. On choisit une vue de référence. Pour chaque pixel de cette vue, on parcourt la demi-droite 3D correspondante (i.e. demi-droite d'extrémité le centre optique de la caméra et qui intersecte le plan image au pixel étudié). Pour chaque point sur cette demi-droite, on calcule le pixel correspondant sur chacune des autres vues. On calcule toutes les Normalized Cross Correlations (NCC). On prend le point de la demi-droite qui a de très bonnes NCC sur un maximum d'autres vues. Cela fournit la carte de profondeur pour cette vue de référence. On fait cela en prenant chaque vue comme vue de référence, et on obtient les cartes de profondeur pour toutes les vues. Il est à noter qu'il est nécessaire de seuiller sur les NCC pour ne pas considérer les points sans information et donc les cartes sont incomplètes.

Ensuite, les cartes de profondeur sont mélangées pour obtenir une surface 3D par la méthode de Curless et al. [CL96]. Cela peut être fait grâce au code source libre de l'article de Curless et al. [CL96].

Cette méthode a l'avantage d'être simple à mettre en oeuvre, et donne des résultats d'un niveau presque équivalent aux meilleures méthodes si l'on dispose de beaucoup de vues et que l'objet est bien texturé (pour bien répondre à la NCC). Mais un problème important est que si on dispose de peu de vues, ou de larges régions sont sans texture riche, la surface finale est trouée. Enfin, cet algorithme est très lent.

8.6.2 Méthodes par expansion de petits éléments 3D

Dans Furukawa et al. [FP06], les auteurs proposent une première méthode qui donne de très bons résultats sur la base de données de [SCD+06]. Ils commencent par trouver des correspondances affines entre des petites régions (patches) des images en utilisant l'algorithme de [MS02]. Cela leur permet d'obtenir des patches 3D orientés (i.e. dont on connaît la normale). Pour obtenir plus de patches, ils utilisent une méthode d'expansion qui permet en utilisant les patches existant d'en rajouter d'autres. Enfin, les patches en dehors de l'enveloppe visuelle et ceux qui sont cachés par d'autres patches sont supprimés. A partir de cet ensemble de patches 3D dense, les auteurs déforment la surface de l'enveloppe visuelle pour approcher les centres des patches. Ensuite, ils rajoutent un terme de photo-cohérence dans cette déformation de surface pour retrouver des détails plus fins. Le problème de cette approche est le temps de calcul très élevé.

Dans un second article [FP07], les mêmes auteurs proposent une autre méthode à base de patches 3D. De la même manière, la première étape consiste à créer un nuage de patches 3D non dense. Ensuite, les auteurs proposent un nouvel algorithme d'expansion pour rajouter de nombreux autres patches. Enfin, une étape de filtrage élimine les patches non cohérents avec la majorité des patches. Le résultat final est un nuage de patches 3D dense. L'avantage majeur de cette méthode est qu'elle n'utilise pas de connaissance a priori sur la surface (telle que l'enveloppe visuelle). Son inconvénient majeur est le temps de calcul.

A noter que dans [HK07], les auteurs proposent une méthode proche. Au lieu d'utiliser des patches, les auteurs reconstruisent des disques 3D. Ils utilisent aussi une méthode par expansion pour rajouter des disques à partir de ceux déjà existants.

8.7 Conclusion

Dans les paragraphes précédents, nous avons procédé à un état de l'art des différentes méthodes pour obtenir une information 3D à partir d'une acquisition d'images. Dans nos travaux, notre premier objectif est de proposer une méthode pour appliquer l'un des algorithmes de l'état de l'art pour reconstruire des voitures en 3D à partir d'une acquisition vidéo. Tout d'abord, nous décidons de ne pas considérer les approches qui nécessitent d'extraire la silhouette de l'objet. La raison est qu'à cause des transparences, cette silhouette serait trop difficile à extraire. D'autre part, nous souhaitons un algorithme quasiment temps réel. C'est-à-dire que si notre système doit être installé sur une autoroute, nous devons obtenir une reconstruction 3D des voitures au fur et à mesure de leur passage. Cela laisse seulement quelques secondes pour obtenir le résultat à partir de l'acquisition. Pour ces raisons, nous avons choisi d'utiliser les méthodes classiques de "Structure From Motion" (présentées en 8.2.2) qui permettent d'obtenir un nuage de points et ensuite d'approcher ce nuage de points par une surface, de manière robuste. A partir de ces résultats, nous chercherons à proposer une méthode pour affiner la reconstruction obtenue en utilisant un critère de photo-cohérence. On pourra aussi exploiter des connaissances a priori

sur l'objet reconstruit (un véhicule), ainsi que ses symétries.

Dans un deuxième chapitre, nous nous éloignons de la reconstruction de voiture pour étudier si les descripteurs SIFT peuvent permettre de nouvelles méthodes pour la reconstruction 3D. Dans cette partie, nous nous intéressons à reconstruire des objets qui sont classiquement utilisés pour tester ce genre d'algorithmes, notamment ceux de la base publique de [\[SCD⁺06\]](#). Pour cette approche, notre choix s'est porté sur une méthode par modèle déformable car elle offre une certaine souplesse pour intégrer des correspondances de SIFT. Nous avons alors cherché à intégrer des correspondances SIFT dans un algorithme inspiré de celui de Hernandez et al. [\[ES04\]](#).

Chapitre 9

Méthode proposée pour la reconstruction d’une voiture

Dans ce chapitre, nous introduisons tout d’abord les motivations industrielles qui nous ont poussé à chercher à reconstruire des véhicules en 3D. Ensuite, nous analysons en quoi ce problème est difficile au vu de l’état de l’art. Nous proposons et évaluons une méthode originale, développée pour aboutir à un modèle simple de voiture, mais de manière robuste (Auclair et al. [ACV07b]). Enfin, nous proposons des pistes pour améliorer le résultat obtenu, en cherchant à exploiter la symétrie d’un véhicule.

Sommaire

9.1 Motivations	180
9.2 Analyse du problème	180
9.3 Algorithme de “Structure From Motion” pour une translation	181
9.4 Reconstruction d’une surface à partir du nuage de points 3D	188
9.5 Evaluation	192
9.6 Perspectives	193
9.7 Conclusion	202



FIGURE 9.1 – Quelques images d’un film qui sert de donnée d’entrée à notre algorithme.

9.1 Motivations

La motivation originale de reconstruire des voitures en 3D est liée à une application de péage automatique. On souhaite étudier si, à partir d’acquisition vidéo sur une autoroute, il est possible de reconstruire les véhicules en 3D. L’objectif est ensuite de pouvoir calculer automatiquement la tarification en fonction des caractéristiques du véhicule (taille, nombre d’essieux...).

Un des problèmes que nous avons rencontrés pour ce sujet est le manque de données de la part de l’industriel partenaire. Les projets d’obtenir des acquisitions d’un site réel n’ont jamais vu le jour. Pour remédier à cela, nous avons donc réalisé nos propres prises de vues. Nous avons simplement posé une caméra sur un trépied au bord d’une route et filmé le passage de voitures. Sur nos images, nous n’avons qu’une voiture à la fois (voie en sens unique). Sur la figure 9.1, nous montrons 4 images extraites d’une telle séquence vidéo.

9.2 Analyse du problème

La reconstruction 3D de véhicule est un sujet complexe car l’hypothèse de surface Lambertienne n’est pas du tout respectée. Or cette hypothèse est requise par la grande majorité des méthodes qui utilisent un critère de photo-cohérence. Sans cela, l’utilisation des mesures locales de photo-cohérences (SSD ou NCC) est moins légitime. Or ces mesures sont généralement utilisées à toutes les étapes (e.g. suivi de points 3D par un algorithme de Lucas-Kanade [TK91], ou minimisation d’une erreur globale comme somme de photo-cohérences locales pour la reconstruction 3D...). Pour une voiture, certaines parties sont transparentes, et de larges parties sont uniformes. De plus, les roues font partie de l’objet mais ont un mouvement relatif par rapport à celui-ci. Cela crée beaucoup de difficultés pour une reconstruction sans a priori. Un algorithme de suivi de points aura du mal à faire la différence entre les points du véhicule et ceux dûs aux reflets de l’environnement sur ce véhicule. S’il y a beaucoup de reflets, le nombre d’outliers parmi les points suivis peut dépasser 50%.

Pour ces raisons, toutes les étapes de reconstruction sont rendues plus difficiles. Nous avons donc cherché quelles solutions sont envisageables pour que la calibration externe et la reconstruction d’une surface 3D soient robustes dans le cas d’un véhicule sur une vidéo dans un environnement extérieur.

La première étape de tout algorithme de reconstruction 3D est d’obtenir les calibrations externes des caméras (la calibration interne est supposée connue).

Pour une acquisition vidéo, la solution classique consiste à suivre des points au fil des images et d'en déduire les positions des caméras. Du fait de la difficulté du problème, sans aucune hypothèse sur le mouvement, les résultats sont peu satisfaisants. Nous avons donc décidé de faire l'hypothèse que la voiture est en translation pure. Cela nous permet de rendre plus robustes les étapes de suivi de points et de reconstruction d'un nuage de points 3D. Cette hypothèse paraît réaliste pour nos vidéos. Elle serait aussi certainement correcte sur un système placé au dessus d'une autoroute. En utilisant cette hypothèse, nous proposons de réaliser le suivi de points en deux passes. La première passe est un suivi de points 2D sur la séquence vidéo et permet de déterminer le point de fuite du mouvement. Une fois ce point de fuite obtenu, nous refaisons une passe de suivi où chaque point à suivre ne peut que s'éloigner du point de fuite. Cela permet d'aboutir à un nuage de points 3D bien plus dense. On montre ensuite qu'à cause de la surface complexe de certains véhicules, certains outliers ne sont pas filtrés par l'étape utilisant un algorithme RANSAC. On filtre alors ces points avec une méthode simple. Notre seconde contribution est de proposer une méthode pour obtenir le repère local de la voiture (i.e. un repère aligné sur la largeur, la hauteur et la profondeur de la voiture). Ensuite, ce repère local permet d'utiliser des connaissances a priori sur l'objet à reconstruire.

9.3 Algorithme de “Structure From Motion” pour une translation

On présente ici les étapes de l'algorithme classique de reconstruction 3D d'un nuage de points que nous utilisons. Nous introduisons le suivi 2D de points employé dans une première passe, un algorithme de recherche du point de fuite ainsi que l'adaptation de l'algorithme de Lucas-Kanade pour suivre les points avec connaissance du point de fuite.

9.3.1 Suivi de points 2D

Le suiveur de points de Lucas-Kanade est un algorithme très connu et largement employé dans la communauté de la vision par ordinateur. L'hypothèse sous-jacente est que les intensités autour d'un même point d'intérêt restent constantes entre deux images. Appliquée aux voitures, cette hypothèse a peu de chances d'être vérifiée, notamment à cause des nombreux reflets. De plus, les points d'intérêt à suivre doivent contenir une variation d'intensité dans deux directions orthogonales, c'est-à-dire être des coins. En utilisant le détecteur de points d'intérêt de Harris [HS88], il y a peu de tels points détectés sur une image de voiture (dans nos expériences, il n'y en avait pas assez pour obtenir une reconstruction correcte). C'est une conséquence du caractère uniforme de la texture sur les surfaces des voitures. Toutefois, en utilisant une implémentation multi-échelle (i.e. pyramidale) du suivi de Lucas-Kanade, tel que décrit dans [Bou00], une bonne proportion des points sont suivis correctement. C'est toutefois insuffisant pour obtenir un nuage dense de points 3D mais cela va nous permettre d'estimer le point de fuite du mouvement.

On rappelle ici brièvement le principe de ce suiveur de points (on trouve sa justification détaillée dans [TK91]). L'idée est de trouver pour une fenêtre de pixels dans l'image à l'instant t , la position de la fenêtre de pixels dans l'image à $t + 1$ qui minimise la différence d'intensité au carré sur ces pixels (notée SSD pour Sum of Square Differences). Le vecteur de déplacement des pixels de la fenêtre, noté d , est solution du système linéaire :

$$\left(\sum_{W_t} gg^\top \right) \cdot d = \sum_{W_t} (I_t - I_{t+1})g, \quad (9.1)$$

où W_t est la fenêtre autour du point d'intérêt suivi au temps t , I_t est l'intensité de l'image au temps t , I_{t+1} l'intensité de l'image à l'instant $t + 1$, g est le gradient d'intensité de l'image I_t . Ce système est appliqué de manière itérative (dite Newton-Raphson), pour trouver le déplacement qui minimise la SSD entre les positions des fenêtres à t et $t + 1$. On utilise ce principe pour suivre des coins de Harris dans la séquence vidéo. A partir des trajectoires ainsi obtenues, on cherche à estimer le point de fuite du mouvement.

9.3.2 Estimation du point de fuite

Avec l'hypothèse d'un objet en translation pure, toutes les trajectoires des points suivis en 2D sont des droites qui s'intersectent en un même point appelé point de fuite. Cela nécessite d'avoir supprimé la distorsion radiale des images. Ceci est une étape simple car nous supposons connus les paramètres internes de la caméra. Avec la connaissance de ce point, on pourra refaire une passe de suivi de points, mais avec une très forte contrainte (les points doivent s'éloigner du point de fuite), et donc rendre le suivi beaucoup plus robuste.

En pratique, le résultat du suivi de points 2D n'aboutit pas à un ensemble de trajectoires qui sont des droites s'intersectant en un unique point. Dans la littérature, on trouve des méthodes qui aboutissent au calcul du point de fuite de manière très précise (e.g. voir [ADV03]). Dans notre cas, nous proposons une méthode simple et robuste. Les trajectoires sont approchées par des segments de droites. Ensuite, nous utilisons un algorithme de type RANSAC [FB81] pour estimer le point de fuite. L'étape de jet aléatoire consiste à choisir deux segments au hasard et à calculer le point d'intersection de leurs droites porteuses (droite contenant le segment). Ensuite, on compte le nombre de segments qui ont une droite porteuse dont la distance à ce point d'intersection est inférieure à un seuil. Ce processus est itéré et le point d'intersection qui obtient le meilleur consensus (nombre de segments qui le valident) est choisi comme point de fuite. La figure 9.2 est une image qui illustre ce principe.

9.3.3 Suivi de points unidimensionnel

Connaissant le point de fuite, on peut simplifier l'équation (9.1) (avec une preuve quasi-identique à celle de [TK91]) en une équation à une seule dimension :



FIGURE 9.2 – Illustration du point de fuite. L'image de fond est une image de la séquence vidéo étudiée. Nous avons dessiné en vert les trajectoires qui s'intersectent en un unique point de fuite et en rouge les autres trajectoires (outliers du RANSAC).

$$\left(\sum_{W_t} (g \cdot dir)^2 \right) d = \sum_{W_t} ((I_t - I_{t+1}) \langle g, dir \rangle) , \quad (9.2)$$

où dir est le vecteur de norme unitaire indiquant la direction du point de fuite au point suivi et d est maintenant un scalaire représentant le déplacement pour minimiser la SSD selon cette direction. Un avantage important de cette seconde version est que pour être bon à suivre, un point a uniquement besoin d'avoir du gradient selon la direction du point de fuite. Cela permet de détecter beaucoup plus de points d'intérêt et d'en suivre correctement une grande proportion. Le résultat étant une meilleure donnée d'entrée pour l'algorithme de "Structure-From-Motion". La table 9.1 montre le nombre d'inliers (validés par les étapes suivantes de reconstruction 3D) que l'on suit pour chacune des deux méthodes, mesuré entre deux images consécutives. Pour les séquences testées, le gain en utilisant le suivi 1D est entre 2.4 et 5.1. Sur certaines séquences, le résultat du suivi 2D comporte un trop fort taux d'outliers pour être utilisé pour l'étape suivante de reconstruction du nuage de points 3D.

9.3.4 Calcul des positions des caméras et du nuage de points 3D

Pour cette étape, nous avons implémenté une méthode proche de celle de Pollefeys et al. [PGV⁺04], mais en adaptant les calculs à un objet en translation pure devant une caméra statique. Dans un premier temps, à partir de deux

TABLE 9.1 – Exemples de nombres de points suivis correctement sur plusieurs vidéos. Pour déclarer correct un suivi de point, on vérifie que ses positions sont cohérentes avec la reconstruction 3D obtenue.

	Suivi 2D	Suivi 1D	gain
Seq 1	138	405	2.9
Seq 2	118	508	4.3
Seq 3	73	370	5.1
Seq 4	55	134	2.4
Seq 5	75	348	4.6
Seq 6	37	120	3.2

images de la séquence vidéo, on construit une première scène (position des caméras ainsi que nuage de points 3D). Ensuite, d'autres images sont ajoutées, générant de nouvelles positions de caméra et de nouveaux points 3D. On détaille dans les paragraphes suivants ces étapes.

Reconstruction initiale à partir de deux images

On présente ici les étapes fondamentales qui permettent pour deux images, avec un objet en translation, de trouver les positions des caméras ainsi que le nuage de points correspondants. On note x un point dans l'image I_i et x' le point correspondant (trouvé par le suivi 1D) dans l'image I_j . On a vu dans l'état de l'art que ces deux points sont liés par l'équation fondamentale (voir [HZ04]) :

$$x'^T F x = 0, \quad (9.3)$$

où F est appelée matrice fondamentale. Cette matrice encapsule la relation spatiale entre les positions des deux caméras. Il existe une littérature vaste sur les algorithmes pour estimer cette matrice à partir de correspondances de points (voir section 8.2.2 et les livres [HZ04, MSKS04]). Toutefois, dans le cas d'une translation pure, cette matrice a une forme bien spécifique : une matrice antisymétrique 3×3 . Dans ce cas particulier, le point de fuite est aussi l'épipoles de la seconde image, que l'on note $e' = (x_e, y_e, z_e)$. La matrice fondamentale F s'exprime alors par :

$$F = [e']_x = \begin{bmatrix} 0 & -z_e & y_e \\ z_e & 0 & -x_e \\ -y_e & x_e & 0 \end{bmatrix} \quad (9.4)$$

On remarque que cette matrice est définie à un facteur d'échelle près (si F est une matrice fondamentale, alors λF avec λ réel l'est aussi). Dans le cas général, l'estimation de cette matrice peut se révéler instable. Pour cette raison, le fait de l'obtenir directement à partir du point de fuite permet d'éviter les potentiels problèmes de cette étape. Toutefois, un problème est que tous les

points suivis avec la méthode unidimensionnelle valident l'équation fondamentale. Il est donc impossible d'utiliser cette équation pour filtrer les paires de points qui ne la respectent pas, ce qui est traditionnellement fait dans le cas classique (i.e. positions des caméras quelconques). Cela signifie que le nuage de points 3D obtenu par cette seule étape pourra contenir des outliers.

En utilisant la calibration interne de la caméra obtenue en étape préliminaire, la matrice fondamentale permet d'obtenir directement les calibrations externes des caméras ([HZ04, MSKS04]). Finalement, le seul facteur à fixer est le facteur d'échelle, qui correspond à la distance séparant les deux caméras.

Une fois les positions des deux caméras fixées, chaque paire de points 2D génère un point 3D obtenu par triangulation. Comme la calibration interne est connue, la triangulation s'effectue dans l'espace Euclidien. Dans ce cas, une simple méthode linéaire donne de bons résultats (voir [HZ04, HS97]).

Ajout de vues supplémentaires

Dans le cas de la translation pure, l'ajout d'une vue est très simple. Il y a une seule inconnue qui est la position de la nouvelle caméra dans la direction de la translation. Chaque couple point3D-point2D (où le point 3D appartient au nuage déjà reconstruit et le point 2D est sa position dans la nouvelle image) génère deux équations de projection. Un seul de ces couples permet d'estimer la position de la nouvelle caméra. Grâce au faible nombre d'échantillons nécessaire à l'estimation du modèle, un algorithme RANSAC est tout à fait approprié pour déterminer la position de la nouvelle caméra. La figure 9.3 montre un nuage de points 3D ainsi que les caméras associées.

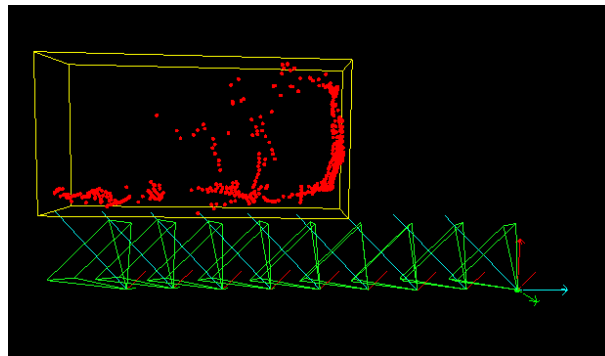
9.3.5 Filtrage du nuage de points 3D

La figure 9.4 montre un nuage de points 3D obtenu par l'algorithme précédent. Ce nuage de points comporte toujours des outliers car cet algorithme ne contient aucune étape de filtrage de ces mauvais points. Le nuage initial est construit à partir de deux vues. Ensuite, d'autres vues et de nouveaux points sont rajoutés, ceux existant sont affinés, mais à aucun moment des points ne sont rejetés. Dans cette partie, nous proposons d'utiliser deux filtres simples qui permettent d'éliminer efficacement les outliers sur les nuages de points 3D obtenus.

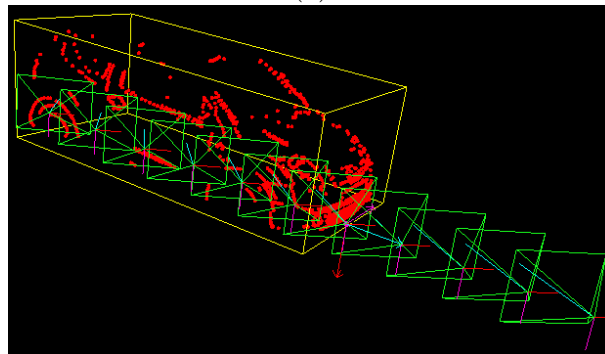
Le premier filtre consiste à imposer un seuil maximal τ sur l'erreur de rétro-projection. Pour un point 3D X_i , son erreur de rétro-projection $err(X_i)$ est définie par :

$$err(X_i) = \max_{c \in \mathcal{F}_i} (\|P_c(X_i) - x_c\|) \quad (9.5)$$

où \mathcal{F}_i est l'ensemble des vues à partir desquelles le point X_i a été reconstruit, P_c est la matrice de projection pour la vue c et x_c la position 2D de ce point (i.e. la position donnée par le tracker de Lucas-Kanade) dans cette vue. Un point 3D X_i est déclaré outlier si $err(X_i) > \tau$.

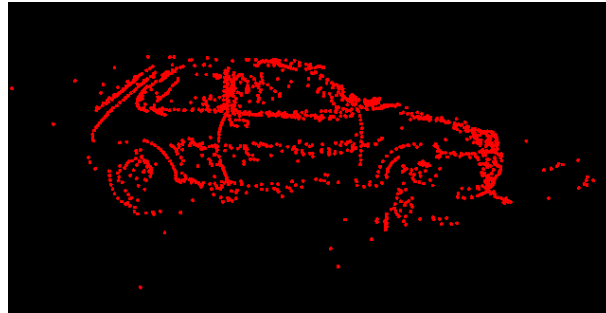


(a)

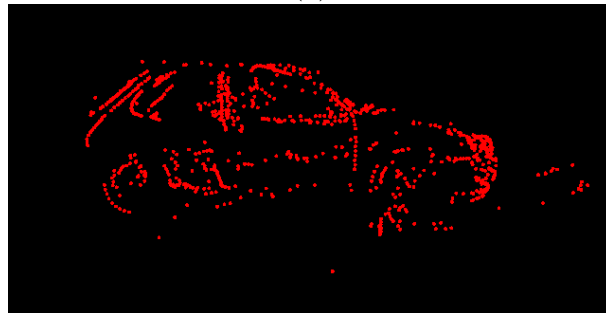


(b)

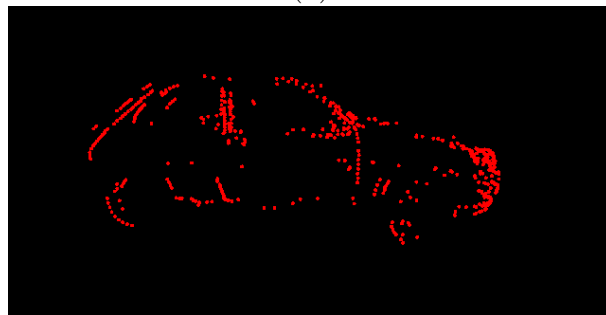
FIGURE 9.3 – (a) Vue du dessus d'un nuage de points 3D et de sa boîte englobante. Les caméras sont sur une ligne au bord de la boîte englobante. (b) Une autre reconstruction. Pour un meilleur affichage, les nuages de points présentés ici ont déjà été filtrés par l'étape présentée dans la partie 9.3.5.



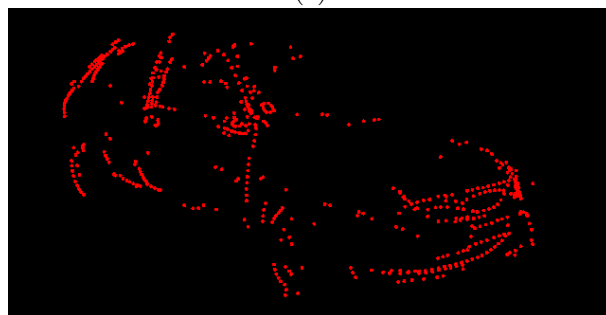
(a)



(b)



(c)



(d)

FIGURE 9.4 – (a) Nuage de points initial. (b) Nuage filtré en imposant une erreur de rétro-projection inférieure à un pixel (c) Nuage filtré en imposant une erreur de rétro-projection inférieure à un pixel et un suivi sur au moins trois images. (d) Vue en perspective d'un nuage de points final.

La figure 9.4(b) montre le résultat avec un seuil τ valant 1 pixel. Sur nos tests, 1 pixel est la valeur minimale acceptable. Si l'on utilise une valeur inférieure, trop de points corrects sont éliminés. Dans la littérature, on considère cependant que ce seuil est plutôt de 0.3 pixel. La raison pour laquelle notre algorithme n'atteint pas cette précision est que nous n'utilisons pas d'étape de minimisation globale des positions des points 3D et des caméras (étape dite de bundle adjustment). Nous avons testé l'ajout de cette étape en utilisant l'implémentation de [LA04]. Nous obtenons alors une erreur de rétro-projection moyenne de 0.1 pixel. Il est alors possible de descendre largement le seuil de filtrage, tout en conservant une majorité des inliers. Toutefois, cette étape est coûteuse en temps de calcul et nous jugeons que pour le problème de la reconstruction de voiture, l'apport en précision ne justifie pas cet allongement du temps de calcul. On choisit donc de ne pas utiliser cette étape dans la suite des résultats présentés.

En filtrant avec l'erreur de rétro-projection, certains outliers restent présents dans le nuage obtenu. La figure 9.4(b) contient toujours des outliers qui sont très éloignés de la surface de la voiture. En analysant l'origine de ces points 3D, on trouve plusieurs sources d'erreurs. Certains points ne sont suivis que sur deux vues consécutives, et sont reconstruits en 3D. Leur erreur de rétro-projection est quasi-nulle. D'autres points sont générés par l'ombre de la voiture au sol. Sur quelques vues consécutives, ces points ont une trajectoire très proche de celle des points appartenant à la voiture. Il est alors très difficile de les détecter comme n'appartenant pas à ce véhicule. Pour filtrer ces points, notre heuristique consiste à rejeter les points qui ont été obtenus par la triangulation de moins de n vues. La figure 9.4(b) montre les résultats obtenus pour $n=3$. En pratique, utiliser $n=4$ ou $n=5$ assure une meilleure robustesse.

9.4 Reconstruction d'une surface à partir du nuage de points 3D

A partir des étapes précédentes, nous obtenons un nuage de points 3D. Toutefois, cela n'est qu'une étape intermédiaire. L'objectif de mesurer des caractéristiques de la voiture requiert une représentation plus évoluée. Nous proposons dans cette partie d'approcher le nuage de points 3D obtenu par une surface paramétrique. Cette représentation sera ensuite très souple à utiliser pour diverses interprétations.

D'autre part, le nuage de points 3D obtenu est complexe. Certains points appartiennent bien à la surface extérieure de la voiture. Ce sont les points les plus intéressants. Mais à cause des surfaces transparentes, certains points à l'intérieur de la voiture sont aussi reconstruits. Certains points des roues, suivis selon la direction du point de fuite, parviennent à passer les filtres précédents et sont donc reconstruits à une profondeur incorrecte. De plus, à cause de la courbure de la surface, certains reflets (e.g. environnement extérieur réfléchi sur les portes de la voiture filmée) ne peuvent pas être éliminés par un simple critère géométrique (voir [SKS⁺02]). Il nous paraît alors nécessaire d'approcher le nuage de points 3D obtenu, de manière robuste, pour obtenir une surface qui

représente uniquement la surface extérieure de la voiture.

Dans notre état de l’art, nous avons présenté plusieurs approches pour approcher un nuage de points 3D avec une surface. Toutefois, les nuages que nous obtenons sont particuliers à traiter, notamment car certains points (les points à l’intérieur) ne font pas partie de l’objet à reconstruire. On cherche plutôt une méthode pour approcher l’enveloppe externe de ces points.

Pour cela, il nous paraît nécessaire d’utiliser des connaissances sur l’objet reconstruit. Nous proposons de travailler dans un repère local à la voiture (largeur, hauteur et profondeur). Cela nous permettra ensuite d’approcher indépendamment, l’avant et le côté de la voiture. Sans ce repère local, les mots “avant” et “côté” du nuage de points n’ont aucun sens. La solution que nous avons adoptée consiste à modéliser les surfaces du côté de la voiture par extrusion d’un polynôme de degré 2 et de l’avant de la voiture par extrusion d’un polynôme de plus haut degré.

9.4.1 Calcul du repère local

Comme les positions des caméras sont duales des positions successives de l’objet, le vecteur entre les caméras successives donne la direction du mouvement de la voiture. Cette direction du mouvement correspond à la direction de la profondeur de la voiture (i.e. avec l’hypothèse de voiture en translation pure). Il reste donc à estimer soit la direction de la largeur, soit celle de la hauteur de la voiture pour obtenir son repère local.

Nous proposons d’estimer la direction selon la largeur de la voiture. Notons que dans nos tests, nous avons développé une méthode qui estime cette direction pour chaque voiture. Toutefois, si l’on considère que la caméra ne bouge pas entre les séquences de voiture, il est possible d’utiliser plusieurs voitures pour estimer cette direction. Le résultat n’en serait que plus robuste. L’avantage de notre méthode est qu’elle n’utilise aucune infrastructure extérieure. On pourrait par exemple imaginer une méthode similaire basée sur des peintures sur la route. On développerait alors une méthode pour calculer la direction de la largeur à partir de la projection de ces peintures dans le plan image de la caméra.

Afin de calculer cette direction de la largeur, nous émettons l’hypothèse que parmi les lignes qu’on peut extraire de l’image d’un véhicule vu de face, la plupart sont alignées selon cette direction. En pratique, cette hypothèse se confirme sur tous les véhicules que nous avons testés. On trouve des lignes droites dans la direction de la largeur par exemple en bas et en haut du pare-brise, sur le pare-choc, la calandre, la plaque d’immatriculation. On pourrait d’ailleurs incorporer à notre système un détecteur de plaques d’immatriculation.

Pour générer une vue de face de la voiture, nous utilisons une approximation grossière du nuage de points par une surface triangulée que nous texturons. Dans cette image virtuelle, nous extrayons les lignes et considérons que la direction majoritaire est la direction de la largeur.

Pour obtenir l’image virtuelle de face, on considère une caméra positionnée devant le nuage de points et qui lui fait face. On n’utilise pas une caméra projective mais une caméra orthographique afin d’assurer que les droites 3D alignées selon la largeur soient parallèles dans le plan image (pour une caméra

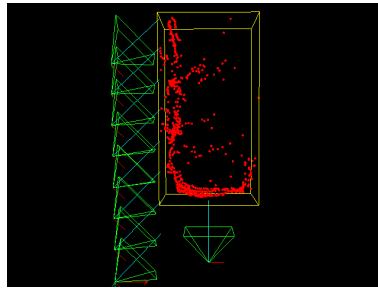
projective mal positionnée, les images de ces droites seraient non parallèles, elles s'intersecteraient en un point de fuite). Et l'axe optique de cette caméra pointe vers la voiture (voir figure 9.5). Pour faire un rendu du véhicule depuis cette caméra, il est nécessaire de texturer cet objet, et il faut alors avoir une représentation sous forme de surface du nuage de points 3D. Dans l'état de l'art, nous avons présenté plusieurs méthodes pour approcher un nuage de points avec une surface. Pour les voitures, nous considérons la surface 3D qui est une surface d'élevation dans le plan focal de la première caméra utilisée dans la reconstruction. Pour obtenir la surface, nous utilisons l'algorithme Multi-Level B-Spline approximation de [LWS97]. Cet algorithme a l'avantage d'être rapide, simple à mettre en oeuvre et donne des résultats tout à fait corrects pour faire un rendu. La figure 9.5 montre les arêtes d'un exemple de maillage ainsi obtenu. Une fois cette surface obtenue, elle est texturée par l'une des images de la séquence vidéo. La figure 9.5.(c) montre un exemple de surface spline ainsi texturée. Pour obtenir une surface correctement texturée, on utilise comme texture l'image de la caméra qui est la plus face au véhicule (e.g. la première image de la séquence vidéo). Sur cette figure, on voit que le résultat n'est pas parfait, mais est suffisant pour rechercher des droites.

La surface 3D ainsi texturée est ensuite projetée dans la caméra de face. Un algorithme de Canny est utilisé pour détecter les contours. Ensuite, une transformation de Hough [DH72] permet de détecter les droites. En discrétisant les directions possibles, nous cherchons la direction qui porte le maximum de ces droites. Par hypothèse, il s'agit de la direction de la largeur. La figure 9.5.d montre une image virtuelle de face ainsi que les droites détectées et la direction principale.

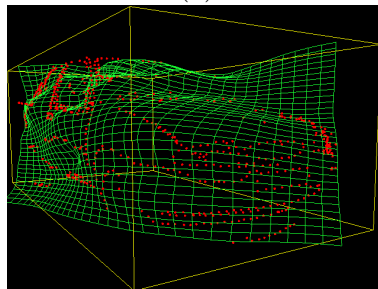
9.4.2 Approximation par un modèle polynomial

Grâce au repère local, on peut aligner la boîte englobante sur les directions naturelles de la voiture (largeur, hauteur et profondeur). Ainsi, les faces de cette boîte englobante sont représentatives (e.g. on peut parler du côté, de l'avant, de l'arrière de la voiture). On va alors projeter le nuage de points sur les faces de cette boîte englobante et en approcher les contours par des polynômes. Cela revient à approcher les profils de la voiture. On présente ici une méthode robuste pour faire cela en considérant uniquement les profils avant et du côté de la voiture.

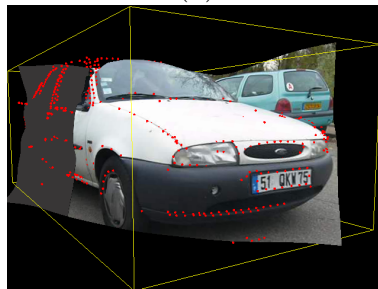
Dans un premier temps, on projette les points 3D sur la face avant de la boîte englobante. On peut alors approcher le côté de ce nuage de points 3D par un polynôme du second degré. Cela revient à estimer le profil du côté de la voiture (en faisant l'hypothèse que la surface du côté de la voiture est une extrusion de ce profil). Le problème revient à approcher des points (x, y) par une fonction polynomiale $y = f(x)$. C'est un problème simple à résoudre par moindres carrés. Toutefois, on doit utiliser une méthode robuste à cause de la présence possible d'outliers. On utilisera alors un algorithme de type moindres carrés itératif avec repondération (dit IRLS pour Iterative Re-weighted Least Square algorithm, voir [Bjo96]). La pondération se fait avec des M-estimators (e.g. Tukey, Huber, voir appendix 6 de [HZ04]). Le profil ainsi obtenu est extrudé



(a)



(b)



(c)



(d)

FIGURE 9.5 – (a) Illustration de la caméra virtuelle positionnée devant le véhicule. (b) La surface spline approchant le nuage de points. (c) Point de vue identique mais la surface spline a été texturée par une des images de la séquence vidéo. Les parties en gris sont des zones de la surface non atteintes par la projection de l'image. (d) La surface spline texturée de (c) rendue à partir de la caméra virtuelle. Les lignes détectées sont en vert. La direction principale est représentée par une droite en rouge.

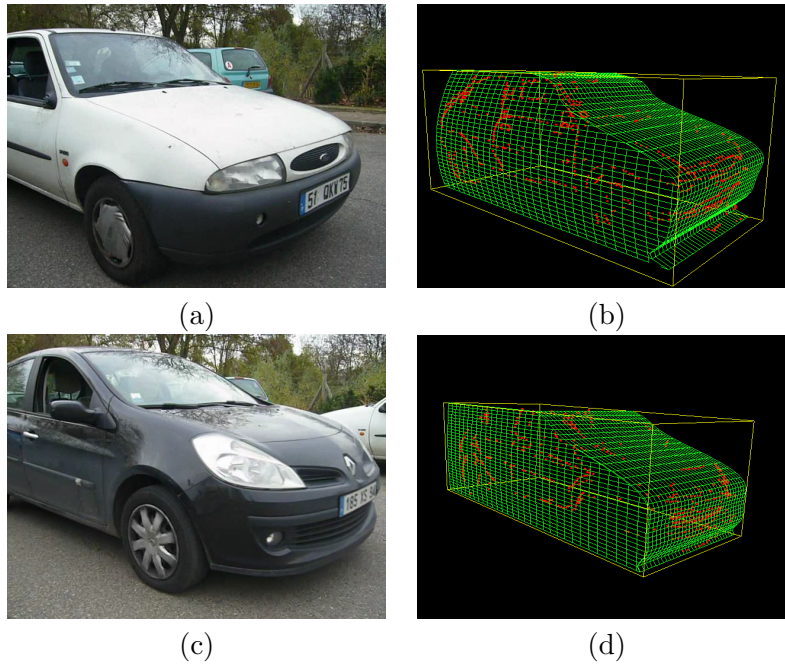


FIGURE 9.6 – Une image extraite d’une séquence vidéo (a) et la reconstruction 3D obtenue par extrusion de polynômes(b). Une image d’une autre séquence (c) et la reconstruction associée (d).

dans la direction de la profondeur pour obtenir la surface complète du côté du véhicule.

Une procédure similaire est appliquée pour la surface de l’avant de la voiture. Les points 3D du nuage sont projetés sur la face du côté de la boîte englobante. Les points sont ensuite approchés par un polynôme de degré 5. Le profil obtenu est alors extrudé selon la direction de la largeur. L’étape finale consiste à fusionner les surfaces du côté et de l’avant du véhicule (voir figure 9.6).

9.5 Evaluation

L’un des problèmes de la reconstruction 3D est celui de la vérité terrain. La méthode idéale consiste à posséder une représentation 3D exacte (ou très fidèle) de l’objet, et ensuite de calculer un score entre une reconstruction obtenue et cette vérité terrain. Dans le chapitre suivant, on verra qu’il existe une base publique où des objets 3D ont été acquis par laser et un algorithme de comparaison de surface triangulée est utilisé pour mesurer l’erreur de reconstruction. Dans le cadre de la reconstruction 3D de voiture, on est face à un problème pour lequel il n’existe pas de base publique. Concernant l’obtention de représentations 3D de voitures, il ne semble pas possible de les obtenir par simple demande aux constructeurs automobile. On ne dispose pas non plus de système d’acquisition laser qui aurait permis d’obtenir un nuage de points

représentant une voiture. Une dernière solution est d'utiliser une voiture virtuelle modélisée avec un modèleur 3D (e.g. Blender, 3DS Max...) et de générer des séquences vidéo virtuelles. Toutefois, comme la difficulté de notre problème vient principalement des conditions réelles, cette approche ne nous a pas paru très pertinente.

Nous nous sommes limité à tester notre méthode sur quelques vidéos. Pour chacune d'elles, notre algorithme permet d'obtenir une reconstruction similaire à celles de la figure 9.6.

9.6 Perspectives

Dans ce chapitre, nous avons proposé une méthode pour utiliser des algorithmes classiques de reconstruction 3D pour des voitures. Notre méthode aboutit à un résultat qui peut être suffisant pour estimer le gabarit d'un véhicule par exemple. Mais si l'on souhaite utiliser la reconstruction pour déterminer le modèle du véhicule, le résultat n'est pas assez précis. Dans cette optique, nous avons donc commencé à étudier des pistes pour aboutir à un résultat plus précis. Ces pistes sont présentées ici, avec des premiers exemples de résultats.

9.6.1 Utilisation de la symétrie

Détection de la symétrie

Quelle que soit la méthode choisie, il paraît intéressant d'utiliser la symétrie de la voiture (selon un plan vertical qui coupe la voiture en partie gauche et partie droite). S'en passer serait se priver d'un élément très caractéristique du problème. De plus, cela restreint les solutions possibles. On peut par exemple ne reconstruire qu'une moitié de la voiture et obtenir la seconde partie par symétrie. Cela permet aussi de travailler uniquement sur la partie de la voiture proche des caméras et donc pour laquelle les images sont de meilleure qualité. Par exemple, la figure 9.7 montre l'image de face de la surface spline 3D approchant la voiture, que l'on a texturée. Il apparaît que la partie gauche est précise alors que la partie droite est floutée (c'est particulièrement visible au niveau du contour vertical du phare). Ceci s'explique par la projection de la texture, qui étire l'image sur la partie droite, plus éloignée de la caméra. Par exemple, les contours du phare sont précis à gauche, et pas très distincts à droite. Cela est vrai, même si l'on a pris soin de prendre une des premières images du film (presque face à la voiture) pour limiter cet effet de projection.

Pour détecter les symétries, on peut utiliser des techniques basées sur des descripteurs locaux ([LE06]), ou utiliser une approche plus globale. Nous avons testé, sans succès, l'utilisation des descripteurs de type SIFT (aussi bien sur des images de la séquence vidéo que sur des images virtuelles de face telles que celle de la figure 9.7). Nous avons aussi testé une approche globale que nous présentons ici. Cette approche permet, d'une manière plus générique, de détecter un plan de symétrie dans un objet 3D. Elle se base sur une minimisation d'une erreur de rétro-projection en s'inspirant de la formulation de Pons et al. [PKF07].



FIGURE 9.7 – Surface spline texturée. Les points affichés sont les points 3D approchés par la surface spline.

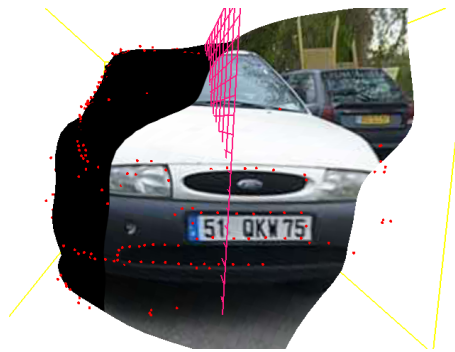


FIGURE 9.8 – Plan de symétrie au milieu de la boîte englobante. On voit que l'erreur est non négligeable. En effet, la boîte englobante est très peu précise.

Admettons pour l'instant qu'on ait obtenu un modèle 3D de la voiture qui soit exact sur la partie gauche de l'image (proche des caméras). On initialise le plan de symétrie (par exemple, milieu de la boîte englobante (Figure 9.8)). On crée la seconde moitié de l'objet (position + texture) par symétrie. Si le plan de symétrie est correct, comme la forme est correcte, l'erreur de rétro-projection est faible. Si le plan est mal positionné, cette erreur est grande, on va la faire diminuer en bougeant le plan de symétrie.

Le plan de symétrie est initialisé par le plan milieu de la boîte englobante. Cela donne le maillage de la figure 9.9. Ensuite, la texture peut être appliquée selon deux possibilités :

- Projection d'une image du film sur tout le maillage. Le problème est que le côté opposé est souvent mal texturé. En effet la normale en un point de la surface est quasi-perpendiculaire avec l'axe reliant ce point au centre optique de la caméra. Cela a pour effet qu'une petite erreur de surface peut causer une grande erreur de texturage.

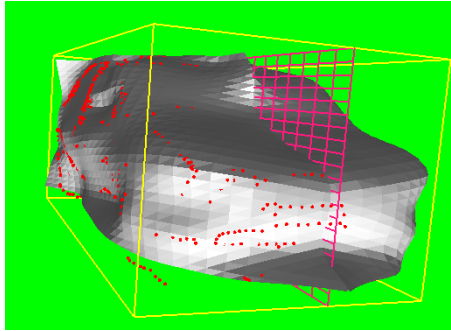


FIGURE 9.9 – Le maillage à gauche est symétrisé.



FIGURE 9.10 – (a) Le maillage à droite utilise une texture projective classique. (b) Le maillage à droite utilise les mêmes coordonnées de texture que la partie gauche, par symétrie.

- Projection d'une image du film sur le maillage gauche, classiquement. Pour le maillage droit, on symétrise le maillage gauche avant de projeter la texture. Au final, cela donne un maillage symétrique, géométriquement, et aussi en texture.

Les figures 9.10.(a) et 9.10.(b) montrent les deux cas. On voit qu'il est très difficile de travailler sur la première image car la texture est de trop mauvaise qualité sur la partie droite. A l'inverse, le second modèle donne une image plus proche de la réalité. On choisira donc plutôt cette seconde approche pour texturer le maillage.

Dans les paragraphes suivants, nous définissons l'erreur que nous cherchons à minimiser en déplaçant le plan de symétrie. Nous exprimons le gradient de cette erreur pour ensuite faire une minimisation par descente de gradient.

Erreur définie et minimisation

L'erreur globale que nous étudions est :

$$E(s) = \sum_{c=0}^n E_i(s) \quad (9.6)$$

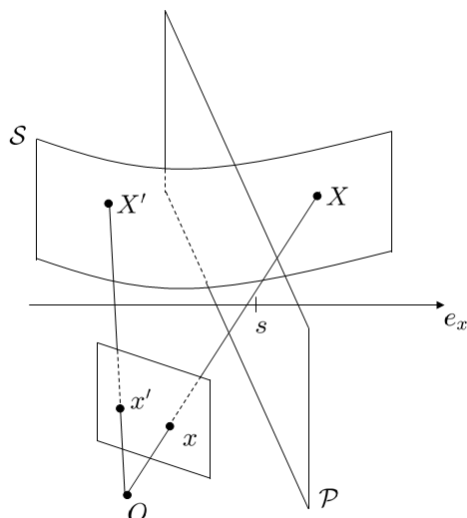


FIGURE 9.11 – Notations pour l'étude de la symétrie.

où s correspond à la position du plan de symétrie sur l'axe de la largeur de la voiture. Nous cherchons à déterminer si la position correcte du plan de symétrie peut être vue comme minimum de cette erreur.

Pour calculer la fonction erreur $E_i(s)$, une gaussienne est appliquée à I_i et $\Pi_i(\mathcal{S}_2)$ et les différences des intensités au carré entre ces deux images sont sommées (en ne prenant en compte que les pixels atteints par $\Pi_i(\mathcal{S}_2)$). Cette erreur est calculée pour plusieurs positions du plan de symétrie. A noter que pour que les erreurs soient indépendantes du nombre de pixels atteints, il faut normaliser $E_i(s)$ par le nombre de pixels sur lesquels est calculée la différence d'intensité.

En pratique, nous avons pu vérifier que le minimum de la fonction d'erreur correspond à la position sur la figure 9.10. Les figures 9.13.(a) et 9.13.(b) montrent les images correspondant à deux autres positions (à gauche et à droite du minimum).

Nous montrons maintenant comment exprimer la dérivée de cette erreur.

Soit \mathcal{P} le plan de symétrie de l'objet. On suppose que la normale à ce plan est l'axe e_x . Soit \mathcal{S} la surface de l'objet (i.e. la surface que l'on cherche). La partie de \mathcal{S} à gauche de \mathcal{P} est notée \mathcal{S}_1 , respectivement \mathcal{S}_2 à droite (les caméras sont à gauche du véhicule).

Soit $S_{\mathcal{P}}$ la symétrie de plan \mathcal{P} , positionné à l'abscisse s , de matrice :

$$S_{\mathcal{P}} = \begin{bmatrix} -1 & 0 & 0 & 2s \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.7)$$

Un algorithme de reconstruction nous a fourni \mathcal{S}_1 qui est supposée correcte.

On construit la partie \mathcal{S}_2 par symétrie (forme et texture) :

$$\mathcal{S}_2 = S_P(\mathcal{S}_1) \quad (9.8)$$

Notons Π_i la projection dans le plan image de la caméra i . L'inverse notée Π_i^{-1} , rétro-projette un point du plan focal x sur la surface. Sur la figure 9.11, on a :

$$x = \Pi_i(X) \quad \text{et} \quad X' = S_P(X) \quad (9.9)$$

et

$$X = \Pi_i^{-1}(x) \quad (9.10)$$

La surface \mathcal{S}_1 est texturée par projection de l'image i . Si le plan de symétrie est bien placé, la surface \mathcal{S}_2 rétro-projetée est proche de l'image i . On note cette erreur :

$$E_i(s) = E(I_i, Im) \quad (9.11)$$

où I_i est l'image i , et Im est : $\Pi_i(\mathcal{S}_1 \cup \mathcal{S}_2)$

On va chercher à minimiser cette erreur en translatant le plan \mathcal{P} selon la direction de sa normale (i.e. paramètre s). Quand on bouge \mathcal{P} , la surface \mathcal{S}_2 évolue (et \mathcal{S}_1 aussi).

On remarque que la couleur au point x est obtenue par la fonction composée suivante (qui retourne en fait la couleur au point x') :

$$I \circ \Pi_i \circ S_P \circ \Pi_i^{-1} \quad (9.12)$$

L'erreur E_i entre la vérité terrain I_i et l'image obtenue Im est donc du type :

$$E(I_i, Im) = E(I_i, I \circ \Pi_i \circ S_P \circ \Pi_i^{-1}) \quad (9.13)$$

On cherche à exprimer :

$$\frac{\partial E_i}{\partial \epsilon}(s + \epsilon \delta s)|_{\epsilon=0} = \int \partial_2 Err(x) \cdot DI_i(x') \cdot D\Pi_i(X') \cdot DS_P(X) \cdot \frac{\partial \Pi_i^{-1}}{\partial \epsilon}(x)|_{\epsilon=0} dx \quad (9.14)$$

où D est la matrice jacobienne d'une fonction et Err est la fonction utilisée pour calculer l'erreur en un point. C'est-à-dire que nous avons :

$$E(I_i, Im) = \int Err(x) dx \quad (9.15)$$

et dans nos tests, nous avons pris pour Err une différence d'intensité au carré :

$$E(I_i, Im) = \int (I_i(x) - Im(x))^2 dx \quad (9.16)$$

mais une autre mesure d'erreur serait tout à fait envisageable.

Il est possible d'exprimer la relation entre le mouvement de la surface en un point X de \mathcal{S}_2 qui correspond au point x :

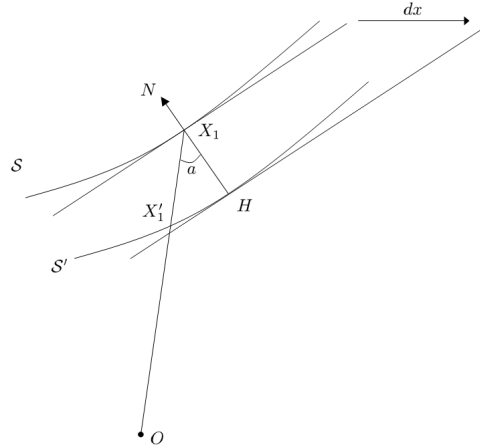


FIGURE 9.12 – Notations pour l'étude d'un déplacement de la surface.

$$\frac{\partial \Pi_i^{-1}(x)|_{\epsilon=0}}{\partial \epsilon} = \frac{2\delta s N^T \cdot e_x}{N^T \cdot d} d \quad (9.17)$$

où d est le vecteur entre le centre optique et le point X et N est la normale unitaire sortante à la surface.

Preuve de 9.17

Les notations sont celles de la figure 9.12. La surface \mathcal{S} est déplacée de dx et devient \mathcal{S}' . On avait $X_1 = \Pi_i^{-1}(x)$, c'est-à-dire que le rayon optique intersectait la surface au point X_1 . Après ce déplacement de surface, quel est le nouveau point d'intersection entre le rayon optique et la nouvelle surface ? La surface est approchée localement par un plan (de normale, la normale à la surface N). La distance entre les plans approchant \mathcal{S} et \mathcal{S}' est :

$$X_1 H = (dx \cdot N) N \quad (9.18)$$

Le cosinus de l'angle a s'exprime de deux façons :

$$\cos(a) = \frac{(N^T dx) N}{X_1 X_1'} = \frac{N^T d}{\|d\|} \quad (9.19)$$

d'où :

$$X_1 X_1' = \frac{N^T dx}{N^T \cdot d} d \quad (9.20)$$

Le plan de symétrie a été translaté de δs donc la surface en X_1 se déplace de $2\delta s$.

Expression finale du gradient

Au final, nous avons :

$$\frac{\partial E_i}{\partial \epsilon}(s + \epsilon \delta s)|_{\epsilon=0} = \int \left[\partial_2 Err(x) \cdot DI_i(x') \cdot D\Pi_i(X') \cdot DS_{\mathcal{P}}(X) \cdot \frac{2N^T \cdot e_x}{N^T \cdot d} d \right] \delta s dx \quad (9.21)$$

D'où l'expression de la dérivée de l'erreur pour un élément de surface du plan focal (pixel) :

$$\frac{dE_i}{ds}(x) = \partial_2 Err(x) \cdot DI_i(x') \cdot D\Pi_i(X') \cdot DS_{\mathcal{P}}(X) \cdot \frac{2N^T \cdot e_x}{N^T \cdot d} d \quad (9.22)$$

On vérifie qu'il s'agit bien d'un scalaire (d est un vecteur 3D, $\frac{2N^T \cdot e_x}{N^T \cdot d}$ est un scalaire, $DS_{\mathcal{P}}(X)$ est une matrice 3×3 , $D\Pi_i(X')$ est une matrice 2×3 , $DI_i(x')$ est de dimension 1×2 et $\partial_2 Err(x)$ est un réel). Il est alors possible de faire une descente de gradient, en bougeant la position s du plan de symétrie. En pratique, la dérivée est calculée uniquement pour les pixels qui sont des projections des sommets du maillage, et ces composantes sont sommées. On ne prend en compte que les sommets dont la normale a un produit scalaire négatif avec d . Cela permet de ne pas prendre en compte les sommets de la surface là où celle-ci ne fait pas face à la caméra.

La matrice jacobienne de la symétrie par rapport au plan de normale e_x est :

$$DS_{\mathcal{P}}(X) = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (9.23)$$

A noter qu'on pourrait très bien utiliser le même processus pour affiner la surface S_1 en utilisant la symétrie (en utilisant une seule vue). Mais dans cette partie, nous présentons uniquement une méthode pour obtenir le plan de symétrie.

9.6.2 Erreur utilisée

Résultats

En pratique, la minimisation présentée ci-dessus est très délicate à utiliser pour des voitures. A cause du manque de gradient dans l'image (des grandes zones sans texture) ainsi que du bruit, notamment dans les phares, la minimisation ne fonctionne pas très bien. Au final, même si l'approche théorique paraît intéressante, il est plus simple de mesurer l'erreur pour certaines positions du plan de symétrie et de choisir la meilleure position (calcul direct de E au lieu de faire une descente de gradient en calculant dE/ds). Avec cette approche, le plan de symétrie est grossièrement correct. En cherchant à affiner sa position en utilisant la dérivée de l'erreur, à cause du bruit, et aussi du fait que la surface correcte (proche de la caméra) peut être imparfaite, nous n'améliorons pas le résultat.



FIGURE 9.13 – Exemples de voiture symétrisée avec (a) un plan trop à gauche et (b) un plan trop à droite.

9.6.3 Modélisation par une surface spline

Dans cette partie, nous présentons une approche que nous avons envisagée pour améliorer la surface obtenue. L'idée est de modéliser la voiture par une surface spline. Ensuite, cette surface spline serait déformée pour minimiser une erreur basée sur des mesures de photo-cohérence. Nous présentons ces différentes étapes dans les paragraphes suivants.

Pour améliorer la reconstruction, après avoir obtenu le plan de symétrie de la voiture, nous cherchons à déformer la surface pour la faire se rapprocher au mieux de la vraie surface de la voiture. A cause des fortes difficultés de la reconstruction (zones transparentes, reflets...), il est nécessaire d'utiliser une représentation où les contraintes seront fortes pour éviter que la surface obtenue soit complètement fautive. Pour cela, une option intéressante est de travailler sur une surface spline. L'un des avantages de ces surfaces est qu'il est suffisant de déplacer les points de contrôle et par ailleurs la surface finale est lisse.

La première étape pour cette approche, est de préciser le domaine de définition de la surface spline.

Paramétrisation de la surface spline

Pour travailler avec une surface spline, il faut définir un morphing entre la surface triangulée et un rectangle (traditionnellement, le domaine $[0, 1] \times [0, 1]$). C'est-à-dire, pour chaque sommet de la surface triangulée, on cherche à obtenir ses coordonnées u, v dans le domaine de définition de la surface spline. Pour cela, on peut utiliser le "barycentric mapping". Une bonne introduction est donnée dans Hormann et al. [Hor07]. Cela revient à définir les arêtes de la surface triangulée comme des ressorts et à minimiser l'énergie du système une fois ramené dans le plan. Le choix de la constante de ressort le plus simple est 1, choix fait dans Tutte et al. [Tut63]. On trouve d'autres approches dans [Flo03, EDD+95, Wac78, DMA02].

Il est aussi nécessaire de fixer les coordonnées de la paramétrisation spline des sommets appartenant aux bords du maillage. Dans notre cas, nous attribuons les coordonnées $(0, 0)$ $(0, 1)$ $(1, 0)$ $(1, 1)$ aux sommets du maillage les plus proches des coins du bas de la boîte englobante de la voiture (notés A,B,C et D

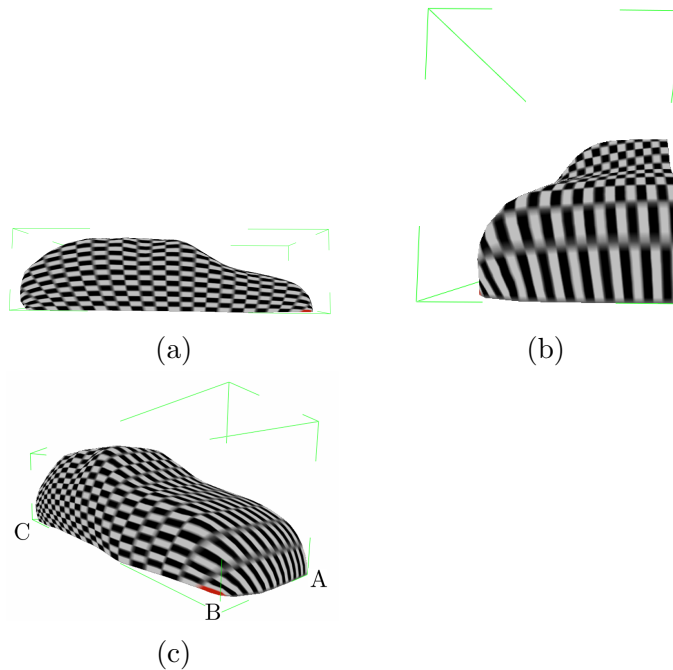


FIGURE 9.14 – Exemple d’une paramétrisation de la surface d’une demi-voiture. (a) : vue de côté de la demi-voiture. (b) vue de face de la demi-voiture. Pour obtenir la seconde moitié de la voiture, il faut appliquer une symétrie de la première moitié. (c) Extrémités du rectangle de contrôle de la surface spline. Sur cette figure, le point D est caché par la surface.

sur la figure 9.14.c). Pour les sommets des bords du maillage autres que ces 4 points, les coordonnées u et v sont interpolées. Ensuite, pour trouver les coordonnées de tous les autres sommets du maillage, on applique l’une des méthodes présentées ci-dessus. La figure 9.14 montre un exemple de résultat.

Ayant choisi une méthode pour paramétrer la surface spline, il est nécessaire de présenter comment déformer cette surface.

Comment déformer la surface spline

Notre approche suppose qu’en utilisant un critère de photo-cohérence (e.g. une erreur du même type de 9.16), nous serons capable de définir un nouveau nuage de points que la surface devrait approcher. L’idée est d’itérer ce processus. La surface initiale approche un nuage de points grossier et permet de définir une surface spline (tel qu’expliqué au paragraphe précédent). En utilisant cette surface pour effectuer des rendus, un nouveau nuage de points est calculé (pour certains points de cette surface, il faut estimer une position plus photo-cohérente). Ensuite, ce nouveau nuage de points est approché avec une nouvelle surface spline, et ainsi de suite jusqu’à convergence. L’étape d’approcher le nouveau nuage de points est un problème d’approximation de nuage de points en 2D et demi tel que présenté dans la bibliographie en 8.3.1.

Dans le chapitre suivant, une approche est présentée pour exprimer comment générer les nouvelles positions de points en utilisant une surface existante. Cette approche y est utilisée pour déformer un maillage mais elle est adaptable pour déformer une surface spline.

9.7 Conclusion

Dans ce chapitre, notre objectif était d'étudier la reconstruction 3D de voitures à partir de séquences vidéos. Rappelons que ce sujet était exploratoire, l'idée était de savoir quels résultats l'on pouvait obtenir, dans le but d'applications industrielles futures. C'est-à-dire qu'il n'y avait pas de critère de qualité défini sur la reconstruction 3D souhaitée.

Nous avons donc dans un premier temps cherché une méthode robuste pour reconstruire une surface grossière du véhicule. L'approche que nous avons proposée, basée sur des méthodes classiques de reconstruction 3D, permet d'aboutir à une représentation simple d'un véhicule. La représentation sous forme d'extrusion de polynômes selon les directions principales de la voiture (longueur et largeur) permet d'être robuste aux points 3D erronés.

Dans une seconde partie, nous avons présenté des pistes pour tout d'abord obtenir le plan de symétrie d'un objet 3D. Cette approche générique, quoique intéressante en théorie, n'est pas praticable sur les voitures. Toutefois, il aurait été très intéressant de la tester sur d'autres objets, par exemple des visages. En alternant les minimisations (entre la surface à reconstruire et la position du plan de symétrie), il est possible d'imaginer une méthode de reconstruction 3D basée sur une erreur image pour reconstruire un objet avec une seule image si celui-ci est symétrique.

Enfin, nous avons proposé une seconde piste d'étude pour modéliser une demi-voiture par une surface spline (et ainsi d'utiliser la symétrie du véhicule). L'intérêt était de déformer ensuite cette surface tout en gardant un aspect correct. Cette partie est liée au chapitre suivant dans lequel nous proposons une méthode itérative pour faire converger une surface triangulée vers la surface d'un objet 3D. La motivation initiale de ce nouveau chapitre était d'appliquer cette déformation de surface sur les surfaces grossières des voitures, en remplaçant la surface triangulée par une surface spline.

Chapitre 10

Reconstruction 3D par déformation de surface guidée par des correspondances SIFT

Nous présentons ici un algorithme original, basé sur une surface déformable, pour reconstruire en 3D un objet (pas forcément une voiture) à partir de plusieurs images. Notre objectif est de proposer un algorithme de reconstruction 3D dans lequel il est simple d'introduire des correspondances entre éléments des images 2D (e.g., faire correspondre le contour d'une vitre de voiture). Dans ces travaux, les calibrations internes et externes des caméras sont connues et l'on souhaite reconstruire de manière précise l'objet présent sur les images. L'algorithme introduit utilise les descripteurs SIFT dans une première passe pour améliorer une surface initiale. Dans une seconde passe, un critère plus dense de photo-cohérence est rajouté pour retrouver des détails plus fins. L'originalité de cet algorithme vient du fait que les correspondances sont cherchées entre images de même point de vue, ce qui n'est possible que grâce à l'utilisation de descripteurs puissants, comme les SIFT. Notre premier apport est d'intégrer ces correspondances SIFT obtenues à partir d'un même point de vue dans une approche par surface déformable. Notre seconde contribution est de proposer une méthode pour minimiser une erreur de photo-cohérence locale, de manière robuste, toujours entre deux images ayant le même point de vue (Auclair et al. [ACV08]).

Sommaire

10.1 Notations	205
10.2 Mesure de photo-cohérence utilisée	206
10.3 Modèle déformable utilisé	208
10.4 Force calculée à partir de correspondances SIFT	210
10.5 Ajout d'un terme plus local	212
10.6 Détails concernant l'implémentation	213
10.7 Résultats	215
10.8 Analyse qualitative	215
10.9 Conclusion	218



FIGURE 10.1 – Exemple d’images qui sont la donnée d’entrée de l’algorithme. En plus de ces images, nous disposons des calibrations internes et externes des caméras.

Dans les chapitres précédents, nous nous sommes concentré sur comment aboutir à une reconstruction 3D d’une voiture. L’objectif était d’obtenir une méthode robuste aux difficultés que représentent les surfaces de voitures. Même si dans ce chapitre, nous présentons une méthode qui n’est pas liée à un type d’objet particulier, une motivation reste de présenter une méthode de reconstruction 3D que nous pourrions ensuite appliquer aux voitures. Pour affiner les surfaces 3D de voitures obtenues dans le chapitre précédent, il est intéressant d’envisager d’utiliser des structures spécifiques à l’objet reconstruit. On cherchera par exemple à détecter le pare-brise, les vitres, les phares ou les roues et à les utiliser pour améliorer la surface 3D. Cet élément est important car il justifie certains choix de ce chapitre.

Dans cette partie nous nous concentrons donc sur une méthode pour améliorer une surface 3D grossière. L’objet à reconstruire peut être quelconque (i.e. pas forcément une voiture). Un avantage de développer une méthode générique est de pouvoir l’appliquer sur les jeux de tests publics de Seitz et al. [SCD⁺06]. D’autre part, dans la première partie de cette thèse, nous avons beaucoup travaillé sur les descripteurs SIFT. Nos résultats confirment que ces descripteurs sont très performants. Aussi nous proposons ici une méthode qui utilise ces descripteurs.

La donnée d’entrée du problème est un jeu d’images tel que celui de la figure 10.1. Les calibrations internes et externes sont connues. L’objectif est d’obtenir le plus précisément possible (et le plus rapidement possible) une représentation 3D de l’objet présent sur ces images.

Dans la littérature, une majorité de méthodes utilisent une surface initiale grossière qui est déformée vers un résultat plus précis. Nous considérons ici que nous disposons aussi d’une telle surface. Dans le cas de voitures, on pourra par exemple utiliser la surface obtenue par l’algorithme présenté dans le chapitre précédent. Dans le cas d’autres données, on pourra utiliser l’enveloppe visuelle (voir 8.4.1).

Un de nos objectifs est de développer une méthode que l’on pourra adapter

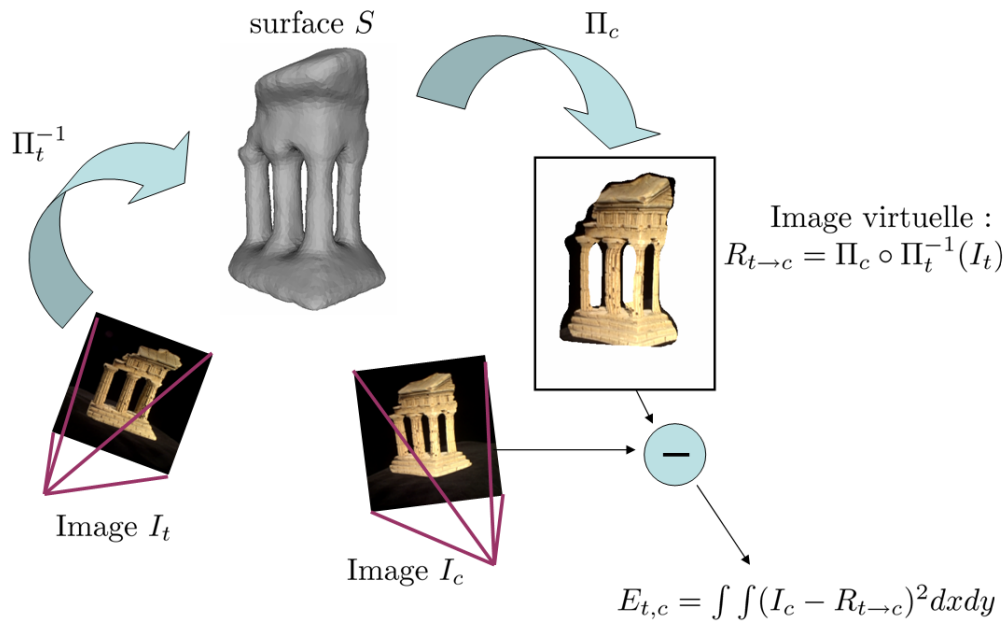


FIGURE 10.2 – Notations et principe. L’indice t souligne que cette image est utilisée comme texture, et l’objet texturé est projeté dans la caméra d’indice c .

au type de surface. Par exemple, si l’on dispose de connaissances a priori sur l’objet et les images (e.g. quels pixels appartiennent à des zones transparentes), on devra pouvoir les incorporer simplement dans l’algorithme. Cette condition nous paraît favoriser une approche par surface déformable. En effet, il n’y a aucune restriction sur les forces que l’on applique aux sommets de la surface triangulée à déformer. De plus, nous considérons que la surface initiale a la bonne topologie. Cela limite l’intérêt d’utiliser une méthode par Level Set. Enfin, un avantage des surfaces déformables sur une approche par Graph-Cut est que l’on peut mieux visualiser les étapes intermédiaires de l’évolution. Cela peut être utile dans le cas où l’on travaille sur des surfaces complexes pour mieux appréhender les défauts de la reconstruction obtenue. L’avantage du Graph-Cut est d’obtenir un minimum global (dans certains cas), mais cela ne permet pas de maîtriser l’évolution de cette minimisation.

10.1 Notations

Soit S la surface déformable que l’on souhaite faire converger vers la surface réelle de l’objet. Les n images en entrée de l’algorithme sont notées $\{I_i\}_{i \in 1..n}$. La matrice de projection de la i^{eme} caméra est notée Π_i . A l’inverse, la fonction qui pour un pixel donné, fournit le plus proche point de la surface S qui se projette sur ce pixel est notée Π_i^{-1} . Pour un pixel p , $\Pi_i^{-1}(p)$ est le point d’intersection entre le rayon optique de la caméra i passant par p et la surface S . S’il y a plusieurs tels points d’intersection, le point le plus proche du centre optique de

la caméra i est choisi. Le résultat de la projection de la surface S , texturée avec l'image I_t , dans la caméra c est noté $R_{t \rightarrow c}$:

$$R_{t \rightarrow c} = \Pi_c \circ \Pi_t^{-1}(I_t) \quad (10.1)$$

Ces notations sont reprises sur la figure 10.2.

10.2 Mesure de photo-cohérence utilisée

Dans la littérature, la mesure de photo-cohérence locale est classiquement calculée entre des images prises selon différentes caméras (i.e. entre I_c et I_t avec $c \neq t$, voir figure 10.3). Toutefois, pour calculer un score tel que la SSD ou la NCC, des différences d'intensité de pixels sont sommées sur une région autour du pixel étudié. Mais il est difficile d'établir la correspondance pixel à pixel au sein de ces deux régions. En général, on travaille sur une fenêtre carrée avec l'hypothèse fronto-parallèle. Celle-ci stipule que la surface de l'objet est parallèle aux plans focaux des caméras. Ainsi on suppose qu'il n'y a pas de déformation entre les fenêtres de pixels et les différences d'intensités sont calculées pixel à pixel sur ces fenêtres carrées dans les deux images. Mais cette hypothèse est peu valide si la surface est inclinée par rapport au plan image ou dans les zones de forte courbure. Pour améliorer un peu ce calcul, certains auteurs font l'hypothèse que la surface de l'objet est localement plane et estiment l'homographie entre les deux fenêtres (i.e. entre les pixels d'une fenêtre sur une image et les pixels de la fenêtre sur la seconde image). Cela renforce le calcul du score de photo-cohérence si l'objet n'est pas parallèle aux plans focaux. Mais cela ne prend toujours pas en compte la courbure de l'objet.

Dans Pons et al. [PKF07], les auteurs définissent une erreur globale qui s'affranchit de ces problèmes. Pour un couple de vues t et c , ils définissent une erreur du type :

$$E_{t,c} = \int \int (I_c - R_{t \rightarrow c})^2 dx dy \quad (10.2)$$

C'est-à-dire que l'erreur est calculée entre l'image I_c et l'image virtuelle $R_{t \rightarrow c}$. Comme ces deux images ont été enregistrées dans le plan image de la même caméra, il n'y a pas de problème de déformation. Ce principe est illustré sur la figure 10.2.

Dans notre algorithme, nous reprenons cette approche en utilisant des descripteurs locaux au lieu d'utiliser une erreur calculée sur les images complètes. L'idée est d'être plus rapide en ne considérant que les points intéressants et d'utiliser des descripteurs locaux plus puissants que seulement les intensités de pixels dans une fenêtre (couplés à une mesure comme SSD ou NCC). C'est-à-dire que pour chaque paire d'images (t, c) , nous calculons des correspondances de descripteurs locaux entre I_c et $R_{t \rightarrow c}$. Ces correspondances sont la mesure de photo-cohérence que nous utilisons pour déformer la surface. La difficulté vient du fait qu'au début du processus, la surface S est très grossière et par conséquent, l'image $R_{t \rightarrow c}$ sera potentiellement très différente de l'image I_c . Toutefois, grâce à la robustesse des SIFT, nous obtenons assez de correspondances

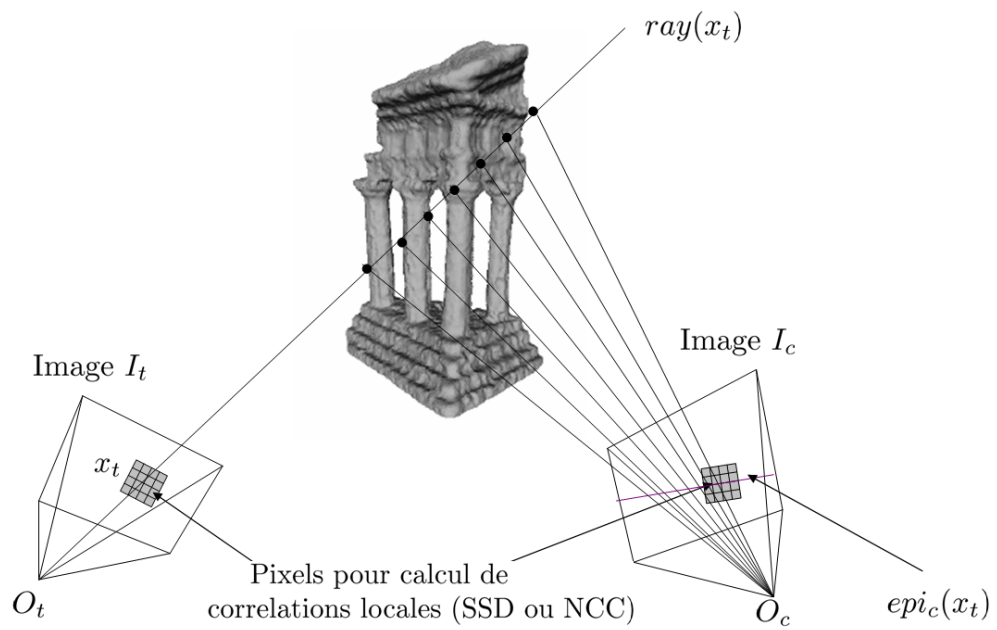


FIGURE 10.3 – Pour un pixel x_t dans l'image I_t , on va calculer des mesures de photo-cohérence avec des positions sur la droite épipolaire correspondante dans l'image I_c . Cela revient à tester si des points 3D sur la demi-droite $[O_t, x_t]$ appartiennent ou non à la surface de l'objet. Pour cela, on calcule un score de type SSD ou NCC entre des patches de pixels.

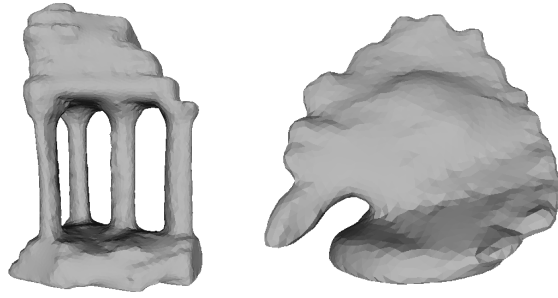


FIGURE 10.4 – Exemples d’enveloppes visuelles.

pour déformer la surface S vers l’objet réel. De plus, pour un point x_c dans l’image I_c , il n’est nécessaire de rechercher sa correspondance que dans un voisinage autour de ce point dans l’image $R_{t \rightarrow c}$. Cela rend encore plus robuste la recherche de correspondances.

Cette déformation de surface est capable de retrouver les concavités de l’objet à partir d’une surface initiale. Toutefois, comme les descripteurs SIFT ne sont pas détectés de manière dense (on utilise le détecteur de points d’intérêt lié aux SIFT), les détails à petite échelle sont difficiles à retrouver uniquement avec cette méthode. Pour obtenir ces détails, nous utilisons une approche voisine de celle utilisée dans Furukawa et al. [FP06]. Toutefois, au lieu de minimiser la NCC comme dans cet article, nous utilisons un algorithme de Lucas-Kanade pour minimiser la SSD. Notre apport est d’utiliser cet algorithme de manière très efficace en limitant l’espace de recherche de correspondances à une seule dimension. De plus, toujours pour éviter les problèmes de distorsion, nous calculons ces correspondances entre une image réelle et une image virtuelle dans la même caméra.

10.3 Modèle déformable utilisé

Dans les paragraphes suivants, nous décrivons les deux étapes nécessaires au modèle déformable : son initialisation et les forces qui lui sont appliquées.

10.3.1 Initialisation

Sans surface initiale a priori, l’enveloppe visuelle de l’objet peut être utilisée. En pratique, nous la calculons de manière simple. On commence par segmenter les n images. Sur les images du jeu de test public de [SCD⁺06], le fond est très sombre donc une simple binarisation aboutit à un résultat correct. Ensuite, l’espace d’étude est divisé en une grille de voxels. Pour chaque voxel, nous comptons le nombre de fois où celui-ci est marqué comme intérieur dans une image. Enfin, après avoir lissé les résultats, un algorithme de marching-cube est appliqué pour obtenir une iso-surface qui a la bonne topologie (deux exemples sont donnés sur la figure 10.4).

10.3.2 Evolution

L'objectif de la déformation est de faire converger la surface vers l'objet réel. Pour cela, nous cherchons à faire évoluer la surface pour qu'elle soit photo-cohérente avec les images, que ses silhouettes (dans toutes les images) correspondent aux silhouettes de l'objet, et enfin que la surface soit lisse. Dans ce but, nous appliquons certaines forces pour déformer la surface (représentée sous forme d'un maillage triangulé). Chaque sommet v de ce maillage est ainsi soumis à trois forces :

$$F(v) = w_c F_c(v) + w_s F_s(v) + w_p F_p(v) \quad (10.3)$$

où F_c est une force de silhouette, F_s est un terme de lissage, F_p est une force de photo-cohérence et les coefficients w_p , w_c et w_s servent à pondérer ces forces.

Force de silhouette

F_c est une force de silhouette, telle que définie dans [NM03]. Elle attire les points détectés comme générateurs de contours vers une position cohérente avec la silhouette de l'objet. Cela nécessite que la silhouette de l'objet ait été calculée sur chaque image I_i . Un sommet du maillage v est dit générateur de contour dans la caméra c si $\Pi_c(v)$ est proche de la silhouette de $\Pi_c(S)$. Pour chaque image d'entrée, on calcule une carte des distances à la silhouette de l'objet. Ces cartes sont calculées off-line par un algorithme de fast-marching (voir [Set99]). En utilisant cette carte, on trouve rapidement le pixel p_v appartenant à la silhouette de l'objet sur la caméra c , et qui est le plus proche de $\Pi_c(v)$. On note X_c le point le plus proche de v sur le rayon optique partant du centre optique O_c et passant par p_v . La force correspondante est $F_c(v) = X_c - v$.

Force de lissage

La force F_s est un terme de lissage classique, calculé comme le Laplacien sur le maillage (comme utilisé dans [ES04]). C'est-à-dire que cette force attire chaque sommet à la position moyenne des sommets qui lui sont voisins.

Force de photo-cohérence

Enfin, la force F_p attire le sommet à une position qui va rendre la surface plus photo-cohérente. Dans notre approche, ce sont les correspondances de descripteurs SIFT qui génèrent cette force. Cette force est introduite en détail dans la section suivante. Le but de cette force est de corriger les erreurs importantes de la surface en cours de déformation (e.g. les grosses concavités qui ne peuvent pas exister dans l'enveloppe visuelle). Enfin, dans la section 4, nous calculons cette force d'une autre façon afin de retrouver les détails plus fins.

10.4 Force calculée à partir de correspondances SIFT

Pour un couple de caméras (c, t) , nous utilisons l'algorithme SIFT pour trouver des correspondances de points entre une image I_c et une image virtuelle $R_{t \rightarrow c}$. Les positions de ces points sont notées respectivement $\{x_c^i\}$ et $\{r_{t \rightarrow c}^j\}$. Lors de la recherche de correspondances, nous ne considérons que les paires potentielles qui respectent la contrainte épipolaire. En général, cette contrainte épipolaire est exprimée pour deux points appartenant à deux images de deux caméras différentes. Dans notre cas, on peut la traduire en disant que $\{x_c^i\}$ et $\{r_{t \rightarrow c}^j\}$ peuvent être une paire valide si $\Pi_t \circ \Pi_c^{-1}(r_{t \rightarrow c}^j)$ appartient à la droite épipolaire de x_c^i dans la caméra t . En pratique, on accepte une paire si le point $\Pi_t \circ \Pi_c^{-1}(r_{t \rightarrow c}^j)$ est proche de la droite épipolaire (voir figure 10.5). Parmi les paires de points qui respectent cette contrainte épipolaire, on choisit celle qui minimise la distance euclidienne entre les descripteurs SIFT (calculée entre leurs vecteurs descripteurs à 128 dimensions). Nous rajoutons aussi la contrainte que la distance entre $\{x_c^i\}$ et $\{r_{t \rightarrow c}^j\}$ doit être inférieure à un seuil. En pratique, pour ce seuil, nous choisissons une valeur élevée, uniquement pour éviter les erreurs importantes. On pourrait toutefois faire évoluer ce seuil au fur et à mesure de la déformation. En effet, initialement, la surface est grossière et on doit donc accepter des paires de points assez éloignées. Mais à la fin de la déformation, la surface est plus précise et l'on pourrait n'accepter que les paires de points qui sont très proches.

Expression de la force à partir des correspondances de points Soit $(x_c, r_{t \rightarrow c})$ une correspondance de points acceptée pour une paire de caméras (c, t) , x_c est un point de l'image I_c et $r_{t \rightarrow c}$ sa position correspondante dans l'image virtuelle $R_{t \rightarrow c}$. En faisant l'hypothèse que cette correspondance de points est valide, si la surface S était exactement la surface de l'objet, le vecteur $x_c - r_{t \rightarrow c}$ serait nul. On va donc chercher la modification de la surface qui annule ce vecteur.

Soit X_c l'antécédent de x_c sur la surface : $X_c = \Pi_c^{-1}(x_c)$. X_t est l'antécédent de $r_{t \rightarrow c}$: $X_t = \Pi_c^{-1}(r_{t \rightarrow c})$. Et x_t est la projection de X_t dans l'image I_t (i.e., $x_t = \Pi_t \circ \Pi_c^{-1}(r_{t \rightarrow c})$, voir figure 10.6). Si la surface S était exacte, nous aurions :

$$\Pi_c \circ \Pi_t^{-1}(x_t) = x_c \quad (10.4)$$

Ceci est équivalent à dire que la demi-droite partant de O_t , passant par X_t intersecte la surface exacte de l'objet en un point 3D dont la projection dans l'image de la caméra c serait x_c . On note X_{new} la position de ce point, obtenu par triangulation de x_t et x_c (i.e. l'intersection des deux demi-droites $[O_t, x_t]$ and $[O_c, x_c]$).

Pour améliorer la surface, nous cherchons donc à la déformer pour que le point X_c se rapproche du point X_{new} , par exemple en définissant une force du type $F(X_c) = X_{new} - X_c$. Un problème majeur est que le point X_c n'est pas un sommet du maillage et qu'il n'est donc pas possible de définir une telle force. Pour pallier ce problème, nous avons proposé de calculer une force à

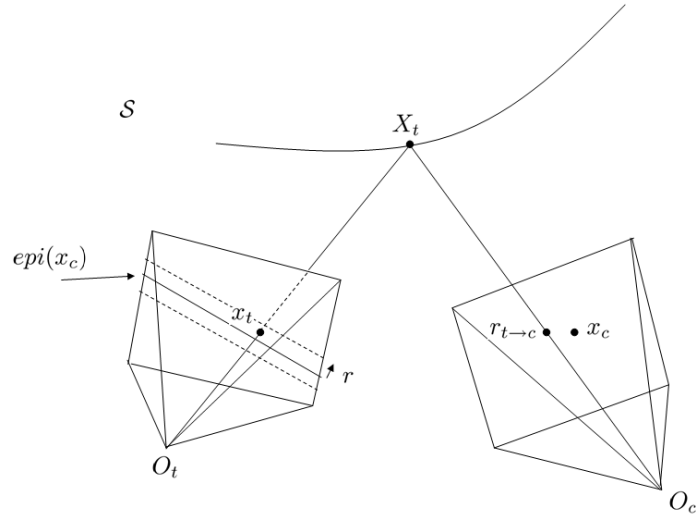


FIGURE 10.5 – Acceptation d’une paire de SIFT. Sur cette figure, nous cherchons à savoir si la paire de SIFT x_c et $\{r_{t \rightarrow c}^j\}$ est acceptable géométriquement. Elle l’est si x_t (i.e. l’antécédent de $\{r_{t \rightarrow c}^j\}$, c’est-à-dire que $x_t = \Pi_t \circ \Pi_c^{-1}(r_{t \rightarrow c})$) est proche de la droite épipolaire de x_c (notée $e\text{pi}(x_c)$). Par “proche”, nous entendons que la distance entre x_t et cette droite est inférieure à un seuil (noté r sur la figure).

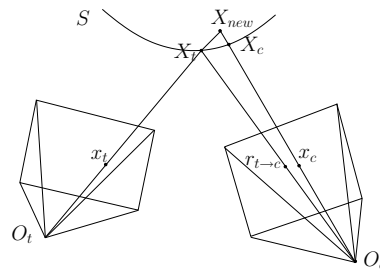


FIGURE 10.6 – Notations. O_c et O_t sont les centres optiques des appareils photos.

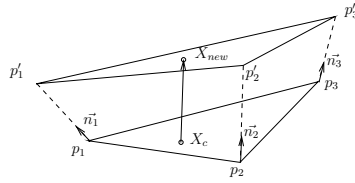


FIGURE 10.7 – Calcul de la force due à une correspondance de SIFT.

chacun des sommets du triangle qui contient X_c . Sur la figure 10.7, ces trois sommets sont notés p_1 , p_2 et p_3 , de normales \vec{n}_1 , \vec{n}_2 et \vec{n}_3 . On note p'_1, p'_2 ces points translatés selon leur normale, de telle manière que le triangle formé par ces nouveaux points soit parallèle au triangle (p_1, p_2, p_3) et contienne X_{new} . La force appliquée au sommet p_1 est alors définie comme : $F(p_1) = p'_1 - p_1$. Le même principe est appliqué aux points p_2 et p_3 . Il est aussi possible de chercher à déformer la surface en X_t vers X_{new} , ou bien aussi de générer deux forces pour ces deux déformations (en X_c et X_t). Nous avons fait quelques tests avec ces variantes, mais sans constater d'amélioration notable des résultats. Nous avons donc préféré travailler avec la version présentée ci-dessus (i.e. qui rapproche X_c de X_{new}).

Pour tous les couples de caméras proches, pour toutes les correspondances SIFT acceptées, les forces telles que définies ci-dessus sont calculées et enregistrées dans une liste pour chaque sommet du maillage. Pour un sommet v donné, la force finale $F_p(v)$ correspond à la moyenne des forces appartenant à sa liste.

Ce schéma de déformation est appliqué jusqu'à ce que le déplacement maximal d'un sommet soit inférieur à un seuil. Dans nos tests, cela permet notamment de creuser les concavités même si elles sont assez importantes. Toutefois, en utilisant uniquement cette approche, les détails les plus fins ne sont pas retrouvés. Dans la partie suivante, nous introduisons une nouvelle force de texture afin de retrouver ces détails.

10.5 Ajout d'un terme plus local

Dans cette seconde partie, nous cherchons à définir une force à appliquer aux sommets du maillage triangulé pour retrouver les petits détails de l'objet. Nous proposons une force qui minimise l'erreur SSD. Toutefois, nous nous démarquons de la littérature en travaillant toujours entre une image et une image virtuelle. L'avantage de cette approche est que nous n'avons pas besoin de calculer la déformation entre les fenêtres utilisées dans chaque image pour calculer cette erreur. Considérons une fenêtre centrée au point x_c dans la caméra c , projection du sommet v du maillage (voir figure 10.8 : on cherche à trouver la position de la fenêtre dans l'image virtuelle $R_{t \rightarrow c}$ qui minimise la SSD. L'algorithme le plus connu pour effectuer cette recherche est un suivi de Lucas-Kanade selon deux dimensions ([TK91]). Cette méthode a été très largement appliquée, notamment pour le suivi de points dans des séquences

vidéo. Toutefois, dans notre cas, il s'agit d'une application difficile car il est possible que les deux images soient largement différentes (car l'image virtuelle peut être issue d'une surface S très grossière). Nous proposons alors d'utiliser la géométrie épipolaire pour réduire l'espace de recherche à une seule dimension, et ainsi être beaucoup plus robuste. Cela serait simple si l'on cherchait la fenêtre qui minimise la SSD avec x_c sur la ligne épipolaire de x_c dans la caméra t (notée $epi_t(x_c)$), en utilisant $\Pi_t \circ \Pi_c^{-1}(x_c)$ comme position initiale. Mais cela souffrirait de distorsion entre les deux fenêtres (cette distorsion peut être forte dans le cas de fenêtres larges, utilisées par exemple pour une implémentation multi-échelle de Lucas-Kanade). Et l'on souhaite éviter ce problème en travaillant entre une image réelle et une image virtuelle, dans la même caméra.

Au lieu de faire cette recherche de point sur la droite $epi_t(x_c)$, le point qui minimise la SSD est cherché dans une direction équivalente, mais dans l'image $R_{t \rightarrow c}$. Pour faire cela, la surface S au point v est approchée par un plan de vecteur normal n (calculé comme la moyenne des vecteurs normaux des triangles ayant v pour sommet). La droite $epi_t(x_c)$ est projetée sur ce plan (notée epi_t^{3D} sur la figure 10.8), puis projetée par Π_c sur $R_{t \rightarrow c}$. La droite ainsi obtenue est notée $epi_t^{retro}(x_c)$. En utilisant la version 1D du suiveur de Lucas-Kanade [TK91], le déplacement d qui minimise la SSD le long de cette droite de vecteur directeur dir est solution de :

$$\left(\sum_{W_c} (g \cdot dir)^2 \right) \cdot d = \sum_{W_c} ((I_c - R_{t \rightarrow c}) \cdot (g \cdot dir)) \quad (10.5)$$

où W_c est la fenêtre d'intérêt autour de x_c et g le gradient des intensités de l'image I_c . Cette équation est résolue de manière itérative (méthode dite Newton-Raphson) pour trouver le déplacement qui minimise la SSD entre deux fenêtres d'intérêt calculées entre les images I_c et $R_{t \rightarrow c}$. L'intérêt de notre méthode est qu'avec la convergence de la surface, il doit y avoir une vraie correspondance pixel à pixel entre ces fenêtres sur les deux images. On rajoute que les points qui ont un gradient selon la direction dir trop faible ne sont pas considérés.

Une fois que l'on a trouvé la position qui minimise la SSD pour la projection d'un sommet du maillage v , on utilise une méthode similaire à celle de la première passe utilisant les SIFT pour exprimer une force. Toutefois, ce cas est plus simple puisque la force exprimée peut tout de suite être appliquée au sommet v . Enfin, cette nouvelle force par sommet est ajoutée à la liste des forces calculées avec les descripteurs SIFT. Pour un sommet v , la nouvelle force $F_p(v)$ est la moyenne de toutes ces forces.

10.6 Détails concernant l'implémentation

Quelques détails sont nécessaires à l'implémentation de la méthode décrite dans les paragraphes précédents. Tout d'abord, il y a le problème de comment calculer efficacement les fonctions Π_c^{-1} pour chaque caméra. La méthode naïve pour calculer $\Pi_c^{-1}(x)$ consiste à calculer l'intersection entre la demi droite $[O_c, x$ et le maillage S . Pour cela, on peut parcourir tous les triangles de manière

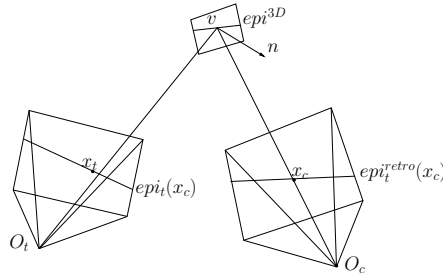


FIGURE 10.8 – Rétro-projection de la droite épipolaire.

exhaustive et vérifier s'ils sont intersectés ou non par la demi-droite. On peut aussi s'inspirer des structures de données utilisées pour faire du ray-tracing (type Octree). Toutefois, nous utilisons une technique qui exploite les capacités de projection de la carte graphique. C'est-à-dire que pour une caméra c , nous faisons un rendu du maillage noté M_c , en colorant chaque triangle de telle manière que l'index du triangle soit codé de manière unique sur les trois canaux RGB de sa couleur (i.e. si id est l'index d'un triangle, sa composante rouge vaut $(id \& 0x000000FF)$, sa composante verte $[(id \& 0x0000FF00)/2^8]$ et sa composante bleue $[(id \& 0x00FF0000)/2^{16}]$ où $\&$ est l'opérateur binaire ET). Ainsi, pour un pixel x_c dans la caméra c , l'index du triangle contenant $\Pi_c^{-1}(x_c)$ est obtenu directement en lisant le pixel $M_c(x_c)$. L'étape finale est de calculer l'intersection entre la demi-droite $[O_c, x_c)$ et cet unique triangle. A noter que cette approche permet d'avoir directement le triangle qui intersecte $[O_c, x_c)$ le plus proche de x_c . Cela est possible grâce à l'utilisation du Z-buffer de la carte graphique permettant d'afficher les triangles visibles depuis O_c (et pas ceux cachés par d'autres triangles).

Concernant la déformation du maillage, nous faisons l'hypothèse que le maillage initial a la bonne topologie (i.e. celle de l'objet réel). Ensuite, l'algorithme de déformation ne gère pas de changement topologique. Il existe des algorithmes de déformation de maillage gérant les changements de topologie (voir [ZBH07] pour un exemple de stéréovision n-vues par déformation de maillage acceptant de tels changements). Dans notre cas, nous utilisons des heuristiques pour éviter ces problèmes (e.g. pour éviter des auto-intersections de la surface). Nous nous assurons que la longueur des arêtes du maillage est bornée inférieurement et supérieurement. Le déplacement d'un sommet est aussi borné par cette longueur d'arête maximale. On s'assure également que l'angle entre deux triangles voisins ne soit pas trop grand. En théorie, ces contraintes ne garantissent pas qu'il n'y aura pas de changement de topologie. Mais en pratique, sur nos tests, cela était suffisant.

Enfin, nous appliquons notre algorithme en deux passes. Dans la première, en utilisant la force de texture due aux correspondances SIFT, nous utilisons un maillage avec des arêtes assez grandes. Cela permet d'autoriser des déplacements de sommets assez grands et donc de converger rapidement vers l'objet réel. Mais cette unique passe ne permet pas de retrouver les détails les plus fins. On applique alors la seconde passe (en ajoutant le critère de minimi-

sation de SSD) sur un maillage où l'on a divisé par deux la taille maximale des arêtes.

10.7 Résultats

Nous avons testé notre algorithme sur la base de test public de Seitz et al. [SCD⁺06]. Cette base est constituée de deux objets : un temple de dimensions $10\text{cm} \times 16\text{cm} \times 8\text{cm}$ et un dinosaure de taille réelle $7\text{cm} \times 9\text{cm} \times 7\text{cm}$. Les images prises de ces objets ont une dimension de 640×480 pixels.

Chaque objet est acquis de trois manières différentes :

1. 16 vues autour de l'objet (jeux appelés Temple-Sparse et Dino-Sparse)
2. 47 ou 48 vues autour de l'objet (jeux appelés Temple-Ring et Dino-Ring)
3. 312 ou 363 vues autour de l'objet (jeux appelés Temple-Full et Dino-Full)

Ces images constituent le jeu de données d'entrée de notre algorithme (ainsi que les paramètres internes de la caméra utilisée). Les auteurs disposent également d'une vérité terrain de ces objets obtenue par mesure laser. Enfin, il est possible de soumettre une surface triangulée sur leur site afin de mesurer plusieurs critères d'erreurs que nous présentons ici.

Dans cette partie, nous présentons d'abord des résultats qualitatifs puis quantitatifs en utilisant les mesures fournies par le site de [SCD⁺06]. Nous avons travaillé sur les deux objets mais n'avons pas testé en utilisant les jeux de données avec le plus grand nombre d'images (312 ou 363 vues) car un aussi grand nombre d'images nécessite un trop long temps de calcul.

10.8 Analyse qualitative

Les résultats de l'évaluation sont disponibles sur <http://vision.middlebury.edu/mview/eval>. La figure 10.9 montre des résultats intermédiaires ainsi que le résultat final. Le premier constat est qu'il n'y a pas d'erreur grossière sur ces surfaces. Des détails comme les escaliers ou encore les écailles dorsales du dinosaure sont bien retrouvées. Toutefois, on constate ici qu'un problème de notre approche est le bruit dans la reconstruction finale. Par exemple, il serait souhaitable que les escaliers soient constitués de marches plates (ce qui est le cas sur la vérité terrain). La zone plate de l'arrière du temple est aussi bruitée. Nous n'avons pas investigué suffisamment ces problèmes pour conclure sur leurs causes. Toutefois, il est possible que ces erreurs viennent de correspondances SIFT erronées. D'autre part, le terme de lissage est aussi en partie responsable de l'erreur (par exemple, avec ce terme, il n'est pas possible de retrouver les escaliers de manière très précise puisque cette force empêche les arêtes vives).

10.8.1 Analyse quantitative

Une fois un objet 3D reconstruit, on cherche à savoir si cette reconstruction est précise (critère appelé *précision*) et si elle est complète (critère appelé *complétude*). Pour mesurer ces valeurs, les auteurs de [SCD⁺06] mesurent

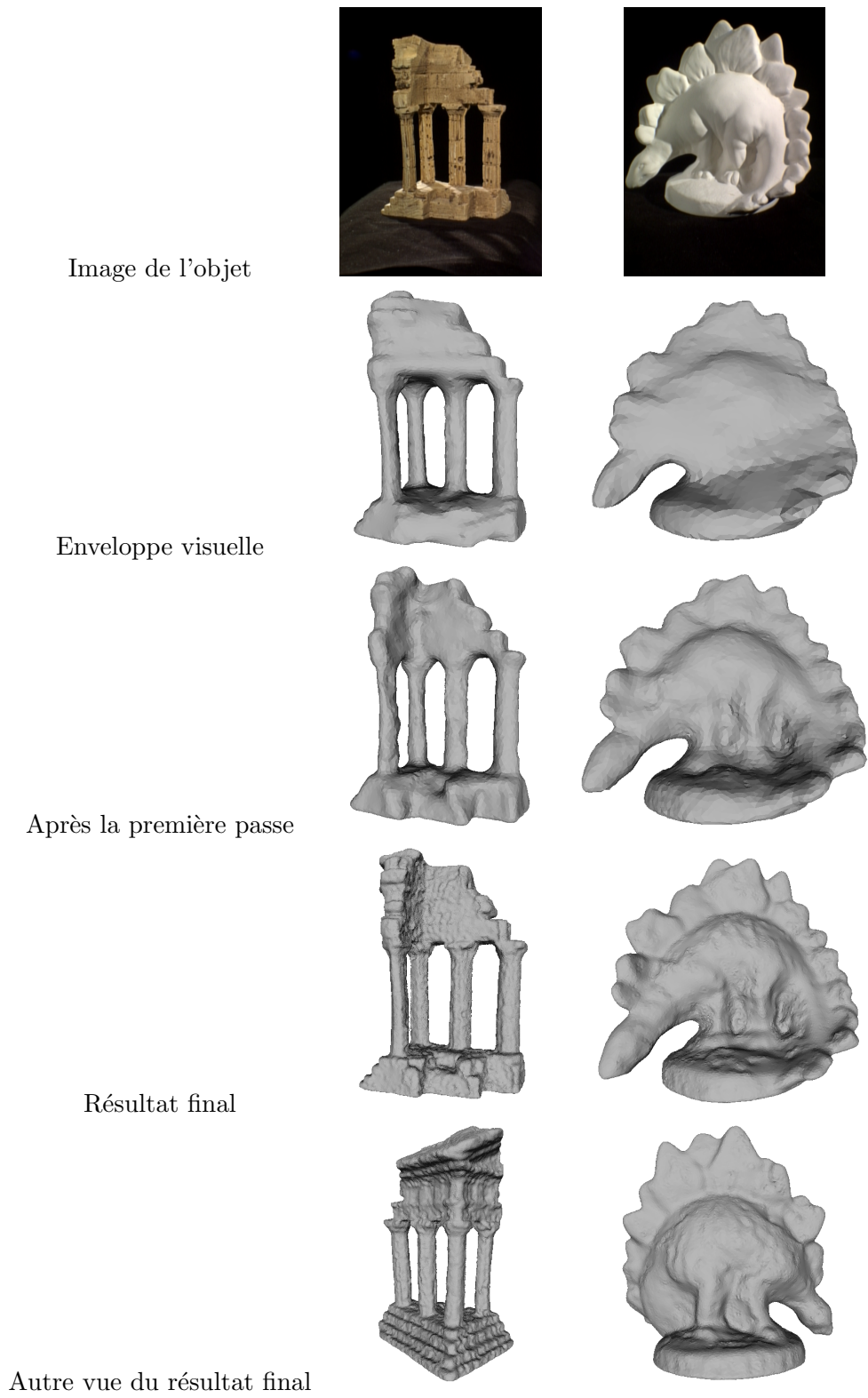


FIGURE 10.9 – Résultats sur le jeu de données d'évaluation de [SCD+06].

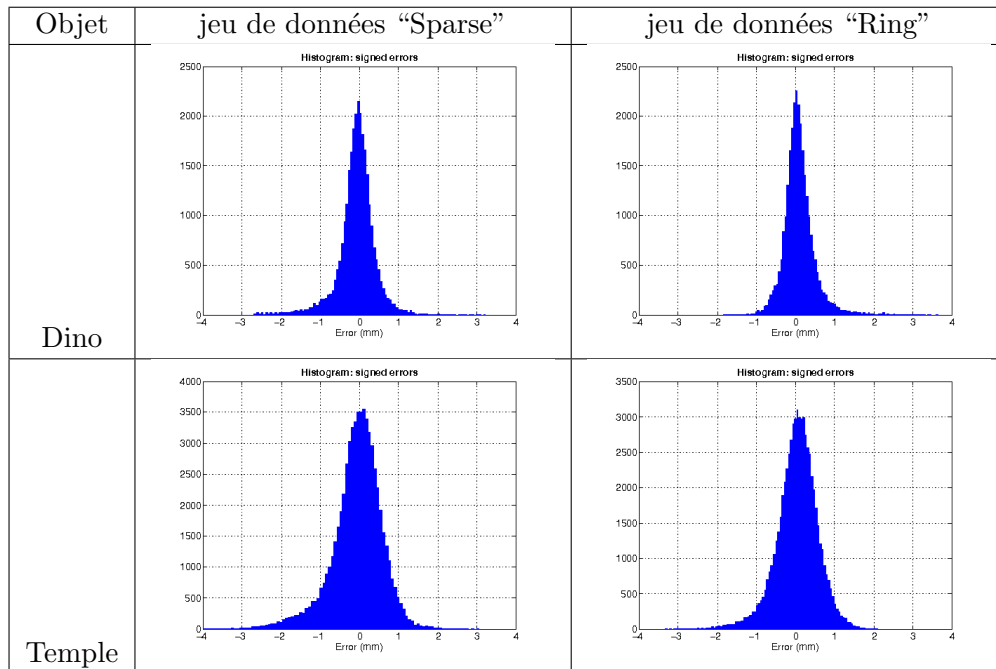


FIGURE 10.10 – Histogrammes des erreurs (distances entre un sommet du maillage et son plus proche point de la vérité terrain). Les titres des histogrammes sont en anglais car ces mesures sont issues du site <http://vision.middlebury.edu/mview/eval>.

d’abord, pour chaque sommet de la surface triangulée testée, sa distance avec le plus proche sommet de la surface triangulée de la vérité terrain (qui est très dense). La distance ainsi obtenue est signée selon que le point soit à l’intérieur ou à l’extérieur de la surface. La valeur de *précision* est ensuite mesurée à partir de ces erreurs comme étant la distance minimale d telle que $X\%$ des points soient à une distance inférieure à d de la vérité terrain. La mesure de *complétude* est le pourcentage de points de la vérité terrain qui sont à une distance inférieure à s de la surface testée.

Dans un premier temps, avant de mesurer les critères de *précision* et *complétude*, nous présentons les valeurs d’erreurs mesurées sur les sommets de la surface triangulée obtenue. Ces erreurs sont représentées dans des histogrammes dans la figure 10.10. Il est visible que ces erreurs approchent une distribution gaussienne, mais il y a notamment sur le Temple-Sparse une partie non négligeable d’erreurs négatives importantes (entre -1 et -3 mm). Nous n’avons pas trouvé d’explication pour ce biais dans les résultats.

Les figures 10.11.(a) et 10.11.(b) montrent les mesures de *précision* et de *complétude*. Ces figures confirment que, plus on a d’images de l’objet, meilleur est le résultat. Sur la première figure, on peut voir que la précision devient moins bonne à partir de 90%. Cela montre que notre algorithme doit être amélioré pour éviter que certaines régions soient vraiment mauvaises. Sur la seconde figure, on peut voir que pour le dinosaure, un fort pourcentage de l’objet est

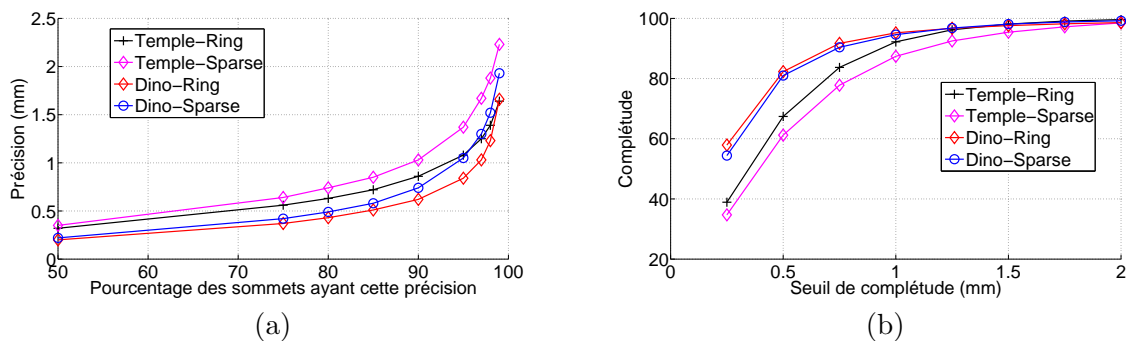


FIGURE 10.11 – (a) Courbes représentant la précision atteinte en fonction du pourcentage de points considérés. Par exemple, pour le jeu de données Dino-Sparse, en prenant 80% des points (les meilleurs), on a une précision de 0,5mm. (b) Complétude (pourcentage) en fonction du seuil d’erreur choisi sur les points. Par exemple, pour le jeu Dino-Sparse, si l’on considère uniquement les points de la vérité terrain à moins de 0,5mm de surface testée, cela représente 80% des points de cette vérité terrain.

correct si l’on place le seuil de complétude à une valeur supérieure à 1 mm. Mais dès que l’on descend en dessous de cette valeur, le pourcentage de complétude diminue fortement.

Enfin, pour comparer notre algorithme aux autres méthodes, nous choisissons comme critère de test la précision obtenue avec 90% des points ainsi que la complétude à 1,25mm. Ces valeurs sont celles classiquement utilisées pour le comparatif du site présentant les mesures. Le tableau 10.12 montre les résultats de plusieurs algorithmes de types différents. L’algorithme qui obtient les meilleurs scores de précision et complétude est celui de Furukawa-2. Toutefois, cet algorithme est assez lent. L’algorithme de Pons-GPU obtient également de meilleurs scores que notre algorithme tout en étant plus rapide, mais il s’agit d’une implémentation sur GPU, délicate à mettre en oeuvre. Enfin, notre algorithme obtient des scores proches de ceux de Vogiatzis basés sur une méthode par Graph-Cut, tout en étant plus rapide.

Un élément important à prendre en compte est que dans ces tests, notre algorithme retourne une surface triangulée qui comporte beaucoup moins de sommets que les autres algorithmes. Il serait intéressant de subdiviser encore notre maillage et de continuer sa déformation.

10.9 Conclusion

Dans ce chapitre, nous cherchions à proposer une méthode qui permettent d’utiliser des correspondances entre éléments 2D pour reconstruire une surface 3D précise, et ce, même si ces éléments sont des objets complexes (e.g. le contour d’une vitre).

Dans ce but, nous avons proposé une méthode de reconstruction 3D multi-vues par surface déformable qui utilise des correspondances 2D uniquement

Algorithme	Temple-Ring			Temple-Sparse			Dino-Ring			Dino-Sparse		
	P	C	Temps	P	C	Temps	P	C	Temps	P	C	Temps
Notre algo	0.86	96.2	52min	1.03	92.5	21min	0.62	96.7	27min	0.74	96.8	22min
Furukawa 2	0.55	99.1	6h02	0.62	99.2	3h33	0.33	99.6	9h04	0.42	99.2	3h44
Pons-GPU	0.6	99.5	31min	0.9	95.4	10min	0.55	99.0	13min	0.71	97.7	3min
Vogiatzis	0.76	96.2	54min	2.77	79.4	40min	0.49	96.7	59min	1.18	90.8	40min

FIGURE 10.12 – Résultats de plusieurs algorithmes sur les 4 jeux de données utilisés. P : précision en mm pour 90%. C : complétude en pourcentage, pour 1.25mm. Temps : temps de calcul normalisé pour un PC à 3.0 GHz.

entre des images de points de vues identiques (entre photos de l’objet et images virtuelles). Comme elles sont cherchées dans des images générées depuis une même caméra, ces correspondances sont simples à trouver. Dans une première passe de notre algorithme, des descripteurs SIFT sont utilisés pour démarrer la déformation de la surface. Même si la surface initiale est éloignée de l’objet véritable, et donc les images virtuelles peu ressemblantes au véritable objet, l’efficacité de ces descripteurs permet de déformer la surface 3D pour corriger les erreurs les plus importantes. Dans une seconde passe, nous minimisons une erreur locale de photo-cohérence, toujours entre images du même point de vue. Pour rendre cette étape robuste, nous avons introduit une méthode pour faire cette minimisation suivant une direction respectant localement la géométrie épipolaire. Dans ces deux passes, un intérêt de notre approche est qu’au fur et à mesure de l’évolution, la surface 3D devient plus proche de l’objet réel, les images virtuelles deviennent donc plus similaires aux photos, ce qui permet de trouver plus de correspondances de points et ainsi d’améliorer davantage la surface 3D. Comparativement, une approche “statique” qui cherche les correspondances au début du processus et ne les met pas à jour sera beaucoup plus dépendante de cette étape initiale.

Au niveau qualitatif, sur le benchmark de [SCD⁺06], notre approche obtient des résultats similaires à certaines méthodes classiques, mais n’obtient pas des résultats aussi précis que ceux des meilleurs algorithmes. Une des raisons est que parfois, certaines paires de SIFT ne sont pas correctes, générant des erreurs notables de la surface 3D. Nous devons améliorer notre algorithme face à ce problème pour améliorer le résultat final. Enfin, un avantage de notre algorithme est que la convergence entre la surface initiale et la surface finale est assez rapide, comparativement à certains autres algorithmes de la littérature.

La perspective principale de ces travaux est bien sûr d’utiliser des correspondances d’éléments 2D plus évolués que des points. Par exemple, dans le cas de reconstruction 3D de voitures, imaginons qu’un algorithme soit capable de détecter certaines caractéristiques sur une image (pare-brise, roue, vitre, plaque d’immatriculation...). Pour intégrer ces informations dans notre algorithme, il sera suffisant d’appliquer cette détection sur les photos initiales ainsi que sur les images virtuelles. La mise en correspondance doit être faite entre images du même point de vue. C’est-à-dire qu’il suffit de faire correspondre des points entre ces objets, à partir de leurs projections selon un même point de vue. Inver-

sement, le problème de faire correspondre deux projections d'un objet complexe (e.g. contour d'une vitre) entre deux images générées depuis des points de vue différents est beaucoup plus compliqué car la transformation entre les deux projections de l'objet n'est pas simple. Au mieux, cette transformation est une homographie si l'objet est plan, mais dans le cas général, il s'agit de la rétro-projection d'une image, depuis une première caméra, sur une surface 3D puis de sa projection dans une seconde caméra. Il faut aussi prendre en compte les occlusions alors qu'elles sont implicitement gérées avec notre méthode. La figure [10.13](#) illustre l'avantage de la méthode proposée.

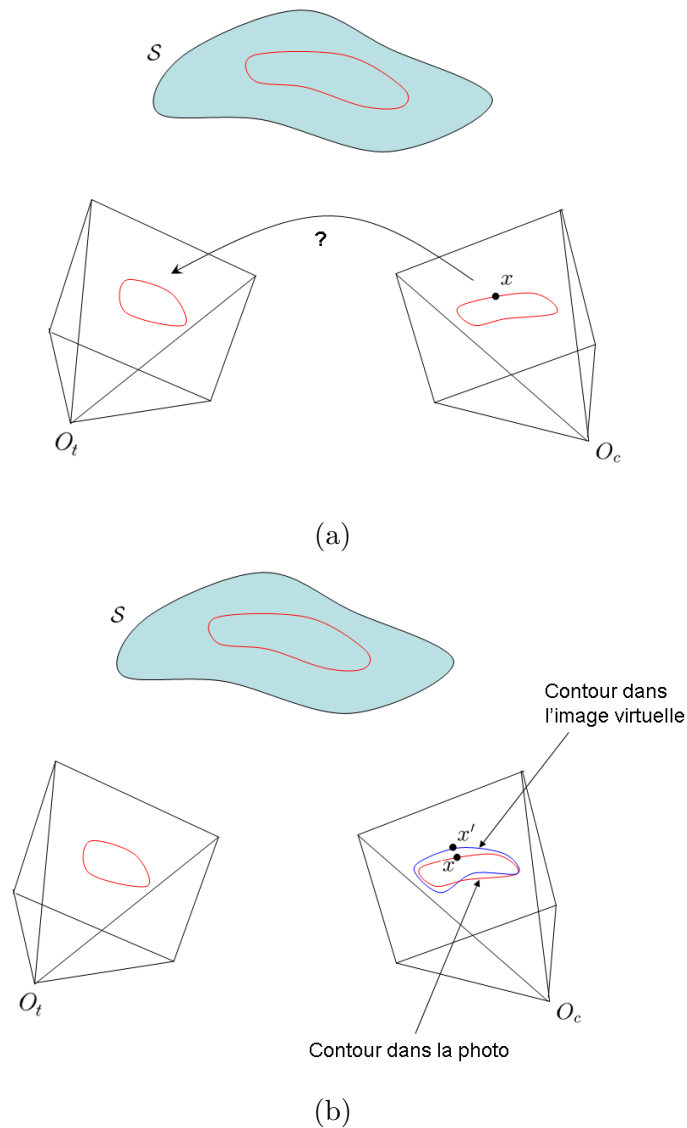


FIGURE 10.13 – (a) Sur cette figure, un contour de la surface \mathcal{S} est détecté sur deux images. La transformation entre les deux contours dans les plans images est complexe. Et le problème d'identifier le point correspondant à x dans l'image de la caméra de gauche n'est pas simple. (b) Dans cette configuration, l'image de gauche est utilisée pour texturer une surface 3D temporaire \mathcal{S} , approchant la surface exacte, qui est ensuite projetée dans le plan image de la caméra de droite. On va alors chercher à faire correspondre au point x , un point x' sur le contour détecté dans l'image virtuelle. Ce problème est beaucoup plus simple que celui du cas (a) car si la surface temporaire \mathcal{S} est proche de la surface de l'objet réel, la transformation entre les deux contours est proche de l'identité. Une fois ces correspondances établies, elles peuvent être utilisées directement dans notre algorithme pour corriger la surface de reconstruction.

Chapitre 11

Conclusion et Perspectives

Dans cette seconde partie de thèse, notre objectif premier était d'étudier la reconstruction 3D de voitures à partir de séquences vidéos. C'est-à-dire que nous souhaitions savoir quel niveau de qualité il était possible d'obtenir avec une méthode assez robuste pour être mise en place en conditions réelles. Un second objectif était d'étudier si les descripteurs SIFT utilisés dans la première partie de cette thèse pouvait permettre de développer des méthodes originales de reconstruction 3D.

Dans un premier temps, au vu de la difficulté de la reconstruction 3D de voitures (reflets, transparences...), nous avons décidé de nous orienter vers une méthode classique de reconstruction de nuage de points 3D que nous avons ensuite approchée de manière robuste par extrusion de polynômes. Notre contribution a été d'abord d'utiliser le point de fuite de la translation du véhicule pour rendre robuste la construction du nuage de points, puis de proposer une modélisation de voitures par extrusion de polynômes (un pour le coté et un pour la face avant). Cette méthode robuste peut être suffisante, par exemple, pour estimer le gabarit d'un véhicule, ou en faire une catégorisation simple (e.g. citadine, berline, monospace). De plus, dans nos tests, la direction de la largeur était recalculée pour chaque voiture. Dans l'optique d'une industrialisation, l'application serait plus robuste si la détermination de cette direction se faisait à partir du passage d'un grand nombre de voitures. Cela permettrait de tolérer des cas où les lignes horizontales de l'avant du véhicule sont mal détectées. A noter qu'au niveau industriel, cette application n'a pas été mise en place par l'entreprise partenaire. Enfin, si l'objectif est de déterminer le modèle du véhicule à partir de sa reconstruction 3D et d'une base de voitures en 3D, les résultats obtenus par cette méthode ne sont pas suffisants. Il est alors nécessaire d'affiner la reconstruction.

Dans ce but d'améliorer les reconstructions de voitures obtenues, nous avons proposé une méthode pour affiner la position d'un plan de symétrie par minimisation d'une erreur de photo-cohérence. Grâce à l'utilisation de la symétrie, nous avons aussi proposé de modéliser une demi-voiture par une surface spline. Toutefois, il reste à évaluer ces algorithmes sur des jeux de données plus importants pour valider leur robustesse si l'objectif est de les industrialiser. Une perspective de ces travaux est d'utiliser la modélisation de la symétrie proposée

pour tester la reconstruction 3D d'un objet symétrique (par rapport à un plan), à partir d'une seule image. On peut imaginer un algorithme qui va alterner une étape de recherche de la position du plan de symétrie, avec une étape de déformation de surface. Cette déformation de surface pourrait se faire en utilisant l'erreur image de Pons et al. [PKF07], mais en utilisant notre modélisation de la symétrie. On pourrait par exemple tester si cette approche est valable pour la reconstruction de visages.

Dans le chapitre 10, dans le but d'affiner les surfaces obtenues (de voiture ou de tout autre objet), nous avons travaillé sur une méthode de déformation de surface. Notre contribution principale a été de proposer une méthode pour laquelle les correspondances de points sont trouvées entre images issues d'un même point de vue (i.e. entre des images de l'objet à reconstruire et des rendus de la surface temporaire). Dans une première passe, l'efficacité des descripteurs SIFT, utilisés dans la première partie de cette thèse, permet de démarrer la déformation de la surface pour corriger les erreurs grossières de l'enveloppe visuelle (e.g. les concavités importantes). Dans une seconde passe, pour retrouver des détails plus fins, nous avons proposé de minimiser une erreur locale de photo-cohérence, toujours entre images de même point de vue. Afin de rendre robuste cette minimisation, nous avons utilisé la géométrie épipolaire pour restreindre l'espace de recherche à une droite. Finalement, cet algorithme offre des performances intéressantes. Il est rapide et permet d'obtenir des surfaces précises, même si le niveau de détail des algorithmes les plus performants de la littérature n'est pas atteint. L'avantage de cette approche est qu'un utilisateur peut fournir en entrée de l'algorithme des correspondances entre des éléments 2D dans des images de même point de vue. Il est tout à fait possible d'utiliser d'autres descripteurs que des points. Une perspective est alors d'appliquer cet algorithme pour affiner les surfaces de voitures obtenues par extrusion de polynômes. On pourrait par exemple chercher à mettre au point un algorithme qui va détecter les contours des vitres et du pare-brise, et s'en servir pour obtenir une reconstruction 3D précise de voitures.

11.0.1 Contributions

En résumé, les contributions de nos travaux sur la reconstruction 3D à partir d'images sont :

- l'utilisation des méthodes classiques pour la reconstruction 3D de voitures.
- un modèle de voiture par extrusion de polynômes, selon la longueur et la largeur de la voiture.
- une modélisation pour trouver un plan de symétrie d'un objet 3D.
- un algorithme original de déformation de surface qui utilise des correspondances SIFT, obtenues entre images générées depuis un même point de vue, pour guider l'évolution de la surface.

Conclusion générale

Dans nos travaux, nous avons présenté séparément deux parties, concernant, dans un premier temps, la recherche d'images similaires dans des grandes bases, puis, la reconstruction 3D à partir d'images. Mais de plus en plus, on trouve des applications liant ces deux problématiques au sein de projets ambitieux. Citons par exemple les travaux de Snavely et al. [SSS06, SSS08] dans le cadre du projet Photo Tour. Dans ce travail, les auteurs utilisent des grandes bases d'images (e.g. Flickr ou des images fournies par des requêtes textuelles sur le moteur Google), pour obtenir une représentation 3D de scènes. Par exemple, à partir de la requête "Trevi Fountain", les auteurs téléchargent un grand nombre d'images de cette fontaine et utilisent ensuite un algorithme de Structure-From-Motion pour en obtenir une reconstruction 3D (sous forme de nuage de points). Dans ce cas, la recherche d'images utilise les mots clés associés aux images. Mais on peut tout à fait imaginer un algorithme complètement automatique qui va utiliser toutes les images disponibles d'une scène, même celles non taggées. Il faudra alors utiliser un algorithme de recherche d'images similaires, tel que celui présenté dans la première partie de cette thèse, qui va fournir les correspondances d'images pour ensuite appliquer un algorithme de reconstruction 3D. Et l'étape suivante sera d'utiliser un algorithme de reconstruction 3D qui aboutit à un résultat précis, à partir de ces images. Il sera alors possible, en utilisant les photos des sites Internet communautaires, tels Flickr, d'obtenir des modélisations 3D d'un très grand nombre de lieux et de monuments, à l'échelle mondiale.

Liste des publications de l'auteur

- A. Auclair and L. D. Cohen and N. Vincent, A Robust Approach for 3D Model Reconstruction from a Video Sequence of Cars, In Proceedings of Scandinavian Conference on Image Analysis (SCIA), Aalborg, Denmark, Juin 2007.
- A. Auclair and L. D. Cohen and N. Vincent, How to use SIFT vectors to Analyze an Image with Database Templates, In Proceedings of the 5th International Workshop on Adaptive Multimedia Retrieval (AMR), Paris, France, Juillet 2007.
- A. Auclair and L. D. Cohen and N. Vincent, Using Point Correspondences Without Projective Deformation For Multi-View Stereo Reconstruction, In Proceedings of the International Conference on Image Processing (ICIP), San Diego, U.S.A., Octobre 2008.
- A. Auclair and L. D. Cohen and N. Vincent, Hachage de descripteurs locaux pour la recherche d'images similaires, In Orasis'09, Congrès des jeunes chercheurs en vision par ordinateur, Trégastel, France, Juin 2009.

Bibliographie

- [AAG96] Y. Amit, G. August, and D. Geman. Shape quantization and recognition with randomized trees. *Neural Computation*, 9 :1545–1588, 1996.
- [ACV07a] A. Auclair, L. D. Cohen, and N. Vincent. How to use SIFT vectors to analyze an image with database templates. In *Proceedings of the 5th International Workshop on Adaptive Multimedia Retrieval (AMR), Paris, France, July 2007*.
- [ACV07b] A. Auclair, L. D. Cohen, and N. Vincent. A robust approach for 3d model reconstruction from a video sequence of cars. In *Proceedings of Scandinavian Conference on Image Analysis (SCIA), Aalborg, Denmark, June 2007*.
- [ACV08] A. Auclair, L. D. Cohen, and N. Vincent. Using point correspondences without projective deformation for multi-view stereo reconstruction. In *Proceedings of the International Conference on Image Processing (ICIP), San Diego, U.S.A., October 2008*.
- [ACV09] A. Auclair, L. D. Cohen, and N. Vincent. Hachage de descripteurs locaux pour la recherche d'images similaires. In *Orasis'09, Trégastel, France, Juin 2009*.
- [ADV03] A. Almansa, A. Desolneux, and S. Vamech. Vanishing points detection without any a priori information. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 25(4) :502–507, april 2003.
- [Ame99] N. Amenta. The crust algorithm for 3D surface reconstruction. In *SCG '99 : Proceedings of the fifteenth annual symposium on Computational geometry, Miami Beach, Florida, United States*, pages 423–424, 1999.
- [AMN⁺98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6) :891–923, 1998.
- [AS95] D. Adalsteinsson and J. A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2) :269–277, 1995.
- [ATL] Automatically tuned linear algebra software (atlas), <http://math-atlas.sourceforge.net>.
- [BBGK06] A. Bronstein, M. Bronstein, E. Gordon, and R. Kimmel. High-resolution structured light range scanner with automatic calibra-

- tion. Technical report, Dept. of Computer Science, Technion, Israel, 2003., 2003-06.
- [BBK01] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces : Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3) :322–373, 2001.
- [BDC01] A. Broadhurst, T. Drummond, and R. Cipolla. A probabilistic framework for space carving. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV), Vancouver, Canada*, pages 388–393, 2001.
- [Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9) :509–517, 1975.
- [BI98] A. Blake and M. Isard. *Active Contours*. Springer-Verlag London, 1998.
- [BJ00] Y. Boykov and M.-P. Jolly. Interactive organ segmentation using graph cuts. In *Medical Image Computing and Computer-Assisted Intervention*, pages 276–286, 2000.
- [Bjo96] A. Bjorck. *Numerical Methods for Least Squares Problems*. SIAM : Society for Industrial and Applied Mathematics, 1996.
- [BKK96] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree : An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB), San Francisco, USA*, pages 28–39. Morgan Kaufmann Publishers, 1996.
- [BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree : an efficient and robust access method for points and rectangles. In *SIGMOD '90 : Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, New York, NY, USA, 1990. ACM Press.
- [BL97] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), San Juan, Puerto Rico*, page 1000, 1997.
- [BL02] M. Brown and D. Lowe. Invariant features from interest point groups. In *Proceedings of the British Machine Vision Conference (BMVC), Cardiff, UK*, 2002.
- [BMP02] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, 24(4) :509–522, 2002.
- [Bou00] J.-Y. Bouguet. Pyramidal implementation of the Lucas Kanade feature tracker, OpenCV library documentation, 2000.
- [Boy03] Y. Boykov. Computing geodesics and minimal surfaces via graph cuts. In *Proceedings of International Conference on Computer Vision (ICCV), Nice, France*, pages 26–33, 2003.

- [Bri95] S. Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, pages 574–584, 1995.
- [BSA06] A. Bevilacqua, L. Di Stefano, and P. Azzari. People tracking using a time-of-flight depth sensor. In *Proc. of IEEE International Conference on Video and Signal Based Surveillance (AVSS), Sydney, Australia*, pages 89 – 89, 2006.
- [BTG06] H. Bay, T. Tuytelaars, and L. J. Van Gool. Surf : Speeded up robust features. In *Proceedings of the European Conference on Computer Vision (ECCV), Graz, Austria*, pages 404–417, 2006.
- [BVZ98] Y. Boykov, O. Veksler, and R. Zabih. Markov random fields with efficient approximations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Santa Barbara, USA*, pages 648–655, 1998.
- [BVZ01] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 23(11) :1222–1239, 2001.
- [CC93] L.D. Cohen and I. Cohen. Finite element methods for active contour models and balloons for 2-D and 3-D images. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 15 :1131–1147, 1993.
- [CG00] R. Cipolla and P. Giblin. *Visual motion of curves and surfaces*. Cambridge University Press, New York, NY, USA, 2000.
- [CGA] Computational geometry algorithms library, <http://www.cgal.org/>.
- [CL96] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH, New Orleans, USA*, pages 303–312, 1996.
- [Coh91] L. D. Cohen. On active contour models and balloons. *Computer Vision, Graphics, and Image Processing. Image Understanding*, 53(2) :211–218, 1991.
- [CPIZ07] O. Chum, J. Philbin, M. Isard, and A. Zisserman. Scalable near identical image and shot detection. In *CIVR '07 : Proceedings of the 6th ACM international conference on Image and video retrieval, Amsterdam, The Netherlands*, pages 549–556, 2007.
- [CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree : An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, August 1997.
- [DF95] F. Devernay and O. Faugeras. Automatic calibration and removal of distortion from scenes of structured environments. In *SPIE, volume 2567*, 1995.
- [DFS04] J.-D. Durou, M. Falcone, and M. Sagona. A Survey of Numerical Methods for Shape from Shading. Rapport de Recherche 2004-2-R, Institut de Recherche en Informatique de Toulouse, Toulouse, France, January 2004.

- [DG03] T. Dey and S. Goswami. Tight cocone : A water-tight surface reconstructor. In *Proceedings of the 8th ACM Symposium on Solid Modeling and Applications, Seattle, USA*, 2003.
- [DH72] R. O. Duda and P.E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1) :11–15, 1972.
- [DIIM04] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG '04 : Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, New York, NY, USA, 2004. ACM.
- [DLÁ⁺07] K. Dadason, H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. Videntifier : identifying pirated videos in real-time. In *ACM Multimedia, Augsburg, Germany*, pages 471–472, 2007.
- [DMA02] M. Desbrun, M. Meyer, and P. Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum, Saarbrücken, Germany*, 21 :209–218, 2002. Eurographics conference proceedings.
- [dVMNK02] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten. Efficient k-nn search on vertically decomposed data. In *SIGMOD '02 : Proceedings of the 2002 ACM SIGMOD international conference on Management of data, Madison, USA*, pages 322–333, 2002.
- [DYQS04] Y. Duan, L. Yang, H. Qin, and D. Samaras. Shape reconstruction from 3D and 2D data using pde-based deformable surfaces. In *Proceedings of the European Conference on Computer Vision (ECCV) (3), Prague, Czech Republic*, pages 238–251, 2004.
- [EDD⁺95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics*, 29(Annual Conference Series) :173–182, 1995.
- [ES04] C. H. Esteban and F. Schmitt. Silhouette and stereo fusion for 3D object modeling. *Comput. Vis. Image Underst. (CVIU)*, 96(3) :367–392, 2004.
- [Fal86] C. Faloutsos. Multiattribute hashing using gray codes. *SIGMOD : Proceedings of the international conference on Management of data, Washington, USA*, 15(2) :227–238, 1986.
- [Fau93] O. Faugeras. *Three-Dimensional Computer Vision*. The MIT Press, 1993.
- [FB81] M. A. Fischler and R. C. Bolles. Random sample consensus : A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6) :381–395, 1981.
- [FK98] O. D. Faugeras and R. Keriven. Complete dense stereovision using level set methods. In *Proceedings of the European Conference on Computer Vision (ECCV), Freiburg, Germany*, pages 379–393, 1998.

- [FKS03] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 301–312, 2003.
- [Flo] Simon Flory. Diplomarbeit : Fitting b-splines curves to point clouds in the presence of obstacles.
- [Flo98] M. S. Floater. Tech. report, how to approximate scattered data by least squares, 1998.
- [Flo03] M. S. Floater. Mean value coordinates. *Comput. Aided Geom. Des.*, 20(1) :19–27, 2003.
- [FLP01] O. Faugeras, Q.-T. Luong, and T. Papadopoulou. *The Geometry of Multiple Images : The Laws That Govern The Formation of Images of A Scene and Some of Their Applications*. MIT Press, Cambridge, MA, USA, 2001.
- [Fod02] I. K. Fodor. A survey of dimension reduction techniques. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, 2002.
- [FP06] Y. Furukawa and J. Ponce. High-fidelity image-based modeling. Technical report, University of Illinois at Urbana Champaign, 2006.
- [FP07] Y. Furukawa and J. Ponce. Accurate, dense, and robust multi-view stereopsis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Minneapolis, USA, 2007*.
- [FS07] J. J. Foo and R. Sinha. Pruning sift for scalable near-duplicate image matching. In James Bailey and Alan Fekete, editors, *Eighteenth Australasian Database Conference (ADC 2007)*, volume 63 of *CRPIT*, pages 63–71, Ballarat, Australia, 2007. ACS.
- [GBS05] J.-M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders. The Amsterdam library of object images. *Int. J. Comput. Vision (IJCV)*, 61(1) :103–112, 2005.
- [GCS06] M. Goesele, B. Curless, and S. M. Seitz. Multi-view stereo revisited. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), New York, USA, volume 2, pages 2402–2409, Los Alamitos, CA, USA, 2006*. IEEE Computer Society.
- [GGB06] M. Grabner, H. Grabner, and H. Bischof. Fast approximated SIFT. In *Proceedings of the Asian Conference on Computer Vision (ACCV) (1), Hyderabad, India, pages 918–927, 2006*.
- [GIM99] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *The VLDB Journal*, pages 518–529, 1999.

- [GMDP00] V. Gouet, P. Montesinos, R. Deriche, and D. Pele. Evaluation de detecteurs de points d'interet pour la couleur. In *Reconnaissance des Formes et Intelligence Artificielle (RFIA), volume II, pages 257–266, Paris, France, 2000.*, 2000.
- [Gut84] A. Guttman. R-trees : a dynamic index structure for spatial searching. In *SIGMOD '84 : Proceedings of the 1984 ACM SIGMOD international conference on Management of data, Boston, USA*, pages 47–57, 1984.
- [Har93] C. Harris. *Geometry from visual motion*. MIT Press, Cambridge, MA, USA, 1993.
- [Har97] R. I. Hartley. In defense of the eight-point algorithm. *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, 19(6) :580–593, 1997.
- [HK06] A. Hornung and L. Kobbelt. Hierarchical volumetric multi-view stereo reconstruction of manifold surfaces based on dual graph embedding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), New York, USA*, pages 503–510, 2006.
- [HK07] M. Habbecke and L. Kobbelt. A surface-growing approach to multi-view stereo reconstruction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Minneapolis, USA*. IEEE Computer Society, 2007.
- [Hor07] Kai Hormann. Tech. report, barycentric mappings, institut für informatik, TU clausthal. Technical report, 2007.
- [HS88] C. Harris and M. Stephens. A Combined Corner and Edge Detector. In *4th ALVEY Vision Conference*, pages 147–151, 1988.
- [HS97] R. I. Hartley and P. Sturm. Triangulation. *Computer Vision and Image Understanding (CVIU)*, 68(2) :146–157, 1997.
- [HSW89] J. Hoschek, F.-J. Schneider, and P. Wassum. Optimal approximate conversion of spline surfaces. *Comput. Aided Geom. Des.*, 6(4) :293–306, 1989.
- [Hus07] T. Hussmann, S. Liepert. Robot vision system based on a 3D-tof camera. In *Instrumentation and Measurement Technology Conference Proceedings, IMTC 2007, IEEE*, pages 1–5, may 2007.
- [HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN : 0521540518, second edition, 2004.
- [IM98] P. Indyk and R. Motwani. Approximate nearest neighbors : Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.
- [Jag90] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD : Proceedings of the international conference on Management of data, Atlantic City, USA*, 19(2) :332–342, 1990.

- [JASG08] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros. Query-adaptative locality sensitive hashing. In *International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 2008.
- [JDS08] H. Jegou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Proceedings of the European Conference on Computer Vision (ECCV), Marseille, France, LNCS*. Springer, oct 2008.
- [JFB04] A. Joly, C. Frélicot, and O. Buisson. Feature statistical retrieval applied to content-based copy identification. In *Proceedings of the International Conference on Image Processing (ICIP), Singapore*, pages 681–684, 2004.
- [KF75] Patrenahalli M Narendra Keinosuke Fukunaga. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, 1975.
- [KKZ03] J. Kim, V. Kolmogorov, and R. Zabih. Visual correspondence using energy minimization and mutual information. In *ICCV '03 : Proceedings of the Ninth IEEE International Conference on Computer Vision, Nice, France*, page 1033. IEEE Computer Society, 2003.
- [Kop00] M. Koppen. The curse of dimensionality. In *5th Online World Conference on Soft Computing in Industrial Applications (WSC5)*, September 2000.
- [Koz98] R. Kozera. An Overview of the Shape from Shading Problem. *Machine Graphics and Vision*, 7(1-2) :291–312, 1998.
- [KS97] N. Katayama and S. Satoh. The sr-tree : an index structure for high-dimensional nearest neighbor queries. In *SIGMOD '97 : Proceedings of the 1997 ACM SIGMOD international conference on Management of data, Tucson, USA*, pages 369–380, 1997.
- [KS00] K.N. Kutulakos and S.M. Seitz. A theory of shape by space carving. *International Journal of Computer Vision (IJCV)*, 38(3) :199–218, July 2000.
- [KS04] Y. Ke and R. Sukthankar. Pca-sift : A more distinctive representation for local image descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Washington, USA*, pages 506–513, 2004.
- [KSH04] Y. Ke, R. Sukthankar, and L. Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MULTIMEDIA '04 : Proceedings of the 12th annual ACM international conference on Multimedia*, pages 869–876, New York, NY, USA, 2004. ACM Press.
- [KWT88] Michael Kass, Andrew Witkin, and Demetri Terzopoulos. Snakes : Active contour models. *International Journal of Computer Vision (IJCV)*, V1(4) :321–331, January 1988.

- [KZ02] V. Kolmogorov and R. Zabih. Multi-camera scene reconstruction via graph cuts. In *Proceedings of the European Conference on Computer Vision (ECCV), Copenhagen, Denmark*, pages 82–96, 2002.
- [KZ04] V. Kolmogorov and R. Zabih. What energy functions can be minimized via graph cuts? *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 26(2) :147–159, 2004.
- [LA04] M.I.A. Lourakis and A.A. Argyros. The design and implementation of a generic sparse bundle adjustment software package based on the levenberg-marquardt algorithm. Technical Report 340, Institute of Computer Science - FORTH, Heraklion, Crete, Greece, Aug. 2004. Available from <http://www.ics.forth.gr/~lourakis/sba>.
- [LÁJA05] H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. Efficient and effective image copyright enforcement. In *BDA, 21èmes Journées Bases de Données Avancées, BDA 2005, Saint Malo, 17-20 octobre 2005, Actes (Informal Proceedings)*, 2005.
- [LAJA06] H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. Scalability of local image descriptors : a comparative study. In *MULTIMEDIA '06 : Proceedings of the 14th annual ACM international conference on Multimedia, Santa Barbara, USA*, pages 589–598. ACM Press, 2006.
- [Lau94] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, 16(2) :150–162, 1994.
- [LE06] Gareth Loy and Jan-Olof Eklundh. Detecting symmetry and symmetric constellations of features. In *Proceedings of the European Conference on Computer Vision (ECCV), Graz, Austria*, volume 2, 2006.
- [LF06] V. Lepetit and P. Fua. Keypoint recognition using randomized trees. *Transactions on Pattern Analysis and Machine Intelligence, PAMI*, 28(9) :1465–1479, 2006.
- [Lin98] T. Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision (IJCV)*, 30(2) :77–116, 1998.
- [LJF94] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree : An index structure for high-dimensional data. *VLDB J.*, 3(4) :517–542, 1994.
- [LK01] J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD International Conference on Management of Data, Santa Barbara, USA*, 30(1) :19–24, 2001.
- [LKP06] P. Labatut, R. Keriven, and J.-P. Pons. Fast level set multi-view stereo on graphics hardware. *Proceedings of the International*

Symposium on 3D Data Processing, Visualization and Transmission (3DPVT), Chapell Hill, USA, 0 :774–781, 2006.

- [LMGY04] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS), Vancouver, Canada, 2004.*
- [LOB] F. Lustenberg, T. Oggier, and B. Butthen. Swissranger sr3000 and first experiences based on miniaturized 3D-tof cameras. Technical report, CSEM, Swiss Center for Electronics and Microtechnology.
- [Lon81] H. C. Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293 :133–135, September 1981.
- [Low99] D. G. Lowe. Object recognition from local scale-invariant features. In *Proc. of the International Conference on Computer Vision ICCV, Corfu*, pages 1150–1157, 1999.
- [Low01] D. G. Lowe. Local feature view clustering for 3D object recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Hawaii, USA, 2001.*
- [Low04] D. G. Lowe. Distinctive image features from scale-invariant keypoints. In *International Journal of Computer Vision (IJCV)*, volume 20, pages 91–110, 2004.
- [LWS97] S. Lee, G. Wolberg, and S. Y. Shin. Scattered data interpolation with multilevel B-splines. *IEEE Transactions on Visualization and Computer Graphics*, 3(3) :228–244, 1997.
- [MAF⁺04] C. M enier, J. Allard, J.-S. Franco, B. Raffin, and E. Boyer. Marker-less real time 3D modeling for virtual reality. In *Immersive Projection Technology, 2004.*
- [MCMP02] J. Matas, O. Chum, U. Martin, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *Proceedings of the British Machine Vision Conference (BMVC), Cardiff, UK*, volume 1, pages 384–393, 2002.
- [MDS05] E. N. Mortensen, H. Deng, and L. Shapiro. A sift descriptor with global context. In *CVPR '05 : Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1, San Diego, USA*, pages 184–190, 2005.
- [Mor66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. IBM Ltd., Ottawa, Canada, 1966.
- [MS01] K. Mikolajczyk and C. Schmid. Indexing based on scale invariant interest points. In *Proceedings of the 8th International Conference on Computer Vision, Vancouver, Canada*, pages 525–531, 2001.
- [MS02] K. Mikolajczyk and C. Schmid. An affine invariant interest point detector. In *Proceedings of the 7th European Conference on Computer Vision (ECCV), Copenhagen, Denmark*, pages 128–142. Springer, 2002. Copenhagen.

- [MS04] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *International Journal of Computer Vision (IJCV)*, 60(1) :63–86, 2004.
- [MS05] K. Mikolajczyk and C. Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 27(10) :1615–1630, 2005.
- [MSKS04] Y. Ma, S. Soatto, J. Kosecka, and S. S. Sastry. *An Invitation to 3-D Vision*. Springer Verlag, 2004.
- [MTJ07] F. Moosmann, B. Triggs, and F. Jurie. Fast discriminative visual codebooks using randomized clustering forests. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Advances in Neural Information Processing Systems 19 (NIPS), Vancouver, Canada*, pages 985–992. MIT Press, 2007.
- [MTS⁺05] K. Mikolajczyk, T. Tuytelaars, C. Schmid, A. Zisserman, J. Matas, F. Schaffalitzky, T. Kadir, and L. Van Gool. A comparison of affine region detectors. *International Journal of Computer Vision (IJCV)*, 65(1/2) :43–72, 2005.
- [MWTN04] T. Matsuyama, X. Wu, T. Takai, and S. Nobuhara. Real-time 3D shape reconstruction, dynamic 3D mesh deformation, and high fidelity visualization for 3D video. *Computer Vision and Image Understanding (CVIU)*, 96(3) :393–434, 2004.
- [Neo] Neodownloader, <http://www.neowise.com/neodownloader/>.
- [NM03] S. Nobuhara and T. Matsuyama. Dynamic 3D shape from multi-viewpoint images using deformable mesh model. In *Processings of 3rd International Symposium on Image and Signal Processing and Analysis, Rome, Italy*, pages pp. 192–197, Septembre 2003.
- [NS06] D. Nistér and H. Stewénus. Scalable recognition with a vocabulary tree. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), New York, USA*, pages 2161–2168, 2006.
- [NVI08] NVIDIA. CUDA computed unified device architecture, programming guide, version 2.0, Juillet 2008.
- [OS88] S. Osher and J. A. Sethian. Fronts propagating with curvature dependent speed : algorithms based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79 :12–49, 1988.
- [Pan06] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA '06 : Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, Miami, Florida*, pages 1186–1195, 2006.
- [PCF06] E. Prados, F. Camilli, and O. Faugeras. A unifying and rigorous shape from shading method adapted to realistic data and applications. *Journal of Mathematical Imaging and Vision*, 25(3) :307–328, 2006.

- [PCI⁺07] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Minneapolis, USA, 2007*.
- [Per06] S. May ; B. Werner ; H. Surmann ; K. Pervolz. 3D time-of-flight cameras for mobile robotics. In *Proceedings of International Conference on Intelligent Robots and Systems, 2006 IEEE/RSJ*, pages 790 – 795, october 2006.
- [PGV⁺04] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch. Visual modeling with a hand-held camera. *Int. J. Comput. Vision (IJCV)*, 59(3) :207–232, 2004.
- [PH03] H. Pottmann and M. Hofer. Geometry of the squared distance function to curves and surfaces. *Visualization and mathematics*, 3 :223–244, 2003.
- [PKF07] J.-P. Pons, R. K., and O. Faugeras. Multi-view stereo reconstruction and scene flow estimation with a global image-based matching score. *Int. J. Comput. Vision (IJCV)*, 72(2) :179–193, 2007.
- [SAH08] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, CVPR 08, Anchorage, USA, 2008*.
- [SCD⁺06] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), New York, USA, volume 1, pages 519–528, 2006*. evaluation available at <http://vision.middlebury.edu/mview/eval>.
- [Sch99] C. Schmid. A structured probabilistic model for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Ft. Collins, USA, pages 2485–2490, 1999*.
- [SCMS01] G. Slabaugh, B. Culbertson, T. Malzbender, and R. Schafer. A survey of methods for volumetric scene reconstruction from photographs. Technical report, 2001.
- [SD84] Geman S. and Geman D. Stochastic relaxation, gibbs distributions and the bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 6 :721–741, 1984.
- [SD99] S. M. Seitz and C. R. Dyer. Photorealistic scene reconstruction by voxel coloring. *Int. J. Computer Vision (IJCV)*, 35(2) :151–173, 1999.
- [SDI06a] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision : Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.

- [SDI06b] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision : Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.
- [Set99] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.
- [SKS+02] R. Swaminathan, S. B. Kang, R. Szeliski, A. Criminisi, and S. K. Nayar. On the motion and appearance of specularities in image sequences. In *Proceedings of the European Conference on Computer Vision (ECCV) (1), Copenhagen, Denmark*, pages 508–523, 2002.
- [SM86] G. Salton and M.J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.
- [SMB00] C. Schmid, R. Mohr, and C. Bauckhage. Evaluation of interest point detectors. *International Journal of Computer Vision (IJCV)*, 37(2) :151–172, 2000.
- [SRF87] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r -tree : A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.
- [SSG06] M. Pollefeys S.N. Sinha, J.M. Frahm and Y. Genc. Gpu-based video feature tracking and matching. In *EDGE 2006, workshop on Edge Computing Using New Commodity Architectures, Chapel Hill, USA*, 2006.
- [SSS06] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism : Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846, New York, NY, USA, 2006. ACM Press.
- [SSS08] N. Snavely, S. M. Seitz, and R. Szeliski. Modeling the world from Internet photo collections. *International Journal of Computer Vision*, 80(2) :189–210, November 2008.
- [SSZ01] D. Scharstein, R. Szeliski, and R. Zabih. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. In *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision, Kauai, HI, Dec. 2001.*, 2001.
- [SZ03] J. Sivic and A. Zisserman. Video Google : A text retrieval approach to object matching in videos. In *Proceedings of the International Conference on Computer Vision (ICCV), Nice, France*, volume 2, pages 1470–1477, October 2003.
- [TD06] S. Tran and L.S. Davis. 3D surface reconstruction using graph cuts with surface constraints. In *Proceedings of the European Conference on Computer Vision (ECCV), Graz, Austria*, pages II : 219–231, 2006.
- [TK91] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.

- [TM97] P. H. S. Torr and D. W. Murray. The development and comparison of robust methods for estimating the fundamental matrix. *Int. J. Comput. Vision (IJCV)*, 24(3) :271–300, 1997.
- [Tor02] P.H.S. Torr. Tech. report, a structure and motion toolkit in matlab : Interactive adventures in S and M, microsoft research. Technical report, May 2002.
- [TPG99] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra : improved iso-surface extraction. *Computers and Graphics*, 23 :583–598, 1999.
- [Tsa86] R. Y. Tsai. An efficient and accurate camera calibration technique for 3D machine vision. In *Proc. of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Miami Beach, USA*, pages 364–374, 1986.
- [Tut63] W.T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13, 1963.
- [TZ00] P. H. S. Torr and A. Zisserman. MLESAC : A new robust estimator with application to estimating image geometry. *Computer Vision and Image Understanding (CVIU)*, 78 :138–156, 2000.
- [Uhl91] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4) :175–179, 1991.
- [VCPF08a] E. Valle, M. Cord, and S. Philipp-Foliguet. Fast identification of visual documents using local descriptors. In *Symposium on Document Engineering*, Sao Paulo, Brésil, sept 2008. ACM.
- [VCPF08b] E. Valle, M. Cord, and S. Philipp-Foliguet. High-dimensional descriptor indexing for large multimedia databases. In *CIKM '08 : Proceeding of the 17th ACM conference on Information and knowledge management*, pages 739–748, New York, NY, USA, 2008. ACM.
- [VJ01] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Hawaii, USA*, volume 1, page 511, Los Alamitos, CA, USA, 2001. IEEE Computer Society.
- [VTC05] G. Vogiatzis, P. H. S. Torr, and R. Cipolla. Multi-view stereo via volumetric graph-cuts. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), San Diego, USA*, volume 2, pages 391–398. IEEE Computer Society, 2005.
- [VV08] P. J. Narayanan V. Vineet. CUDA cuts : Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition (CVPR) Workshop on Visual Computer Vision on GPUs, Anchorage, USA*, 2008.
- [Wac78] E. L. Wachpress. A rational finite element basis. *SIREV*, 20 :195–196, 1978.

- [WEL87] H. E. Cline W. E. Lorensen. Marching cubes : A high resolution 3D surface construction algorithm. *Computer Graphics*, 21(4), July 1987.
- [WJ96] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *ICDE '96 : Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.
- [WPL06] W. Wang, H. Pottmann, and Y. Liu. Fitting b-spline curves to point clouds by curvature-based squared distance minimization. *ACM Trans. Graph.*, 25(2) :214–238, 2006.
- [WSB98] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB, New York, USA*, pages 194–205, 24–27 1998.
- [WTK87] A. Witkin, D. Terzopoulos, and M. Kass. Signal matching through scale space. *International Journal of Computer Vision (IJCV)*, 1 :133–144, 1987.
- [XMPM98] C. Xu, Student Member, Jerry L. Prince, and Senior Member. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing*, 7 :359–369, 1998.
- [Yia93] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, Philadelphia, USA*, pages 311–321, 1993.
- [YPW03] R. Yang, M. Pollefeys, and G. Welch. Dealing with textureless regions and specular highlights-a progressive space carving scheme using a novel photo-consistency measure. In *Proceedings of the ninth IEEE International Conference on Computer Vision (ICCV), Nice, France*, volume 01, page 576. IEEE Computer Society, 2003.
- [ZBH07] A. Zaharescu, E. Boyer, and R. P. Horaud. Transformesh : a topology-adaptive mesh-based approach to surface evolution. In *Proceedings of the Eighth Asian Conference on Computer Vision (ACCV)*, volume II of *LNCS 4844*, pages 166–175, Tokyo, Japan, November 2007. Springer.
- [ZTCS99] R. Zhang, P.-S. Tsai, J. E. Cryer, and M. Shah. Shape from Shading : A Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 21(8) :690–706, August 1999.