Sujet du stage

# Reconstruction 3D et génération de textures par *graph cuts*

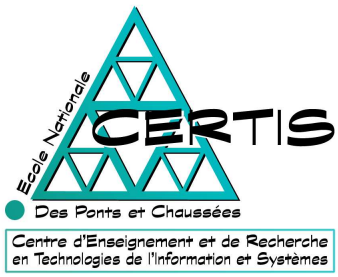
**Résumé** : les méthodes de traitement d'images utilisent traditionnellement des techniques analytiques de minimisation d'une énergie bien définie. Nous présentons ici deux applications (reconstruction 3D et génération de textures) d'une technique issue de l'optimisation combinatoire, appelée *graph cuts*.

Rapport de stage de Master

**Emmanuel Cornet**
École normale supérieure
MPRI

Sous la direction de **Renaud Keriven**

**Été 2005**




# Table des matières

**Introduction** **7**

**I Les *graph cuts*** **9**
I Couper des graphes ? . . . 11
I.1 Réseau de transport et flots . . . 11
I.2 Coupe minimale et flot maximal . . . 12
I.3 Mais où allons-nous ? . . . 13
II Chemins améliorants et *push-relabel* . . . 14
II.1 Ford-Fulkerson . . . 14
II.2 *Push-relabel* . . . 15
III Algorithme de Boykov-Kolmogorov . . . 15

**II Reconstruction 3D** **19**
I Algorithme . . . 21
II Résultats . . . 24
II.1 Test simple . . . 24
II.2 Objets non connexes et obstruction . . . 24
II.3 Robustesse . . . 25
II.4 Jeu de données réelles . . . 26
II.5 Conclusion . . . 26

**III Génération de textures** **29**
I Algorithme . . . 31
II Résultats . . . 33

**Perspectives** **37**

# Table des figures


1 Un graphe. . . . . 11
2 Un réseau de transport. . . . . 11
3 Un flot. . . . . 12
4 Coupe dans une image 2D. . . . . 14
5 Arbres de recherche de l'algorithme de Boykov-Kolmogorov. . . . . 16
6 Projection d'une image dans l'espace 3D (1). . . . . 21
7 Projection d'une image dans l'espace 3D (2). . . . . 22
8 Rectangle englobant. . . . . 22
9 Coupe dans un graphe 3D. . . . . 23
10 Cube . . . . . 24
11 Cube – reconstruction – 2 caméras . . . . . 24
12 Cube – reconstruction – 4 caméras . . . . . 25
13 Cube – reconstruction – 6 caméras . . . . . 25
14 Deux sphères . . . . . 25
15 Deux sphères – reconstruction . . . . . 25
16 Cube et texture . . . . . 26
17 Cube et texture – reconstruction . . . . . 26
18 Bouddha . . . . . 26
19 Bouddha – reconstruction . . . . . 27
20 Reproduction d'une image non auto-similaire. . . . . 31
21 Création de la nouvelle texture par *graph cuts*. . . . . 32
22 Génération de textures . . . . . 34
23 Génération de textures auto-similaires . . . . . 35

# Introduction

Traditionnellement le traitement d'images et la vision par ordinateur appliquent préférentiellement des méthodes analytiques (équations aux dérivées partielles, géométrie différentielle, etc.) à des fonctions continues. Cependant, certains algorithmes issus des mathématiques discrètes et de l'optimisation combinatoire se montrent parfaitement adaptés à plusieurs problèmes « classiques » en traitement d'images et offrent de très bons résultats.

Greig et al. (dans [GBS89][1]) ont eu, les premiers (en 1989), l'idée d'appliquer aux images des algorithmes issus de la théorie des graphes[2]. Malheureusement, cette nouveauté est passée quasiment inaperçue car Greig et al. ne l'ont utilisée que dans le cadre de la restauration d'images binaires, dont les applications restent très limitées.

Ce n'est qu'environ 10 ans plus tard que de nombreuses techniques de traitement d'images ont commencé à se servir d'algorithmes de flot maximal dans un graphe pour résoudre des problèmes plus intéressants.

En 2004, Yuri Boykov et Vladimir Kolmogorov ont conçu un nouvel algorithme de recherche du flot maximal dans un graphe, particulièrement adapté aux instances couramment rencontrées en traitement d'images ([BK04]) et dont le temps de calcul est beaucoup plus court (quasiment linéaire, en pratique).

Nous exposerons dans une première partie, après les pré-requis nécessaires, en quoi consiste ce nouvel algorithme. Puis nous montrerons comment nous avons utilisé la méthode des « *graph cuts* » pour résoudre les deux problèmes suivants :

- Reconstruction 3D : construire un modèle 3D d'un objet dont on possède des images, de plusieurs points de vue différents (algorithme original).
- Génération de textures : étant donné un échantillon de texture (image présentant déjà un caractère périodique) de taille réduite, générer une texture d'apparence naturelle, de plus grande taille, utilisant l'image de départ (raffinement d'un algorithme déjà existant).

Ce rapport est disponible au format PDF à l'adresse suivante :

`www.manucornet.net/Stage/Rapport_Cornet.pdf`

Nous avons volontairement évité d'insérer trop d'équations mathématiques dans cet exposé. Si elles sont nécessaires pour assurer des bases formelles stables aux concepts introduits, on pourra tout à fait se faire une idée relativement précise des algorithmes proposés en les « sautant » et en suivant simplement les explications textuelles.

[1] Les références entre crochets renvoient à la bibliographie, située page 39.

[2] Plus précisément, de la théorie des flots dans un graphe.

# Première partie

# Les *graph cuts*

# I Couper des graphes ?

## I.1 Réseau de transport et flots

Un **graphe orienté** G est un couple (S, A) où S est un ensemble de sommets (ou encore « nœuds ») et A est un ensemble d'arêtes (orientées par des flèches) reliant ces sommets entre eux.

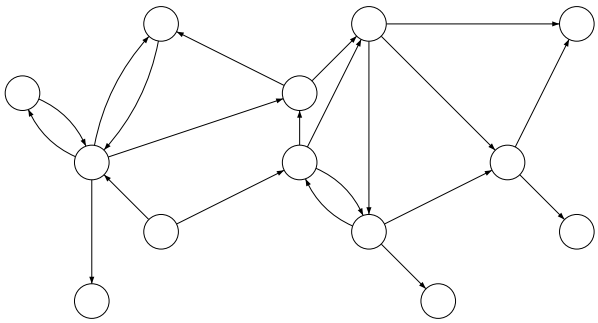


FIG. 1 – Un graphe.

Un **réseau de transport** (au sens de [CLRS]) est un graphe orienté où l'on attribue à chaque arête $(u, v) \in \mathrm{A}$ une capacité $c(u, v) \geqslant 0$. En outre, deux sommets sont distingués des autres : la source, notée $s$, et le puits, noté $t$. La figure 2 montre un exemple (repris de [CLRS]) de réseau de transport, où les nombres représentent les capacités de ces différents « tuyaux ».

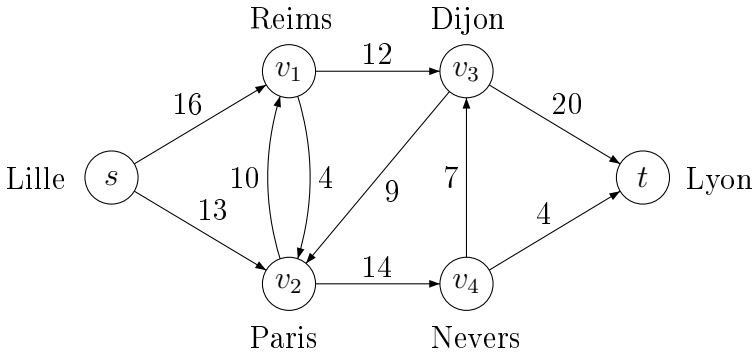



FIG. 2 – Un réseau de transport.

Un **flot** $f$ est une fonction qui associe à chaque couple de sommets une valeur réelle qui doit satisfaire trois conditions précises. On peut voir les arêtes comme des tuyaux reliant des villes (les sommets), et dont la capacité dépend du diamètre (plus il est important, plus on peut faire passer d'eau par ce tuyau chaque seconde). L'eau part de la source et arrive au puits, en traversant les tuyaux et plusieurs villes secondaires (les sommets autres que $s$ et $t$). Un flot est une façon donnée de faire passer l'eau de la source vers le puits, en décidant quel débit d'eau on choisit de faire passer par chaque tuyau. Voici donc les trois contraintes que doit satisfaire tout flot :

– **Contrainte de capacité** : on ne peut pas faire passer dans un tuyau de débit supérieur à sa capacité. Ceci s'écrit

$$\forall (u, v) \in \mathrm{S}^2 \qquad f(u, v) \leqslant c(u, v)$$

On dit qu'une arête est **saturée** lorsque le débit qui la traverse atteint exactement sa capacité.

– **Conservation du flot** : aucune ville intermédiaire (autre que la source ou le puits) ne produit ni ne consomme d'eau. Autrement dit,

$$\forall (u,v) \in (\mathrm{S} - \{s;t\})^2 \qquad \sum_{v \in \mathrm{S}} f(u,v) = 0$$

– **Symétrie** : lorsqu'on fait passer un débit $d$ de la ville A à la ville B, on considère également que le débit $-d$ circule de la ville B vers la ville A. Symboliquement,

$$\forall (u,v) \in \mathrm{S}^2 \qquad f(u,v) = -f(v,u)$$

La figure 3 montre un exemple de flot pour le réseau de transport de la figure 2. Seuls les flux positifs sont représentés (sous la forme « flot/capacité ») ; lorsque le flux est négatif, seule la capacité apparaît.

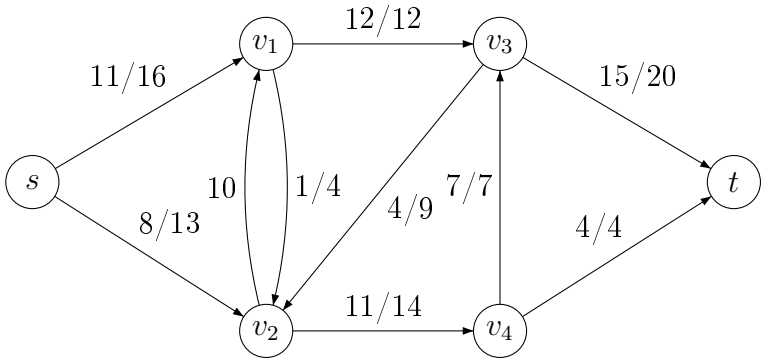



FIG. 3 – Un flot.

La **valeur** d'un flot est le flux total partant de la source ou, de façon équivalente, le flux total arrivant au puits :

$$|f| = \sum_{v \in \mathrm{S}} f(s,v) = \sum_{u \in \mathrm{S}} f(u,t)$$

Le **problème du flot maximal** consiste, étant donné un réseau de transport, à trouver un flot de valeur maximale.

Remarquons que ce type de modèle peut s'appliquer à de nombreux problèmes concrets, comme par exemple le transfert de marchandises d'une usine de production (source) vers un entrepôt (puits) grâce à un réseau de transport de topologie et de capacité données.

## I.2 Coupe minimale et flot maximal

Une autre façon de poser le problème du flot maximal est de ne plus parler de « flot », mais plutôt de « coupe ».

Le problème est le suivant : on se donne, comme ci-dessus, un graphe orienté, dont les arêtes sont étiquetées par des capacités (ou plutôt, ici, des poids), et toujours deux sommets que l'on distingue, la source $s$ et le puits $t$. On cherche à présent, non plus à faire passer de l'eau (ou des marchandises) de $s$ vers $t$, mais à couper le réseau, de façon à ce que $t$ ne soit plus accessible depuis $s$. Mais on ne veut pas couper n'importe comment : d'une part on ne peut couper que des arêtes (les tuyaux

– chaque sommet reste intègre), mais surtout couper une arête nous coûte d'autant plus cher que son poids est important (pour poursuivre sur l'image des tuyaux, ils sont d'autant plus difficiles à couper qu'ils sont gros).

Formellement, une **coupe** C peut être définie comme un sous-ensemble de l'ensemble des arêtes A tel que $t$ ne soit plus accessible à partir de $s$ dans le graphe $\mathrm{G(C)} = (\mathrm{S}, \mathrm{A} - \mathrm{C})$ induit et tel qu'aucun sous-ensemble propre $\mathrm{C}'$ de C ne vérifie cette propriété[3]. Remarquons que ce concept de coupe est « dirigé », car il n'est fait aucune hypothèse sur les chemins partant de $t$ pour atteindre $s$[4].

Le **coût** d'une coupe C est la somme des poids des arêtes qu'elle contient :

$$|\mathrm{C}| = \sum_{(u,v)\in \mathrm{C}} c(u,v)$$

> Le **problème de la coupe minimale** consiste, étant donné un réseau de transport, à trouver une coupe de coût minimal.

Le théorème de Ford et Fulkerson ([FF]) assure que le problème du flot maximal et celui de la coupe minimale sont en fait équivalents :

> L'ensemble des arêtes saturées par un flot maximal de $s$ à $t$ définit une coupe minimale C.

## I.3 Mais où allons-nous ?

Si nous n'avons pas parlé d'images ou d'espace 3D jusqu'ici, c'est que les correspondances sont assez simples à établir. De manière générale, chaque sommet du graphe correspond à un pixel (image 2D) ou à un voxel (image 3D). Les arêtes relient chaque sommet à ses plus proches voisins, dans les deux sens. Puis on peut imaginer, par exemple pour une image 2D, que la source est placée au-dessus du plan de l'image, tandis que le puits est placé en-dessous.

Les poids des arêtes sont en général calculés en fonction de la différence d'intensité lumineuse entre les sommets. En effet, lorsqu'on cherche à « couper » une image, on veut la plupart du temps le faire de la façon la plus naturelle possible, donc en suivant les contours des objets que l'on peut détecter dans l'image. Or, le contour d'un objet se distingue le plus souvent par une différence d'intensité lumineuse (zone de pixels clairs/zone de pixels plus sombres) entre l'intérieur et l'extérieur de l'objet. On affecte donc à une arête un poids d'autant plus faible que la différence entre les intensités des deux pixels qu'elle relie est importante. Ainsi, les arêtes reliant deux pixels très différents sont coupées plus facilement ; et inversement, couper au milieu d'un objet est moins probable.

Si l'on cherche à couper l'image de façon à séparer, par exemple, sa zone supérieure gauche et sa zone inférieure droite, on pourrait obtenir un graphe ressemblant à la figure 4. On choisit de relier la source et le puits aux deux zones de l'image que l'on souhaite séparer. La ligne verte en pointillés représente la coupe : toutes les arêtes (jaunes) qu'elle traverse sont celles qui seraient saturées par un flot maximal de $s$ à $t$.

[3] Nous avons choisi une définition légèrement différente de celle exposée dans [CLRS] (où une coupe est une partition de S en deux sous-ensembles) ; du point de vue des théorèmes que nous utilisons, cela ne prête pas à conséquence.

[4] Le fait que les graphes sur lesquels s'appuient tous ces concepts sont orientés est en réalité plus une facilité de notation (via la contrainte de « symétrie ») qu'une réelle nécessité intrinsèque, même si dans certaines applications en traitement d'images, la bi-directionnalité est nécessaire.

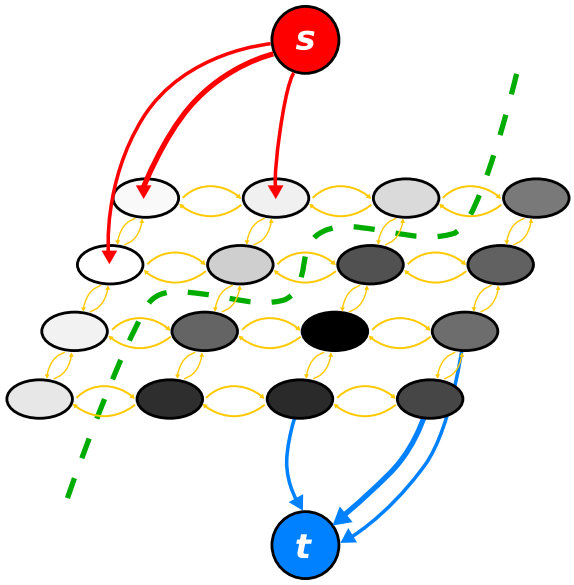



FIG. 4 – Coupe dans une image 2D.

# II Chemins améliorants et *push-relabel*

## II.1 Ford-Fulkerson

La méthode[5] « classique » de recherche d'un flot maximal est celle de Ford et Fulkerson ([FF]), basée sur la notion de chemin améliorant.

Pour un flot donné dans un réseau de transport, le **réseau résiduel** est celui qu'on obtient en ne considérant que « l'espace vide » restant dans les tuyaux (on pourrait y faire passer plus d'eau que dans la situation actuelle). Pour chaque arête (chaque tuyau), sa **capacité résiduelle** est l'« espace vide restant », c'est-à-dire la différence entre sa capacité et le flux qui la traverse :

$$c_f(u,v) = c(u,v) - f(u,v)$$

Le réseau résiduel a le même ensemble de sommets que le réseau de départ, mais ses arêtes doivent avoir une capacité résiduelle strictement positive.

On appelle alors **chemin améliorant** un chemin de $s$ vers $t$ dans le réseau résiduel. S'il existe un tel chemin, c'est qu'il existe au moins un trajet que l'on peut emprunter de $s$ à $t$ sans jamais rencontrer d'arête saturée (de tuyau plein). On peut donc augmenter le flux le long de ce chemin, jusqu'à ce que l'une de ses arêtes soit saturée (un tuyau a atteint son débit maximum, on ne peut pas fournir plus d'eau selon ce chemin). Puis on cherche un nouveau chemin améliorant, et ainsi de suite.

La méthode peut donc se présenter ainsi :

FORD-FULKERSON $(G, s, t)$

Initialiser le flot $f$ à 0
**Tant qu'**il existe un chemin améliorant $p$ de $s$ à $t$ dans le réseau résiduel
    Chercher le minimum $c$ des capacités résiduelles le long de $p$
    Ajouter $c$ au flux le long de $p$

[5] Tout comme [CLRS], nous utilisons le terme « méthode » plutôt qu'« algorithme », car elle comporte plusieurs implémentations de temps d'exécution différents.

En particulier, l'agorithme de Dinic (présenté dans [Din70]) utilise cette méthode. Il utilise une recherche en largeur pour chercher les chemins améliorants les plus courts dans le réseau résiduel. Une fois que tous les chemins améliorants de longueur $k$ ont été traités, la recherche en largeur reprend de zéro (ceci sera, justement, amélioré dans l'algorithme présenté en section III) pour les chemins de longueur $k+1$. Cette recherche de chemins améliorants les plus courts possible améliore la complexité théorique de cet algorithme, qui est en $O(m\,n^2)$, où $n$ est le nombre de sommets, et $m$ le nombre d'arêtes du graphe.

Pour plus de détails, d'exemples, et d'algorithmes détaillés sur la méthode de Ford-Fulkerson, nous conseillons au lecteur curieux la lecture de [CLRS] (pages 625 – 644) ou de [CCPS].

### II.2 *Push-relabel*

Nous passerons plus rapidement sur le principe des algorithme dits *push-relabel* (que nous mentionnons par souci d'exhaustivité), car nous ne nous en servirons pas par la suite.

Les algorithmes *push-relabel* (dont le principe est introduit dans [GT88]) ne gardent pas à jour une version valide du flot (*ie* qui satisfait les trois contraintes évoquées page 11). Au contraire, l'algorithme progresse de la source vers le puits, et des « nœuds actifs » (situés sur le « front », plus près du puits) portent un certain « excès de flot ». L'algorithme met également à jour un étiquetage des sommets qui donne une estimation, pour chacun d'entre eux, de la borne inférieure de la distance qui les sépare du puits, selon un chemin non saturé. L'algorithme tente de « pousser » (*push*) l'excès de flot vers le puits selon les arêtes non saturées. À la fin, les flots qui n'ont pas pu être acheminés jusqu'au puits sont ramenés vers la source.

Pour une description plus complète et détaillée de cette méthode, on pourra se reporter à [CCPS].

## III Algorithme de Boykov-Kolmogorov

Yuri Boykov et Vladimir Kolmogorov ont conçu un nouvel algorithme, exposé dans [BK04]. Nous avons vu que l'algorithme de Dinic nécessitait de reprendre à zéro la recherche en largeur de chemins améliorants dès que la recherche de chemins d'une longueur donnée était terminée. Or, en traitement d'images, construire un arbre de recherche en largeur implique en général de passer en revue la majorité des pixels (ou des voxels). C'est donc, en pratique, une opération très coûteuse si elle est répétée trop souvent ; et les algorithmes correspondants ne sont donc (expérimentalement, voir [BK04]) pas performants sur les graphes habituellement rencontrés en traitement d'images.

L'algorithme de Boykov-Kolmogorov procède de façon similaire à celui de Dinic (et utilise donc la méthode des chemins améliorants), mais il construit deux arbres de recherche, l'un partant de la source, l'autre partant du puits. Ces arbres sont réutilisés tout au long de l'algorithme et ne sont jamais reconstruits de zéro.

Les deux arbres (notés S et T) sont disjoints ; S et T ont pour racine $s$ et $t$, respectivement. Dans S, toutes les arêtes reliant un nœud père à l'un de ses fils doivent être non saturées. Inversement, dans T, toute arête reliant un fils à son père doit être non saturée. Tous les nœuds qui ne sont ni dans S ni dans T sont « libres ».

Les nœuds de S et T sont soit « actifs » soit « passifs ». Les nœuds actifs représentent en fait la frontière extérieure de l'arbre ; ils sont capables d'étendre l'arbre en adoptant de nouveaux nœuds, tandis que les nœuds passifs ne bougent plus, bloqués par les autres nœuds du même arbre. On a trouvé un chemin améliorant dès qu'un nœud passif de l'un des arbres devient voisin d'un nœud de l'autre arbre.

Au départ, les deux arbres S et T sont réduits à leurs racines $s$ et $t$ (qui sont des nœuds actifs). L'aglorithme itère ensuite trois étapes distinctes :

- **Croissance** : S et T croissent jusqu'à ce qu'ils se touchent, produisant ainsi un chemin améliorant.
- **Augmentation** : le chemin améliorant est augmenté ; les deux arbres sont « cassés » en forêts.
- **Adoption** : S et T se reconstituent.

Pendant la phase de croissance, les nœuds actifs explorent les nœuds voisins et prennent possession des nœuds encore libres et reliés à eux par une arête non saturée. Les nœuds nouvellement acquis deviennent de nouveaux nœuds actifs de l'arbre correspondant. Lorsque tous les voisins d'un nœud actif ont été explorés, le nœud devient passif. La phase de croissance s'arrête lorsqu'un nœud actif devient voisin (via une arête non saturée) d'un nœud de l'arbre opposé. On a alors trouvé un chemin de $s$ vers $t$ (voir la figure 5).

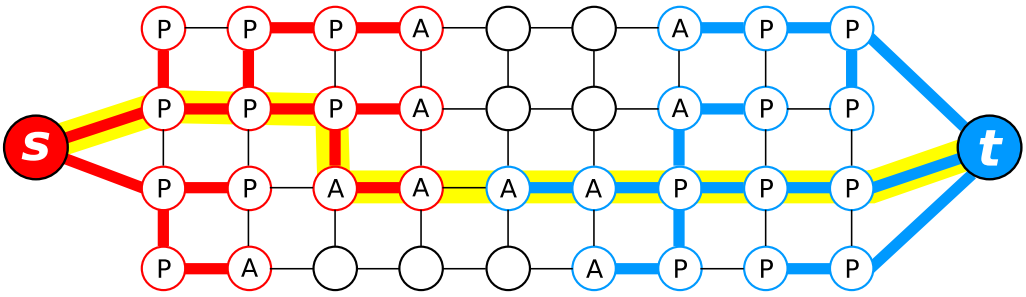



FIG. 5 – Arbres de recherche de l'algorithme de Boykov-Kolmogorov.

La phase d'augmentation, comme dans la méthode de Ford-Fulkerson, sature le chemin améliorant trouvé dans la phase précédente. Certaines arêtes deviennent donc saturées, ce qui implique (voir les règles ci-dessus) que des nœuds deviennent orphelins : les arêtes qui les reliaient à leur parents ne sont plus valides (puisqu'elles sont saturées). Les deux arbres se séparent donc en plusieurs forêts, ayant pour racines $s$, $t$, et les différents nœuds orphelins.

La phase d'adoption vise à reconstituer les arbres S et T (toujours avec les racines respectives $s$ et $t$). On cherche donc à donner un nouveau parent valide (via une arête non saturée) à chaque orphelin, appartenant au même arbre que lui. Si un orphelin ne trouve aucun parent, il devient un nœud libre, et tous ses fils deviennent orphelins. L'adoption se termine lorsqu'il ne reste plus d'orphelins (la structure d'arbre de S et de T est donc restaurée). Puis, l'algorithme repasse en phase de croissance.

Le processus se termine lorsque les arbres ne peuvent plus croître (pas de nœuds actifs) et que les arbres sont séparés par des arêtes saturées : on a atteint un flot maximal (et donc une coupe minimale).

Le défaut de cet algorithme, en comparaison avec celui de Dinic, est que les chemins améliorants ne sont pas nécessairement de longueur minimale. Sa complexité

théorique ($O(m\,n^2\,|C|)$, où $|C|$ est le coût de la coupe minimale) est donc supérieure ; cependant, en pratique, sur les instances traitées en vision, l'aglorithme de Boykov-Kolmogorov est bien meilleur que celui de Dinic.

# Deuxième partie

# Reconstruction 3D

# I Algorithme

Dans cette partie (qui a occupé la majeure partie du stage, en durée) on se pose le problème suivant : étant donné au moins deux prises de vue d'un même objet et les paramètres des caméras correspondantes, reconstruire un modèle 3D de cet objet.

On travaille ici sur des graphes 3D, de forme cubique. On pourra simplement se représenter un grand cube, quadrillé de façon régulière en de très nombreux petits cubes. Il s'agit donc d'un espace discret : même si la forme de l'objet à reconstituer est le plus souvent continue, on ne pourra avec cette méthode en reconstruire qu'une forme discrétisée, comme si l'on cherchait à imiter la forme de départ en agençant et en empilant correctement de petits cubes (à la manière des « Lego »). On pourra cependant diminuer petit à petit le côté des cubes élémentaires pour obtenir une plus grande précision (dans la limite des ressources matérielles de la machine de calcul).

L'algorithme que nous proposons va « éclairer » l'espace 3D à travers les images 2D fournies. La projection d'une seule de ces images dans l'espace 3D (figure 6, un cube blanc sur fond noir) produit un cône de lumière ; c'est à partir de 2 images que, en observant l'intersection de cônes, on peut commencer à reconstruire un volume (figure 7). Ici, le cube original est déformé car, avec seulement deux faisceaux de lumière, il ne peut s'agir que de l'intersection de deux cônes de section carrée ; c'est en rajoutant des angles de vue (par exemple, une vue « de dos ») que l'on pourra reconstruire la forme originale du cube.

Les « faisceaux lumineux » sont envoyés par les parties des images qui comportent de forts « contrastes » (différences d'intensité lumineuse), ici les bords du cube. La méthode des *graph cuts* est utilisée pour retrouver la forme de l'objet 3D à partir des traces laissées par ces « faisceaux lumineux » dans le graphe.

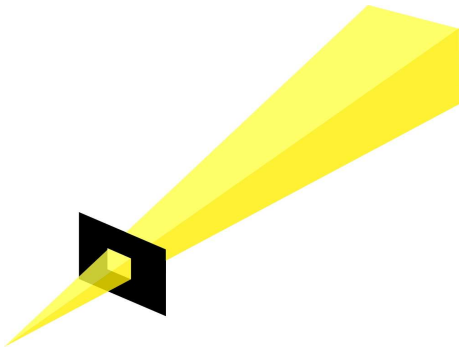


FIG. 6 – Projection d'une image dans l'espace 3D (1).

En réalité, c'est dans l'autre sens que notre algorithme effectue la projection : depuis l'espace 3D vers chacune des images 2D ; mais le principe reste le même. Une partie du travail a consisté à implémenter les formules matricielles de projection d'une scène sur un plan focal, étant donné les paramètres de la caméra (notre algorithme sait traiter des paramètres de caméra donnés sous forme « naturelle » – position, direction du regard, focale, direction de la verticale – où sous la forme d'une matrice $3 \times 4$ en coordonnées projectives). Cependant, nous ne détaillerons pas ici les fastidieuses équations matricielles de projection et les paramètres des caméras : si elles sont indispensables à l'implémentation de l'algorithme, elles comportent

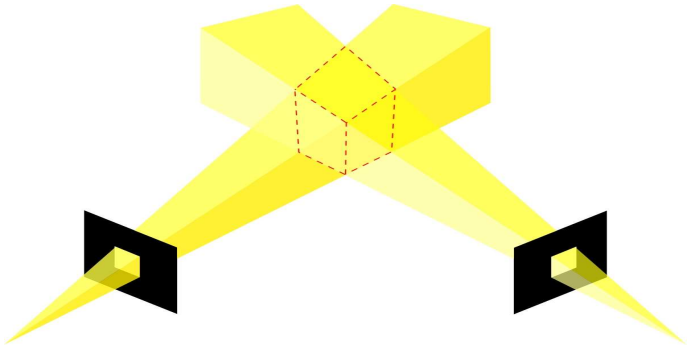


FIG. 7 – Projection d'une image dans l'espace 3D (2).

(de notre point de vue !) peut d'intérêt, sur les plans mathématique comme algorithmique (même si elles sont la base des techniques classiques de « stéréovision », le but de notre travail était justement d'en prendre de contre-pied). Elles alourdiraient en outre ce rapport de plusieurs pages, de lecture passablement rébarbative. Nous conseillons donc au lecteur friand de coordonnées homogènes, de matrices de projection et de transformations de l'espace de consulter [HM].

Voici comment l'algorithme procède :

RECONSTRUCTION3D

Initialiser toutes les arêtes du graphe 3D à 0
**Pour** chaque cube élémentaire C du graphe
    **Pour** chaque image 2D I
        Projeter C sur I
        Trouver R le rectangle englobant (fig. 8) la projection de C sur I
        Intégrer le gradient de l'intensité lumineuse sur R
        Ajouter le résultat au poids de chacune des 12 arêtes de C
Appliquer une fonction de la forme $x \mapsto \dfrac{k}{(1 + k' x^2)}$ au poids de chaque arête
Relier les six faces extérieures du graphe au puits (fig. 9)
Relier à la source un sommet situé au centre de l'objet à reconstituer
Couper le graphe avec l'algorithme de Boykov-Kolmogorov

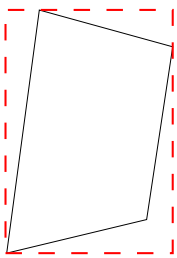


FIG. 8 – Rectangle englobant.

Sur la figure 9, le puits (relié aux faces externes du graphes) est représenté en rouge, la source (reliée à un unique sommet, placé au centre de l'objet à reconstituer) figure en bleu, et l'objet reconstruit (un simple cube), en vert.

Le code de notre implémentation est entièrement disponible à l'adresse

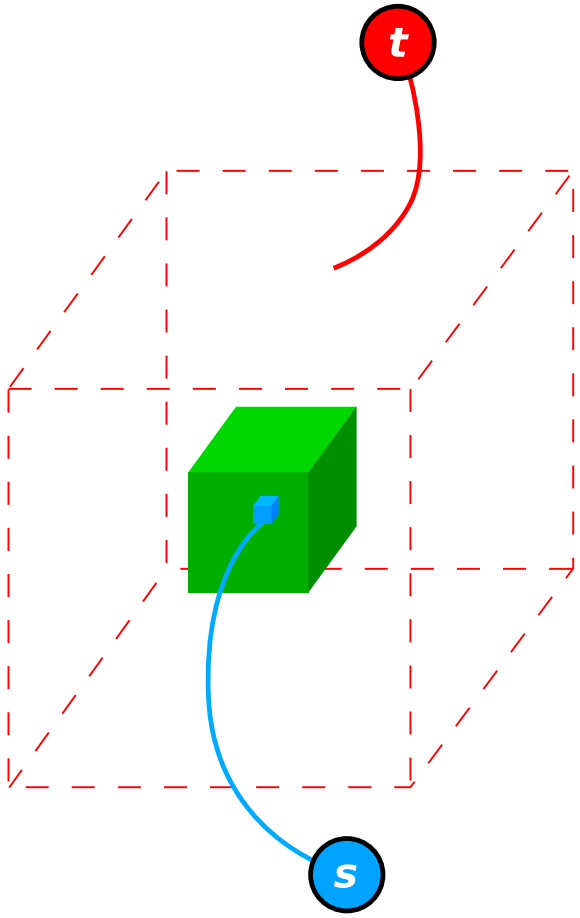



FIG. 9 – Coupe dans un graphe 3D.

`www.manucornet.net/Stage/Code/Beams_3D`

et nous utilisons une implémentation de Yuri Boykov pour le calcul de la coupe minimale.

D'autres travaux utilisant les *graph cuts* pour reconstruire des objets 3D ont déjà, ou sont en train d'être effectués. Notre approche se démarque sur plusieurs aspects :

- Toutes les informations de gradient contenues dans chaque image sont utilisées et projetées dans le graphe : les images ne sont pas préalablement « segmentées » (transformées en images binaires séparant le premier plan du fond), comme dans [SVZ00] par exemple.
- La phase de *graph cuts* est unique, finale et définitive ; en outre, elle prend en compte immédiatement l'ensemble de l'espace 3D considéré. D'autres méthodes utilisent les *graph cuts* de façon itérative et locale, comme celle exposée dans [ZPQ05].
- Nous utilisons exclusivement les *graph cuts* comme méthode d'optimisation, tandis que les *graph cuts* sont parfois utilisés de concert avec d'autres techniques comme les contours actifs ([XBA03]) ou le *carving* ([ZPQ05]).

# II Résultats

Voici quelques résultats, présentés par ordre croissant de complexité, obtenus grâce à l'algorithme présenté dans la section précédente.

Toutes les images de base, ainsi que les résultats (sous forme de films QuickTime – à visionner en « boucle ») sont disponibles à cette adresse :

`www.manucornet.net/Stage/Resultats/3D`

Tous les objets reconstruits ont une couleur uniforme et ne sont donc pas coloriés en fonction des données, car nous nous sommes concentrés sur le problème de reconstruction de la forme d'un objet. Le problème consistant à colorier un modèle 3D déjà donné en fonction des couleurs observées sur les images se résout facilement. Signalons toutefois que certains algorithmes s'aident des informations de couleur contenues dans les images pour construire le modèle 3D (ce n'est pas notre cas).

## II.1 Test simple

Le premier jeu de données (images synthétiques) est extrêmement simple : un cube blanc sur fond noir, vu perpendiculairement à chacune de ses faces (les sources de lumière virtuelles sont placées de telle sorte que l'éclairage n'est pas partout le même – et la vue n'est pas toujours parfaitement perpendiculaire – mais tout cela ne doit pas perturber la reconstruction, puisque les paramètres des caméras sont connus).

FIG. 10 – Cube

Pour ce premier exemple simple, nous montrons trois résultats de reconstruction, après prise en compte de seulement 2 images,

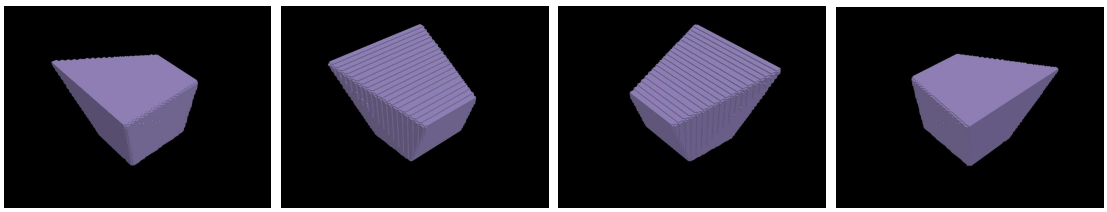


FIG. 11 – Cube – reconstruction – 2 caméras

de 4 images,
et de 6 images :

## II.2 Objets non connexes et obstruction

Le problème à l'origine de beaucoup de difficultés en reconstruction 3D est l'obstruction : un objet (ou une partie d'un objet) peut en cacher un autre. Dans ce cas,

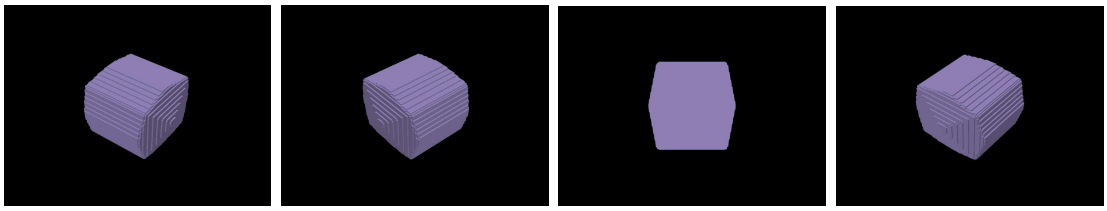


FIG. 12 – Cube – reconstruction – 4 caméras

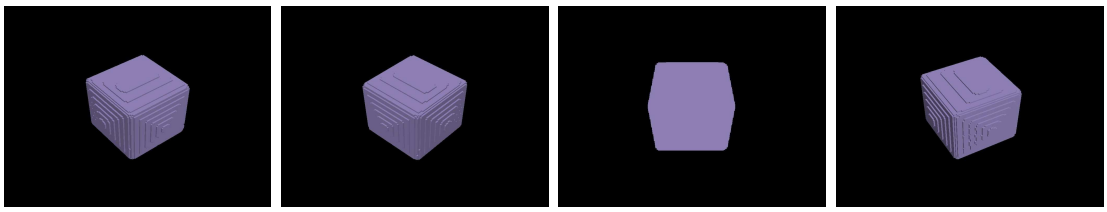


FIG. 13 – Cube – reconstruction – 6 caméras

quelle image prendre en compte pour reconstruire quel objet ? Comment savoir, dans une image donnée, quelle partie d'un objet est cachée puisque, justement, on ne sait pas encore quelle est sa position exacte par rapport au reste de l'environnement ?

Nous avons donc testé, sur un exemple simple, la capacité de l'algorithme à reconstruire deux objets distincts, l'un caché par l'autre. Pour ces données, nous avons donc relié deux voxels à la source (un au centre de chaque sphère).

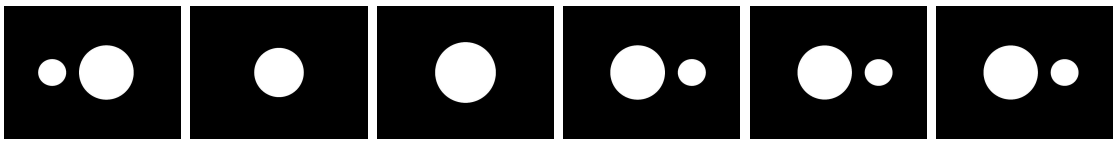


FIG. 14 – Deux sphères

Le fait que les informations contenues dans chaque image sont projetées dans l'espace 3D et affectent l'ensemble du graphe permet de résoudre le problème simplement, sans avoir besoin de faire de corrélation entre les différentes images :

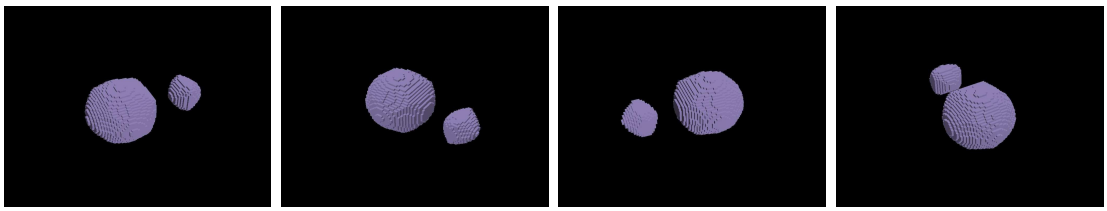


FIG. 15 – Deux sphères – reconstruction

## II.3 Robustesse

Un autre problème souvent ennuyeux en reconstruction vient du fait que le fond des photos introduit des perturbations. Comment le programme sait-il quels sont les

objets que nous souhaitons reconstruire et ceux qui ne nous intéressent pas ? Pour voir si l'algorithme était capable de reconstruire un objet dans des « conditions difficiles », nous avons ajouté un fond très contrasté derrière les images du cube blanc.

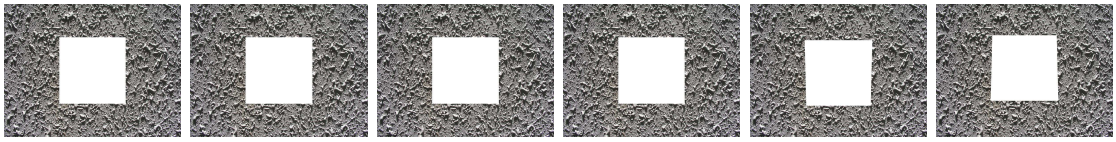


FIG. 16 – Cube et texture

Le caractère additif des « rayons » envoyés dans le graphe pour ajuster le poids des arêtes permet là encore de résoudre le problème de façon très simple : l'algorithme ne coupe qu'aux endroits où le plus de « rayons lumineux » sont passés, et laisse de côté les autres.

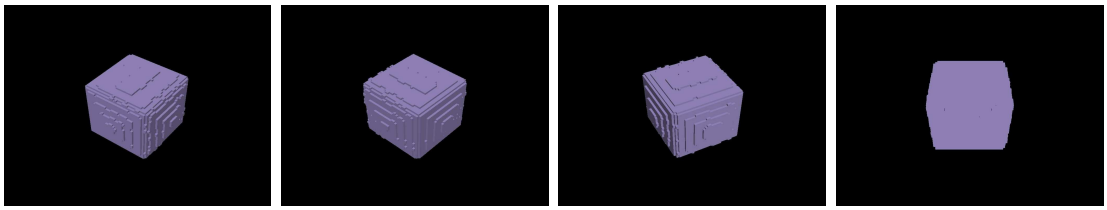


FIG. 17 – Cube et texture – reconstruction

## II.4 Jeu de données réelles

Nous avons enfin testé l'algorithme sur des données réelles. Voici ce que nous avons obtenu pour cette statue de Bouddha, avec seulement 4 prises de vue.

FIG. 18 – Bouddha

## II.5 Conclusion

Notre algorithme n'a pas la prétention de concurrencer les meilleurs algorithmes de reconstruction 3D, basés sur des méthodes analytiques bien connues et long-temps raffinées. Ces derniers donnent aujourd'hui d'excellents résultats du point de vue de la précision des modèles 3D reconstruits. Notre méthode a plutôt le mérite d'être originale, et de présenter des caractéristiques différentes des algorithmes plus classiques :

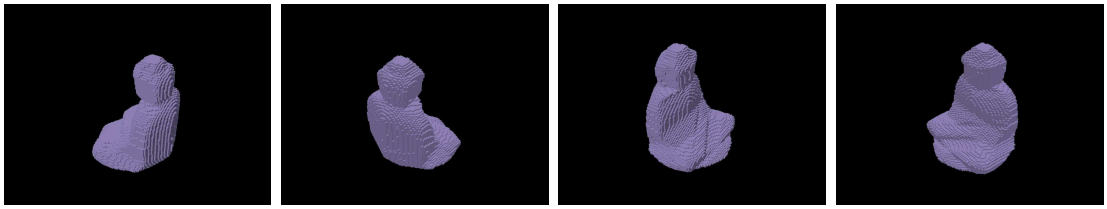


FIG. 19 – Bouddha – reconstruction

- Notre algorithme est extrêmement rapide sur une machine relativement récente[6] : aucun des résultats présentés n'a pris plus d'une minute à calculer (toutes étapes comprises, à partir des photos 2D et des données de caméra). Même si les meilleurs algorithmes de reconstruction 3D parviennent à des modélisations bien plus précises, leur temps de calcul est plutôt de l'ordre de plusieurs heures (ou d'au moins 30 minutes).
- Le revers de la médaille, c'est que la faible complexité en temps est compensée par une grande complexité en espace, puisque l'on doit stocker une valeur de poids d'arête pour chaque nœud d'un graphe cubique de côté $a$ (donc $a^3$ valeurs). Nous disposions déjà d'une bonne implémentation de l'algorithme des *graph cuts*, mais peu d'accès sur les structures manipulées en interne : nous devions donc garder une copie des données pour calculer le poids des arêtes, avant de transférer toutes ces données vers le programme de coupe. En intégrant ce dernier dans notre programme, on pourrait donc gagner un facteur 2 pour la mémoire utilisée. La précision maximale des résultats est directement dépendante de la quantité de mémoire de la machine de calcul.
- On peut constater que les résultats obtenus sont plausibles à partir de seulement 2 images, et que 4 images donnent déjà une modélisation du profil de l'objet. Les algorithmes utilisant des méthodes analytiques ont besoin de plus d'images (nous avons d'ailleurs extrait les 4 images du bouddha dans une base de données qui en comprenait 25).
- La nature même de l'algorithme fait qu'il est robuste vis-à-vis des perturbations introduites par le fond des images et aux obstructions éventuelles.

[6]Processeur à 2 GHz.

# Troisième partie

# Génération de textures

Le problème posé ici est de construire une large texture à partir d'une image réduite donnant un échantillon de cette texture. Certaines images sont spécialement conçues pour pouvoir être dupliquées et collées côte à côte tout en préservant une continuité (nous les appellerons « auto-similaires »), mais dans le cas général cette procédure simple ne donne bien sûr pas de bons résultats :

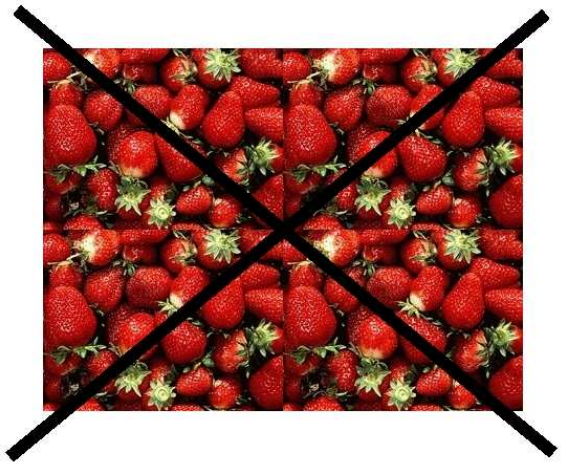


FIG. 20 – Reproduction d'une image non auto-similaire.

La génération de textures vise donc à créer une image de grande taille, en conservant l'aspect naturel de la texture. Notre algorithme permet également de créer, justement, de grandes textures auto-similaires.

# I Algorithme

Contrairement à la reconstruction 3D, nous ne proposons pas ici un algorithme original, mais une version améliorée d'un algorithme existant, ainsi que son implémentation sous la forme d'un *plugin* pour THE GIMP[7], nommé Texturize (sous licence libre – GPL – lui-même), actuellement en version 2.0[8]. Il s'agit en fait d'un travail déjà commencé en janvier 2005, en collaboration avec Jean-Baptiste Rouquier, qui a été amélioré dans le cadre du stage (en particulier, le code de la partie gérant les graphes a été totalement réécrit).

La base de l'algorithme utilisé est décrite dans [KSE03]. Notre implémentation permettait déjà d'utiliser cette base en pratique dans un programme très efficace :

`www.manucornet.net/Texturize.php`

Les améliorations effectuées dans la version 2.0 permettent de créer des textures « auto-similaires » (*tileable*), afin de pouvoir les disposer en mosaïque et garder une texture naturelle[9].

Nous appellerons « image » le résultat du calcul, c'est-à-dire la texture de grande taille, et « patch » l'image de départ, qui va être collée à plusieurs reprises sur l'image.

[7] *GNU Image Manipulation Program*, un programme de retouche d'images, sous licence libre (GPL), disponible pour Windows, Mac OS X et Linux : `www.gimp.org` .

[8] Le paquet correspondant est d'ailleurs inclus dans la distribution Debian stable officielle.

[9] Il s'agissait de la fonction la plus demandée par les utilisateurs de la première version de Texturize.

À chaque fois que l'on veut « coller » un patch sur l'image (voir la figure 21), on passe par deux étapes :

- On cherche **où** exactement placer le patch dans une zone déterminée, en fonction des pixels déjà existants dans l'image (simple calcul de similarité entre les pixels).
- Une fois la position de placement déterminée, on détermine quels pixels du patch doivent être effectivement « collés » sur l'image, en déterminant une coupe minimale entre la source et le puits : même si cette étape varie si l'on veut obtenir une texture auto-similaire ou non, grosso-modo (voir figure 21, où les parties déjà remplies de l'image figurent en gris) on relie à la source les pixels situés au bord du patch et situés « au-dessus » d'un pixel rempli de l'image, et on relie au puits les pixels du patch eux-mêmes situés au-dessus d'un pixel rempli, et dont l'un des voisins est au-dessus d'un pixel non rempli de l'image.

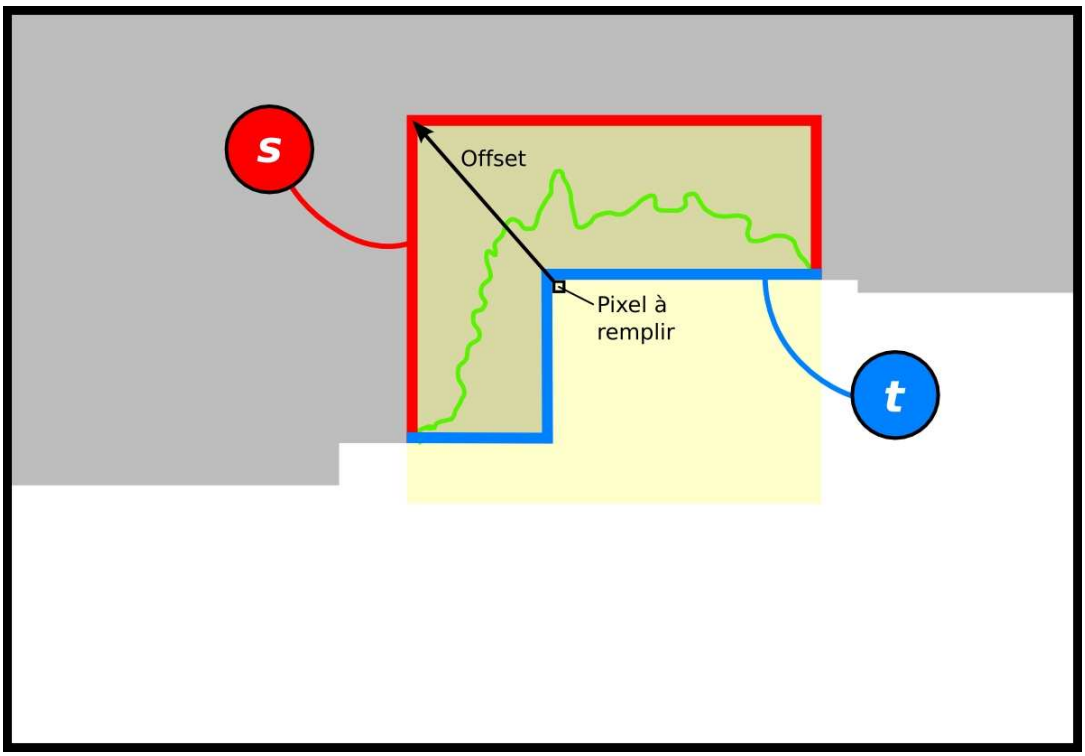



FIG. 21 – Création de la nouvelle texture par *graph cuts*.

Le poids de l'arête entre deux pixels $s$ et $t$ dépend de la valeur de ces pixels, à la fois dans le patch P et dans l'image I :

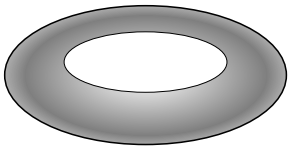
$$c(s,t) = \|\mathrm{P}(s) - \mathrm{I}(s)\| + \|\mathrm{P}(t) - \mathrm{I}(t)\|$$

Autrement dit, si les deux pixels $s$ et $t$ contenus dans le patch à coller sont très similaires à ceux qui sont « en-dessous » d'eux, dans l'image, alors couper entre ces deux pixels n'introduira pas de grande discontinuité : on affecte un poids faible à l'arête qui les relie pour qu'elle puisse être coupée facilement si nécessaire. En revanche, si l'on se trouve dans une zone très différente entre le patch et l'image, ce n'est pas le moment de couper : on verrait de nettes discontinuités apparaître ; on affecte donc un poids important à l'arête correspondante.

Notre implémentation de l'algorithme fonctionne donc de la façon suivante :

TEXTURIZE

Coller un patch en $(0, 0)$ de l'image
Déterminer le prochain pixel vide $p$, dans l'ordre lexicographique $(y, x)$
**Tant que** $p$ existe
    Déterminer l'offset optimal (placement du patch)
    Déterminer les pixels à inclure dans l'image (coupe minimale)
    Déterminer le prochain pixel vide $p$, dans l'ordre lexicographique $(y, x)$



Cette procédure (et en particulier la façon de relier les pixels à la source et au puits) a été conçue pour fonctionner lorsque, dans le cadre de la création d'une texture auto-similaire, on ne travaille plus sur une image (zone rectangulaire plane) mais sur un tore ($\mathbb{Z}^2$ est ramené dans la zone de l'image par modulo), en particulier lorsqu'un patch ne recouvre pas des zones remplies de l'image nécessairement connexes.

L'ensemble du code de notre programme est disponible sur SourceForge, à l'URL

`http ://gimp-texturize.sf.net`

# II Résultats

De nombreux exemples de textures produites grâce à cet algorithme sont disponibles sur la page

`http ://www.manucornet.net/Texturize.php`

En voici quelques exemples, d'abord non auto-similaires (qui existaient donc avant le stage) ; l'image de base apparaît à gauche, la texture générée, à droite :

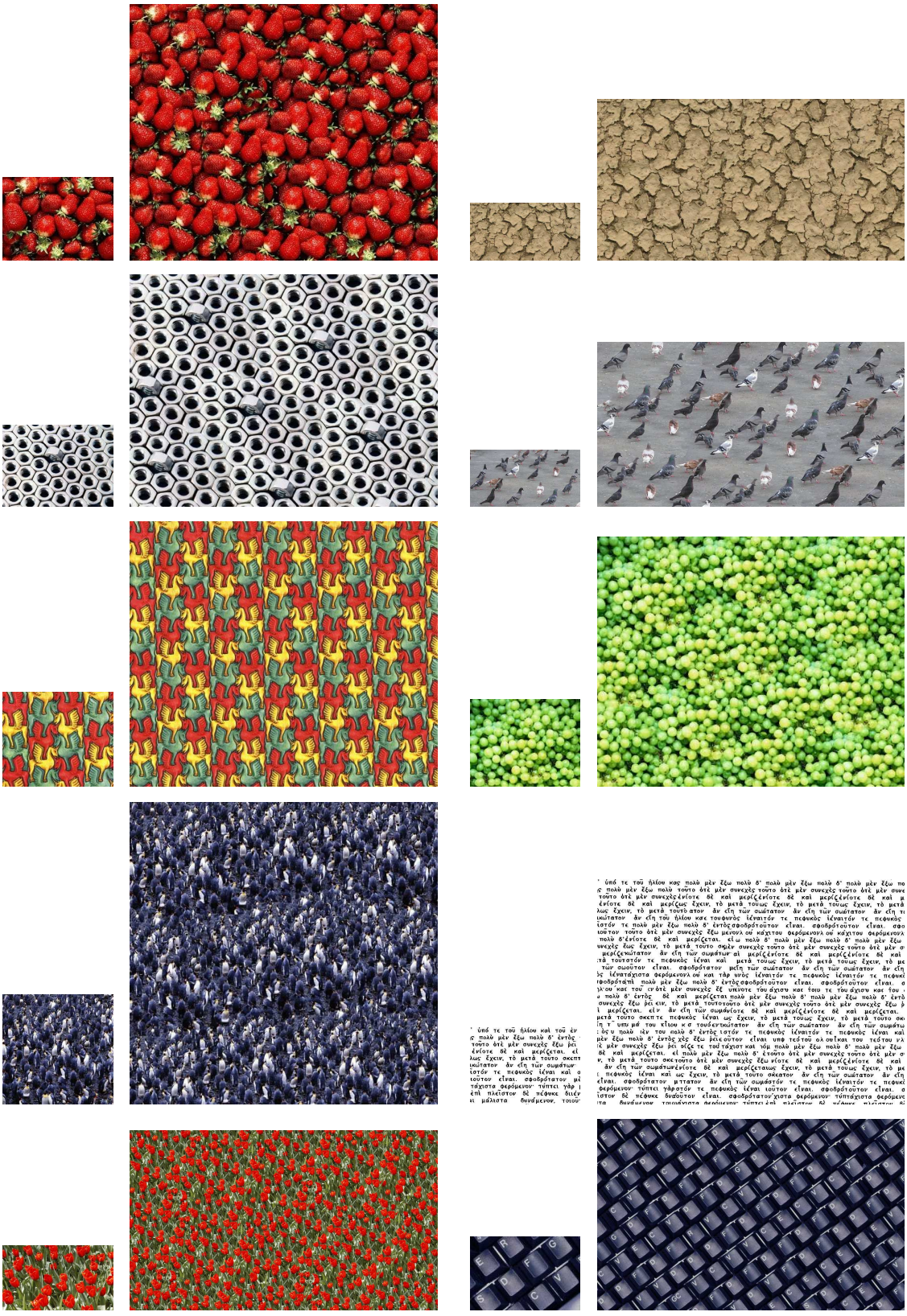


FIG. 22 – Génération de textures

Voici à présent trois exemples de génération des textures auto-similaires. De gauche à droite, figurent l'image de base, la texture générée (auto-similaire), et le résultat obtenu en juxtaposant six copies de cette texture (l'échelle n'est pas toujours respectée, pour les besoins de la mise en page).

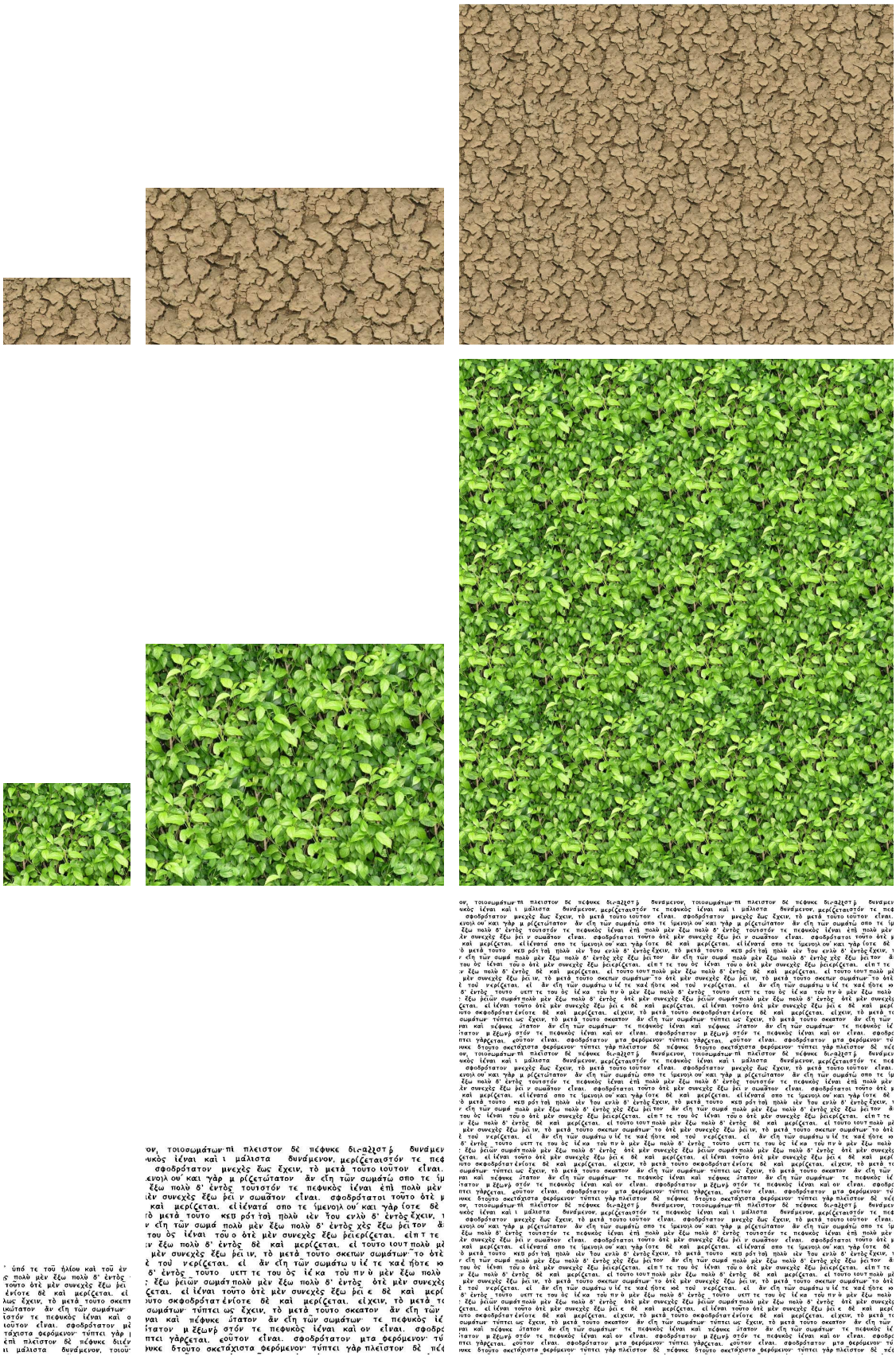


FIG. 23 – Génération de textures auto-similaires

# Perspectives

Nous avons donc proposé deux méthodes (l'une originale, l'autre étant l'amélioration d'un algorithme récent) appliquant l'algorithme de coupe dans un graphe de Boykov-Kolmogorov à des problèmes de traitement d'images. Voici quelques idées d'améliorations pour les programmes présentés :

- Nous avons utilisé une implémentation du calcul des *graph cuts* de façon externe depuis notre programme, sans y apporter de modifications. En intégrant dans notre programme et/ou en réécrivant cette bibliothèque, on pourrait réduire fortement la mémoire utilisée (et donc augmenter la précision de la reconstruction pour une machine donnée).
- Notre algorithme de reconstruction 3D impose de fixer la position de la source (n'importe où à l'intérieur de chaque objet à reconstruire) manuellement. Une amélioration de la simplicité d'utilisation présenterait à l'utilisateur deux des images fournies et lui demanderait de pointer une zone située à l'intérieur de chacun des objets à reconstituer, pour chaque image. La position de la (ou des) source(s) serait ensuite simplement déterminée par intersection, connaissant les paramètres des caméras correspondantes.
- Notre programme de génération de textures ne sait pas bien gérer les images de base dont la luminosité n'est pas uniforme (la page web montre quelques exemples). L'un des raffinements à apporter pour la prochaine version sera donc de filtrer, avant tout traitement, les composantes de basse fréquence de le variation d'intensité lumineuse de l'image.
- La génération de textures ne prend pas encore en compte l'étape de « raffinement » décrite dans [KSE03]. Il faudrait garder, entre deux coupes, le souvenir des traces (« cicatrices ») des anciennes coupes et les recouvrir avec de nouveaux patches. Ceci suppose donc de conserver le graphe après la coupe, ce que l'implémentation des *graph cuts* que nous avons utilisée ne sait pas encore faire.

Un prolongement intéressant à ce travail serait de se pencher plus précisément sur l'algorithme de *graph cuts* même. Les chercheurs en vision/traitement d'images utilisent de plus en plus la capacité des cartes graphiques récentes à pouvoir être programmées pour s'en servir comme unités de calcul extrêmement puissantes (mais dont l'utilisation impose de nombreuses contraintes). Le calcul d'une coupe sur carte graphique nécessiterait, pour un temps d'exécution optimal, que le calcul puisse être effectué de façon parallèle (c'est ainsi que fonctionnent les unités de calcul des cartes graphiques). Ceci suppose de repenser totalement l'algorithme de *graph cuts* et ses enjeux.

# Bibliographie


[BK04] Yuri Boykov et Vladimir Kolmogorov, *An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 26(9) : 1124–1137, septembre 2004.

[BVZ01] Yuri Boykov, Olga Veksler et Ramin Zabih, *Fast approximate energy minimization via graph cuts*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 23(11) : 1222 – 1239, novembre 2001.

[CCPS] William J. Cook, William H. Cunningham, William R. Pulleyblank et Alexander Schrijver, *Combinatorial Optimization*, John Wiley & Sons, 1998.

[CLRS] Thomas Cormen, Charles Leiserson, Ronald Rivest et Clifford Stein, *Introduction à l'algorithmique*, 2e édition, Dunod, 2002.

[Din70] E.A. Dinic, *Algorithm for solution of a problem of maximum flow in networks with power estimation*, Soviet Math. Dokl., 11 : 1277–1280, 1970.

[FF] L. Ford et D. Fulkerson, *Flows in Networks*, Princeton University Press, 1962.

[GT88] Andrew V. Goldberg et Robert E. Tarjan, *A new approach to the maximum-flow problem*, Journal of the Association for Computing Machinery, 35(4) : 921–940, octobre 1988.

[GBS89] D. Greig, B. Porteous, A. Seheult et *Exact maximum a posteriori estimation for binary images*, Journal of the Royal Statistical Society, Series B, 51(2) : 271–279, 1989.

[HM] Radu Horaud et Olivier Monga, *Vision par ordinateur*, Hermès, 1995.

[Kar00] David R. Karger, *Minimum Cuts in Near-Linear Time*, Journal of the Association for Computing Machinery, 47(1) : 46–76, 2000.

[KS00] Kiriakos N. Kutulakos et Steven M. Seitz, *A Theory of Shape by Space Carving*, The International Journal of Computer Vision, 38(3) : 199–218, juillet 2000.

[KSE03] Viviek Kwatra, Arno Schödl, Irfan Essa, Greg Turk et Aaron Bobick, *Graphcut Textures : Image and Video Synthesis using Graph Cuts*, Proc. ACM SIGGRAPH, 2003.

[PKF04] Jean-Philippe Pons, Renaud Keriven et Olivier Faugeras, *Modelling Dynamic Scenes by Registrating Multi-View Image Sequences*, Rapport de recherche INRIA n°5321, septembre 2004.

[SD99] Steven M. Seitz et Charles R. Dyer, *Photorealistic Scene Reconstruction by Voxel Coloring*, International Journal of Computer Vision, 35(2) : 151–173, 1999.

[SVZ00] Dan Snow, Paul Viola et Ramin Zabih, *Exact voxel occupancy with Graph Cuts*, Proceedings of International Conference on Computer Vision and Pattern Recognition, vol. 3 : 345 – 352, 2000.

[XBA03] , Ning Xu, Ravi Bansal et Narendra Ahuja, *Object Segmentation Using Graph Cuts Based Active Contours*, Conference on Computer Vision and Pattern Recognition, II : 46–53, 2003.

[ZPQ05] Gang Zeng, Sylvain Paris, Long Quan et François Sillion, *Progressive Surface Reconstruction from Images Using a Local Prior*, International Conference on Computer Vision, 2005.