

Montage photo par inpainting

Arthur Bresnu¹, Michaël Laporte², Marius Michetti³, Simon Blotias⁴

Résumé

Ce papier fait partie d'une démarche d'initiation à la recherche à l'École nationale des ponts et chaussées. Notre projet dans le cadre de cette démarche est le suivant : reproduire au mieux possible, à partir d'un article de recherche existant, l'algorithme qui y est décrit. Dans notre cas l'article en question est le suivant : Region Filling and Object Removal by Exemplar-Based Image Inpainting[1], par A. Criminisi*, P. Perez et K. Toyama. L'algorithme qui y est décrit et que nous avons tenté de reproduire permet d'effacer des éléments indésirables d'une photo en complétant l'image de façon intelligente. L'élément indésirable est alors remplacé par un nuage de pixels qui colle avec le fond et le reste de l'image, laissant penser que la photo ne comportait pas cet élément indésirable au préalable.

Mots-clés

Inpainting – Gradient – C++ – Montage

¹Department 1A ,Ecole Nationale des Ponts et Chaussées, Paris, France

²Department 1A ,Ecole Nationale des Ponts et Chaussées, Paris, France

³Department 1A ,Ecole Nationale des Ponts et Chaussées, Paris, France

⁴Department 1A ,Ecole Nationale des Ponts et Chaussées, Paris, France

Table des matières

| | |
|--|-----------|
| Introduction | 1 |
| 1 Fonctionnement général : | 2 |
| 2 Mise en place : | 3 |
| 2.1 Structures | 3 |
| cord et cord double • pixel bord • File de priorité | |
| 2.2 Fonctions : | 4 |
| drawclicks • EspaceBlanc • OmegalsEmpty • PointBordwOmega • convertGrey • ListeD • ListeConf • Prio • CopyImageData • findq • MainLoop | |
| 3 Résultats et Discussion : | 7 |
| 3.1 Test avec des images simples : | 7 |
| 3.2 Test sur des photos réelles : | 8 |
| 3.3 Impact de la taille du patch | 10 |
| 4 Annexe | 10 |
| 4.1 Cheminement pour la fonction <i>PointBordwOmega</i> : 10 | |
| 4.2 Cheminement pour la fonction <i>Findq</i> : | 11 |
| Références | 11 |

Introduction

Pour mieux comprendre le fonctionnement de l'algorithme dont il est question, il est nécessaire de revenir un peu en détail sur la théorie exposée initialement dans le papier de recherche de A. Criminisi*, P. Perez et K. Toyama. L'idée est donc ici de dépeindre avec nos mots la stratégie imaginée par les chercheurs pour chaque étape du traitement de l'image et de mentionner au passage les fonctions et structures que nous avons imaginées de notre côté pour remplir ces différentes étapes. Nous reviendrons par la suite dans une nouvelle partie plus en détail sur le code de chacune de ces fonctions qui ont été dans notre cas réalisé dans le langage C++. L'algorithme de traitement d'image ici exposé diffère beaucoup de ce qui se faisait précédemment pour remplacer des parties d'image. En effet, dans des travaux antérieurs, plusieurs chercheurs ont considéré la synthèse de texture comme moyen de remplir de grandes régions d'image avec des textures dites "pures" à base de motifs bidimensionnels répétitifs. Cette méthode fonctionne très bien pour remplir de grande partie d'image avec des textures consistantes mais a de grandes difficultés à remplacer des trous dans une image représentant des scènes du monde réel. En effet, ce type d'image comporte des structures linéaires qui se voient cassées dans la zone remplacée par ces algorithmes. Ainsi une idée qui permet d'éviter cette

perte de cohérence dans le remplacement du trou dans l'image est la technique dite de l' "image inpainting" qui provient du domaine de la restauration d'image. Elle consiste à propager par diffusion les structures linéaires existantes dans l'image trouée vers l'intérieur du trou. Le problème majeur de cette technique est que dès que la région est trop grande, elle introduit une forme de flou qui est visible et dérangeant pour l'œil. La technique imaginée par les chercheurs du papier que nous avons étudiée est donc de combiner le meilleur de chacun de ces deux types d'algorithmes pour obtenir un remplacement plausible de l'élément indésirable, quasiment invisible à l'œil.

1. Fonctionnement général :

Fonctionnement général : Dans un premier temps pour que l'algorithme puisse supprimer un élément indésirable de l'image, il convient de lui indiquer où se trouve cet élément et donc de délimiter sa zone de travail. Pour ce faire, avec la fonction *drawclicks* on demande à l'utilisateur de sélectionner avec sa souris un polygone entourant l'élément cible dont il indique la position des sommets par des clics gauches à répétition. Tout au long de l'opération on relie un à un les sommets sélectionnés. Une fois que l'utilisateur a terminé sa sélection, il exécute un clic droit qui termine et complète le dernier côté du polygone. Une fois la zone de travail délimitée par l'utilisateur, il est maintenant nécessaire de répertorier tous les pixels qui se trouvent à l'intérieur de ce polygone noté Ω , c'est le rôle de notre fonction *Espace Blanc*.

Par la suite, il nous faudra également les points se situant à la frontière entre la zone de l'image et la zone de travail, c'est-à-dire les pixels qui appartiennent aux différentes arêtes du polygone notés $\partial\Omega$, notre fonction *Point Bord w Omega* se charge de cela . La zone de l'image en dehors de la zone de travail sera appelée la région source et notée Φ .

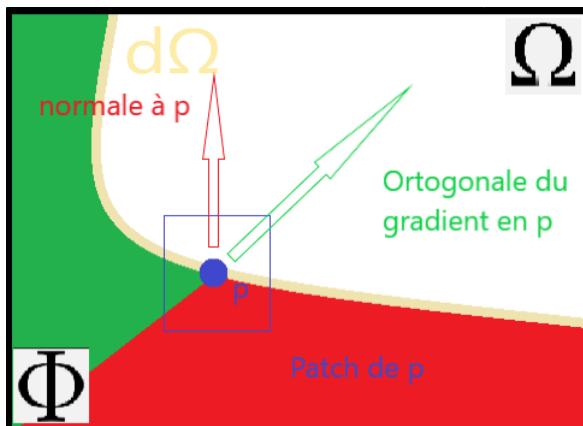


FIGURE 1. Schéma explicatif

Ensuite, l'algorithme remplit la zone Ω par un processus itératif qui se termine seulement lorsque $\Omega = \emptyset$. Cette opération itérative est réalisée par la fonction *MainLoop*. Le remplissage se fait de l'extérieur vers l'intérieur de la zone cible et donc à chaque étape les pixels modifiés se trouvent

dans $\partial\Omega$. Les étapes de remplissage de la zone de travail reposent sur deux observations clé ; et il faut absolument faire attention à l'ordre dans lequel on remplace les pixels au bord de la zone cible et il faut s'inspirer des motifs existants et ressemblants dans la région source afin de compléter la région cible.

Pour compléter Ω , on ne remplit pas pixel par pixel mais patch par patch, un patch étant une zone modulable (choisie par l'utilisateur) autour d'un pixel de $\partial\Omega$ considéré, par exemple les 8 pixels qui entourent un pixel, ou alors seulement les 4 adjacents Nord, Sud, Est et Ouest. Pour ce qui est de l'ordre dans lequel l'algorithme réalise le remplissage, c'est à dire le pixel appartenant à $\partial\Omega$ qu'il choisit de remplir à chaque itération, il est déterminé par le calcul d'une priorité $P(p)$ accordée à chaque pixel p du contour $\partial\Omega$. Le calcul de cette priorité est le suivant $P(p) = C(p) * D(p)$. Où $C(p)$ et $D(p)$ représentent respectivement le facteur de confiance et le facteur "data" du pixel p . Ces derniers sont définis par les formules suivantes :

$$C(\mathbf{p}) = \frac{\sum_{\mathbf{q} \in \Psi_p \cap (I - \Omega)} C(\mathbf{q})}{|\Psi_p|} \quad D(\mathbf{p}) = \frac{|\nabla I_p^\perp \cdot \mathbf{n}_p|}{\alpha}$$

Le terme $C(\mathbf{p})$ est appelé priorité de confiance, dans sa formule I représente l'image et Ω représente la zone de travail ainsi $I - \Omega$ représente les pixels utilisables en tant que données, Ψ_p représente le patch choisi i.e. un ensemble de pixel, $|\Psi_p|$ représente alors l'aire du patch (le nombre de pixels). L'initialisation est la suivante :

$$\forall \mathbf{p} \in I - \Omega, C(\mathbf{p}) = 1 \quad \text{et} \quad \forall \mathbf{p} \in \Omega, C(\mathbf{p}) = 0$$

La priorité de confiance représente la quantité d'informations fiables présentes autour du pixel considéré. Si les pixels d'un patch Ψ_p ont déjà été remplis par l'algorithme ou appartiennent à l'image utilisable $I - \Omega$ alors sa confiance est grande, en revanche si peu ou aucun pixel du patch n'a été rempli sa confiance sera faible. On calcule la confiance pour tous les points de $\partial\Omega$ avec la fonction *Listconf*.

On présente sur la figure suivante un exemple de calcul de ce terme de confiance sur une forme particulière.

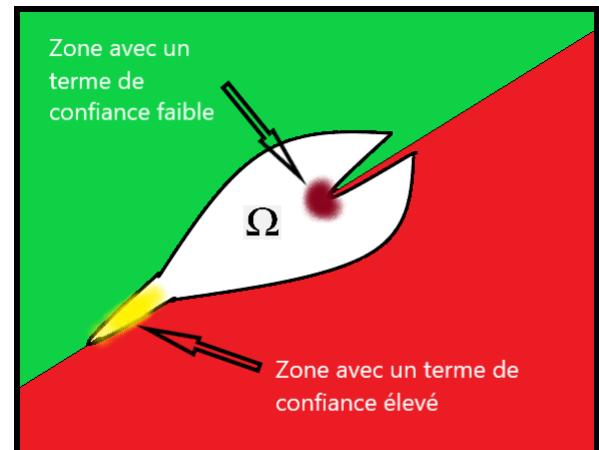


FIGURE 2. Schéma explicatif du terme de confiance

Le terme $D(\mathbf{p})$ est appelé priorité de data, dans sa formule n_p représente la normale au contour au pixel p , ∇I_p^\perp représente l'orthogonal du gradient au pixel p (on explique plus loin comment calculer ce terme), enfin α représente un coefficient de normalisation valant 255 afin que le terme de data ne soit pas trop influant par rapport au terme de confiance. Le terme de data à la différence du terme de confiance n'est calculé que sur le contour.

$$\forall p \in \partial\Omega \quad D(\mathbf{p}) = \frac{|\nabla I_p^\perp \cdot \mathbf{n}_p|}{\alpha}$$

Le terme de "data" représente l'idée suivante : si il y a une forte démarcation dans l'image source ($I - \Omega$) au niveau du contour et que cette démarcation pointe vers la zone de travail alors il est logique de se dire que cette forte démarcation se prolonge dans la zone de travail (Ω). On présente sur la figure suivante un exemple de calcul de ce terme de "data" sur la même forme particulière que pour le terme de confiance.

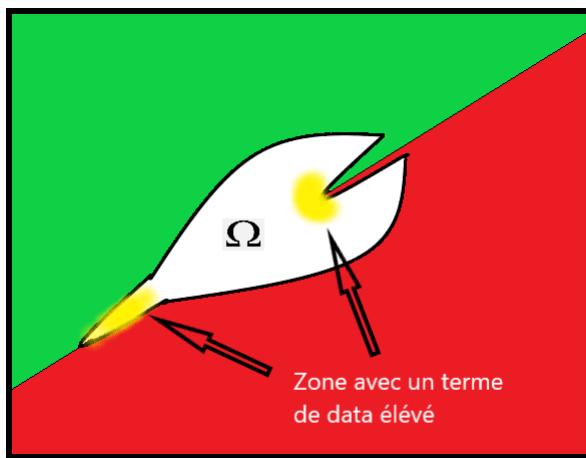


FIGURE 3. Schéma explicatif du terme de "data"

On calcule tous les termes de "data" de $\partial\Omega$ avec la fonction [liste D](#), en amont de cette fonction il nous faut avoir l'image en nuance de gris qui nous permet d'obtenir le gradient nécessaire au calcul de $D(\mathbf{p})$, c'est le rôle de [convert Grey](#).

Une fois la priorité de chacun des pixels du contour $\partial\Omega$ déterminée, on met la totalité de ces derniers dans une file de priorité avec la fonction [prio](#), ce qui nous permet d'en retirer le pixel à remplir en premier, c'est-à-dire celui avec la priorité la plus grande.

Maintenant que l'algorithme a déterminé le pixel cible et son patch associé qu'il doit remplir en priorité, il lui reste à déterminer avec quelles couleurs remplir ce dernier. Pour cela l'algorithme parcourt l'image source ($I - \Omega$) patch par patch et cherche le patch le plus proche des informations existantes dans notre patch cible. Une fois le patch le plus proche trouvé, l'algorithme associe les couleurs du patch le plus proche au patch cible. Une fois arrivé ici on recommence l'opération jusqu'à ce qu'il ne reste plus aucun pixel dans la zone de travail.

2. Mise en place :

Dans cette partie nous nous penchons plus en détail sur la mise sous forme de code des fonctions décrites précédemment. Il est important de noter que dans toute la suite nous utilisons un grand nombre de fonctions de la bibliothèque Imagine++, pour les distinguer dans ce rapport des fonctions que nous avons implémenté, en bleu, nous leur avons donné la couleur violet.

2.1 Structures

2.1.1 cord et cord double

La structure cord (resp. cord double) est une structure qui possède deux entiers (resp. deux doubles) qui vont servir par la suite à décrire la position d'un pixel sur l'image (resp. à avoir les valeurs d'un vecteur tel qu'un gradient par exemple). Les structures sont munies des opérateurs classiques (+, -, ×) (par un entier (resp. un double) ou par un autre cord (resp. cord double) donc le produit scalaire). Elles ont aussi toutes les deux la norme L^2 (et en plus la norme L^∞ pour cord). La structure cord a aussi une méthode angle qui retourne l'angle (orienté) entre deux vecteurs. La structure cord double a quand à elle une méthode rotation qui prend en argument un angle et fait tourner le vecteur de cet angle (ce qui va nous être utile pour calculer le gradient orthogonal). Et une méthode normalize qui divise chaque composante par la norme L^2 du cord double.

2.1.2 pixel bord

La structure pixel bord est une structure qui possède un cord qui nous donne la place du pixel dans l'image, ainsi qu'un double p représentant la priorité du pixel. Cette structure possède également des opérateurs de comparaison classiques ($<$, $>$, $=$) sur cette donnée de priorité. Elle nous permet de caractériser les pixels se situant sur le bord de la zone de travail de l'algorithme en leur conférant une priorité suivant le principe expliqué dans l'introduction.

2.1.3 File de priorité

La file de priorité est une structure de donnée déjà bien connue et qui s'adaptait parfaitement à notre situation. En effet cette structure permet de stocker les éléments qui nous intéressent tout en nous permettant de sortir uniquement l'élément le plus pertinent, à savoir l'élément avec la plus grande priorité dans notre cas.

L'ajout d'un élément ou le retrait de l'élément pertinent se fait en $O(\log n)$ donc relativement rapidement. Cette structure peut se représenter comme un arbre binaire, un parent ayant une priorité supérieure à celle de ses fils. Notre file de priorité est une classe construite à l'aide des vecteurs déjà présents dans C++ et nous l'avons dotée des méthodes *push*, qui ajoute l'élément donné en argument et l'y met à sa place dans l'arbre, *pop* qui retire l'élément à la racine et réajuste l'arbre et *empty* qui vérifie si la file est vide ou non.

Le choix de la file de priorité n'est qu'un choix d'optimisation de l'algorithme. Celui-ci se base sur un choix intelligent de la zone à remplir, décidé à partir de la priorité du pixel

représentant la zone. La zone dont le pixel représentant a la priorité la plus grande est choisie comme zone à remplir. On pourrait alors remplir une liste des priorités et chercher l'élément ayant la plus grande priorité ensuite, mais la recherche de l'élément le plus grand d'une liste se fait en $O(n)$ là où la file de priorité nous le permet en $O(\log n)$.

2.2 Fonctions :

2.2.1 drawclicks

But de la fonction : Remplir un vecteur de coordonées initialement vide avec l'ensemble des points qui composent le polygone qui représente l'ensemble Ω .

Fonctionnement de la fonction : La fonction *drawclicks* est une méthode qui ne retourne rien mais qui va modifier un vecteur de cord ici nommé *ListePoint*. Pour se faire on utilise la fonction *getmouse* qui a deux entiers x,y modifie leurs valeurs pour leur donner celle de la position associée au clic qui vient d'être effectué. De plus pour tout clic effectué, cette fonction renvoie 1 si c'est un clic gauche. Ainsi, tant que l'utilisateur fait un clic gauche on dessine le segment entre le clic à l'instant t et $t-1$ grâce à la fonction *drawline*. En parallèle, on ajoute son clic à *ListePoint*. Dès que celui-ci effectue un clic droit alors on dessine un segment entre le premier coordonné de la liste et le dernier, et on sort de la boucle. La fonction est alors terminée et on a bien notre *ListePoint* qui contient tous les coordonnées de notre Polygone.

Input: *ListePoint*-vecteur de cord

Output: *None*

Function *drawclicks* :

```

→ x,y
getmouse(x,y)
ListePoint ← {x,y}
stop ← false
while !stop do
    if getmouse(x,y) == 1 then
        O ← ListePoint[-1]
        drawline(x,y,O.x,O.y)
        ListePoint ← {x,y}
    end
    else
        O ← ListePoint[-1]
        P ← ListePoint[0]
        drawline(P.x,P.y,O.x,O.y)
        stop ← true
    end
end
return
End Function

```

2.2.2 EspaceBlanc

But de la fonction : Modifier l'Image des confiances pour que tous les points appartenant à Ω aient une valeur de confiance 0 et les autres 1.

Fonctionnement de la fonction : La fonction *EspaceBlanc* est une méthode qui va modifier l'image de double *conf*

avec les bonnes valeurs de confiances. Pour cela on parcourt l'image sur tous ses pixels et on regarde si le pixel est à l'intérieur ou à l'extérieur de notre ensemble Ω à l'aide de la fonction *in*. Dans cette fonction, on calcule pour tous les sommets du polygone l'angle orienté entre : le vecteur partant du pixel jusqu'au sommet, et le même vecteur mais avec le sommet d'après. Sachant que pour le dernier sommet on regarde l'angle avec le premier sommet de la liste. On effectue ensuite la somme de ces angles, si celle-ci vaut 2π alors on est à l'intérieur. Si on écrit cela de manière quantitative on a :

$$\forall P \in \Omega, \left(\sum_{k=0}^{nb.sommets-1} \widehat{S_k P S_{k+1}} \right) + \widehat{S_{nb.sommets} P S_0} = 2\pi$$

Si la fonction retourne *true* alors on met à jour la *conf* = 0 et le pixel en blanc. Ensuite on reparcourt l'image sur tous ses pixels et maintenant on regarde si les 8 voisins du pixels sont à l'intérieur à l'aide de la fonction *autour* en regardant la valeur de *conf* en ces voisins. Si elle renvoie *true* alors tous ses voisins sont à l'intérieur, donc on en déduit alors que le pixel est bien à l'intérieur donc on met à jour ses caractéristiques. Ce double parcours est ici nécessaire car le calcul d'angle est soumis à une marge d'erreur qui peut entraîner un oubli de certains pixels dans l'ensemble Ω .

Input: *W*-entier , *H*-entier , *ListePoint*-vecteur de cord, *r,g,b* - 3 tableaux de byte, *conf* -Image de double

Output: *None*

Function *EspaceBlanc* :

```

foreach p ∈ Image do
    if in(ListePoint, p) then
        | (r,g,b)[p] ← 255
        | conf(p) ← 0
    end
    else
        | conf(p) ← 1
    end
end
foreach p ∈ Image do
    if autour(p,W,H,conf) then
        | (r,g,b)[p] ← 255
        | conf(p) ← 0
    end
end
return
End Function

```

2.2.3 OmegalsEmpty

But de la fonction : Cette fonction a pour but de nous dire si l'ensemble Ω est vide ou non.

Fonctionnement de la fonction : La fonction *OmegalsEmpty* parcourt toute l'image et regarde la confiance du pixel à chaque fois. Si une des ces valeurs est nulle, alors il reste un ou plusieurs pixels dans Ω donc on retourne *false*. Sinon on retourne *true* car tous les pixels ont une confiance non nulle donc Ω est vide.

2.2.4 PointBordwOmega

But de la fonction : Renvoie la liste des pixels qui sont dans sur le contour de Ω

La rédaction de cette fonction a nécessité le passage par plusieurs possibilités de code décrit ici : 4.1

Fonctionnement de la fonction : La fonction prend en argument la largeur W, la hauteur H de l'image et un tableau d'entiers de la taille de l'image : TableIn rempli par des 0 et des 1 et représentant l'appartenance d'un pixel à Ω . On commence par créer la liste de pixel bord des pixels appartenant à $\partial\Omega$: ListeBord. On parcourt l'entièreté de l'image, dès qu'un pixel n'est pas dans Ω on regarde ses voisins Nord, Sud, Est et West. Si l'un d'entre eux est dans Ω alors c'est que le pixel qu'on considère est dans la frontière on l'ajoute à ListeBord. Finalement, on revoie ListeBord.

Input: W -entier , H -entier , TableIn-Tableau d'entiers

Output: ListeBord-liste de pixel bord

Function PointBordwOmega :

```

→ ListeBord
foreach p ∈ Image do
    if TableIn[p]! = 0 then
        if TableIn[voisinsp] == 0 then
            | ListeBord ← p
        end
    end
end
return ListeBord
End Function

```

2.2.5 convertGrey

But de la fonction : Cette fonction a pour but de convertir notre image en couleur en une image en noir et blanc (nuance de gris).

Fonctionnement de la fonction : La fonction *convertGrey* a donc en entrée 3 tableau de byte r, g, b ayant des valeurs de 0 à 255. Et renvoie un tableau de byte t ayant des valeurs de 0 à 255. Pour savoir comment pondérer les valeurs de (r,g,b) pour chaque pixel, on a pris la recommandation 601 de la I.T.U (International Telecommunication Union) qui conseille d'avoir :

$$t = \lfloor 0.299 \times r + 0.587 \times g + 0.144 \times b \rfloor$$

2.2.6 ListeD

But de la fonction : Calculer la priorité de data sur l'ensemble du contour $\partial\Omega$

Fonctionnement de la fonction : La fonction calcule le gradient sur l'ensemble du contour $\partial\Omega$ et la normale au contour sur les mêmes points, ensuite on peut directement calculer la priorité de data via la formule $D(\mathbf{p}) = \frac{|\nabla I_p^\perp \cdot \mathbf{n}_p|}{\alpha}$, ainsi la fonction *ListeD* ne fait qu'appeler les fonctions *normal* et *liste grad* qu'on explicite dans cette sous-partie.

Fonctionnement de liste grad : L'algorithme a pour objectif de donner le gradient associé à chaque point du contour $\partial\Omega$.

Le gradient d'un pixel du bord est défini comme la valeur maximale du gradient dans la totalité du patch autour du pixel p . Cependant tous les pixels de confiance 0 ne sont pas pris en compte¹.

Fonctionnement de liste normal : L'algorithme prend en entrée la liste des pixels bord et les niveaux de confiance et nous renvoie la liste des normales en chaque point du contour. Pour un pixel bord donné, il regarde l'ensemble des points appartenant à la zone de travail parmi ses 8 voisins directs - cet ensemble est connexe - ainsi en parcourant dans un sens arbitraire (trigonométrique ou horaire) et connaissant le premier et dernier pixel de cet ensemble connexe, on détermine les deux pixels voisins du pixel p appartenant au contour. Il suffit juste de faire la différence de ces deux pixels pour avoir le vecteur tangent, et d'effectuer une rotation pour avoir le vecteur normal².

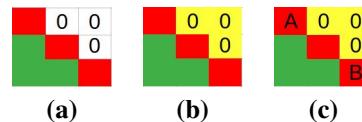


FIGURE 4. (a) patch du pixel p et confiance des points de la zone de travail

(b) recherche de la forme connexe de confiance 0, ici en jaune

(c) A et B sont bien les points voisins du pixel p et les points encadrant la forme connexe

Input: Liste pixel bord- vecteur de cord,

I - Image de byte,

conf - Image de double

Output: listeD - liste de double

Function listeD :

```

liste grad (I,conf,Liste pixel bord) → grad
liste normal (I,conf,Liste pixel bord) → normal
foreach p ∈ Liste pixel bord do
    | grad[p] · normal[p] | → listeD[p]
    | listeD[p] |
    α → listeD[p]
end
return listeD
End Function

```

2.2.7 ListeConf

But de la fonction : Cette fonction a pour but de calculer la confiance des pixels bord.

Elle prend en argument une liste de pixels bord et renvoie la liste de leur taux de confiance, le terme $C(p)$ défini précédemment. Chaque élément d'indice i de cette liste correspond à la confiance

1. Tant dans le calcul du gradient, grâce à une fonction faite par nous-même(*gradRev*), que dans le calcul du maximum car on force le gradient de ces points à être nul.

2. Le sens de la rotation et du vecteur tangent n'importe pas car la priorité de data calcule la **valeur absolue** du produit scalaire entre le gradient orthogonal et la normale

du pixel bord à l'emplacement i dans la liste passée en argument.

La confiance en un pixel bord est calculée par la formule :

$$C(p) = \frac{\sum_{p \in \Psi_p \cap (I - \Omega)}}{|\Psi_p|}$$

La confiance d'un pixel bord p est donc celle de la moyenne de son patch Ψ_p ce qui est logique puisque le but est de compléter d'abord là où on a le plus d'informations fiables et par groupement de pixels, les patchs, la confiance d'un pixel bord se doit donc d'être celle de son patch associé.

Pour notre fonction, nous avons décidé d'utiliser des vecteurs de la bibliothèque standard de C++ pour représenter nos listes, l'avantage étant que l'on peut y ajouter des éléments. Nos patchs sont eux aussi des vecteurs de cord, calculés à l'aide de la fonction *calc patch* qui nous permet de calculer plusieurs types de patch, et plusieurs tailles de patch en fonctions de paramètres globaux déjà donnés avant de lancer l'algorithme. Nous accédons à la confiance des pixels (cord) grâce au tableau de confiance *conf* passé en argument. Nous rappelons au passage que la confiance d'un pixel de Ω est de 0. La taille du patch $|\Psi_p|$ est donnée par le nombre de pixels dans le patch, *ie* la taille du vecteur le représentant.

2.2.8 Prio

But de la fonction : Créer la file de priorité des pixels bord.

Cette fonction prend en argument une liste de cord et deux listes de doubles. La liste de cord est la liste de nos pixels appartenant à $\partial\Omega$, la première liste de double est celle des confiances, calculée avec *ListeConf*, la deuxième liste de doubles est celle des coefficients de data ($D(p)$), calculés avec *ListeD*.

Nous créons une file de priorié F. La fonction parcours la liste des cords et crée à chaque fois un pixel bord associé au cord. La priorité est créée à partir des listes de confiances et de data. On rappelle que le pixel à l'indice i dans la liste des pixels de $\partial\Omega$ a pour confiance celle stockée au même indice dans la liste des confiance. Il en va de même pour les termes de données ($D(p)$). Nous pouvons donc facilement extraire les bons éléments afin de calculer la priorité $P(p) = C(p)D(p)$ du pixel p . Le pixel bord ainsi créé est alors ajouté à la file de priorité à l'aide de la méthode *push* et se retrouve ainsi à la place correspondant à sa priorité dans l'arbre des priorités. A la fin du parcours de la liste des pixels de $\partial\Omega$, la file de priorité est terminée et la fonction la renvoie afin qu'elle puisse être utilisée dans la suite de cet algorithme. Cette fonction peut paraître simple mais est essentielle à l'algorithme, dont la particularité est le choix intelligent de la zone à remplir à chaque itération, choix fait à partir des priorités.

2.2.9 CopyImageData

But de la fonction : Copier les informations du patch q dans la zone non connue du patch p.

Fonctionnement de la fonction : La fonction prend en argument p qui correspond au pixel où on travaille et q le pixel tel que la distance entre les deux patchs centrés en p et q soit minimale. Elle fait premièrement appel à la fonction *calcPatch* (expliquer précédemment) pour générer le patch centrer en p. Ensuite elle va parcourir chaque élément du patch et regarder si il appartient à Ω . S'il appartient à cet ensemble, alors on recopie dans ce pixel les informations du pixels correspondant dans le patch centré en q.

Input: *q-cord , p-cord , conf-Image de double,r,g,b*
- 3 tableaux de byte

Output: *None*

Function *CopyImageData* :

```

patch ← calcPatch (p, conf.W, conf.H)
foreach  $p_k \in patch$  do
    if  $conf(p_k) == 0$  then
         $q_k \leftarrow p_k - p + q$ 
         $(r, g, b)[p_k] \leftarrow (r, g, b)[q_k]$ 
    end
end
return
End Function

```

Alternative de la fonction : Pour les cas où le gradient est nul (ou faible) on a fait une fonction *CopyImageDataForGradNul* similaire) à la précédente fonction sauf que celle-ci va copier dans tous les pixels du patch p appartenant à cette ensemble les informations du pixels p.

2.2.10 findq

But de la fonction : trouver le patch qui correspond le mieux

La rédaction de cette fonction a nécessité le passage par plusieurs possibilités de code décrit ici : 4.2

Notre algorithme se base sur deux hypothèses importantes :

- (1) la géométrie de l'image doit être conservée (c'est pourquoi le gradient compte dans la priorité),
- (2) les motifs de l'image ont tendance à se répéter.

C'est cette deuxième hypothèse qui justifie la fonction *findq*. Cette fonction parcours l'image et trouve le patch ayant la meilleure correspondance avec le patch qui doit être complété. Cette correspondance ne se base que sur les pixels du patch qui appartiennent à $I - \Omega$. Il faut de plus que le représentant du patch appartienne à $I - \Omega$ avant le début de l'algorithme, *ie* qu'il ait une confiance de 1 car sinon l'algoriyhme peut juste répéter le même motif en boucle.

Comme la recherche sur toute l'image sera à terme appliquée à tous les points de la zone de travail Ω , le code n'est pas très rapide. Une façon d'optimiser cela, est de vérifier si le patch de l'image source $I - \Omega$ se trouve le long de l'orthogonal du gradient, à une certaine tolérance près, ceci traduit

le fait qu'un fort gradient orienté vers la zone de travail a de fortes chances de se reproduire (hypothèse (1)), ainsi il est logique de chercher dans cette zone, de plus cela optimise efficacement notre code.

Une bande qui va dans le sens de l'orthogonal du gradient est alors délimitée et c'est dedans que la recherche s'opère.

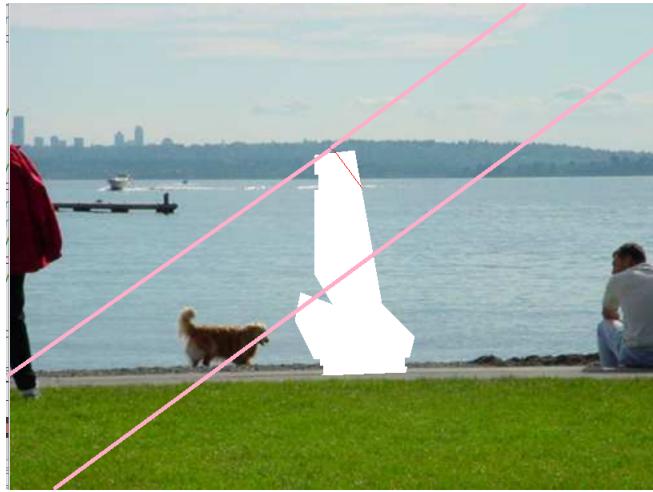


FIGURE 5. exemple de la restriction de recherche

On peut voir ici en rouge la direction du gradient. Les lignes roses délimitent le bandeau de recherche. On remarque bien que la recherche est restreinte à une zone d'aire bien inférieure à celle de l'image. On remarque aussi qu'une proportion non négligeable de la bande est remplie de pixels de Ω et donc par des pixels non pris en compte lors de la recherche ce qui accélère encore plus la recherche.

La correspondance se fait à l'aide d'une norme 2. Les patchs sont considérés comme des vecteurs de $(R^3)^{|\Psi_p \cap (I - \Omega)|}$ où p est le pixel représentant le patch à compléter. Ensuite le patch le plus proche de Ψ_p selon la norme 2 sur cet espace est alors choisi comme modèle pour la complétion de Ψ_p .

2.2.11 MainLoop

But de la fonction : Cette fonction est le corps de notre algorithme, elle a pour but de remplir l'espace Ω .

Fonctionnement de la fonction : La fonction *MainLoop* est le cœur de notre code, elle remplit la zone de travail Ω progressivement.

Tant que notre zone de travail Ω n'est pas vide (*OmegaIsEmpty*) on met à jour le contour $\partial\Omega$, ensuite on calcule la priorité de chaque pixel $C(\mathbf{p}) \cdot D(\mathbf{p}) \quad \forall p \in \partial\Omega$, puis on tri les pixels par priorité, enfin on choisit le pixel bord à la plus grande priorité p_{max} , ce sera le pixel que l'on remplit en premier.

Maintenant comment remplit-on le patch, pour ceci on parcourt toute l'image source $I - \Omega$ et on cherche le minimum de la distance entre notre patch p_{max} et les patchs de l'image

source à l'aide de la fonction *findq*, cependant si le gradient est très faible dans le patch de p_{max} alors on remplit le patch par la couleur de p_{max} .

Input: W -entier , H -entier , *ListePoint*-vecteur de cord,r,g,b - 3 tableaux de byte

Output: None

Function *MainLoop* :

```

→ conf(W,H),ImageInGrey(W,H)
→ D,C,F,ListePixelBord
drawclicks(ListePoint)
EspaceBlanc(W,H,ListePoint,r,g,
b,conf)
while !OmegaIsEmpty(W,H,conf) do
    putColorImage(0,0,r,g,b,W,H)
    ListePixelBord ← PointBordwOmega
    (W,H,conf)
    convertGrey(W,H,r,g,b)
    D ← ListeD
    (ImageInGrey,ListePixelBord,conf)
    C ← ListeConf (ListePixelBord,conf)
    F ← prio (ListePixelBord,C,D)
    pmax ← F.pop()
    grad ← gradRev(pmax.P,conf,ImageInGrey)
    if grad.norm2() ≤ 3 then
        | CopyImageDataForGradNul(pmax.P,conf,r,
        | g,b)
    end
    else
        | q ← findq(W,H,pmax,conf,r,g,b)
        | CopyImageData(q,pmax.P,conf,r,g,b)
    end
    UpdateConf(pmax.P,conf)
end
return
End Function

```

3. Résultats et Discussion :

3.1 Test avec des images simples :

Dans un premier temps nous avons voulu tester notre algorithme dans le cas d'images simples, c'est-à-dire pour des images comportant un nombre limité de couleurs. L'exemple le plus simple étant une image bicolore dont les couleurs sont séparées par une ligne droite.

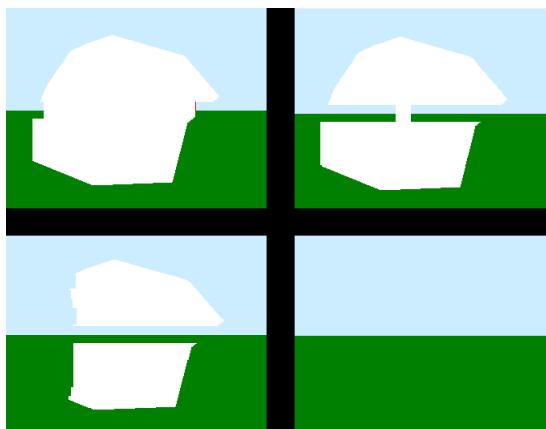


FIGURE 6. Test de l'algorithme sur une image bicolore

On peut donc observer sur la Figure 6 l'application de l'algorithme à une image de ce type. On observe alors que l'algorithme commence bien par se concentrer sur les zones où le gradient est le plus fort, ce qui correspond à un terme de "data" élevé ce qui permet de prolonger la ligne droite de l'horizon. Une fois l'horizon tracé, il poursuit bien par l'élimination progressive des zones de gradient nul, finalement on obtient bien de nouveau l'image initiale.

Nous nous sommes par la suite penchés sur l'idée de réaliser un test de l'algorithme sur une image simple de nouveau mais cette fois tricolore.

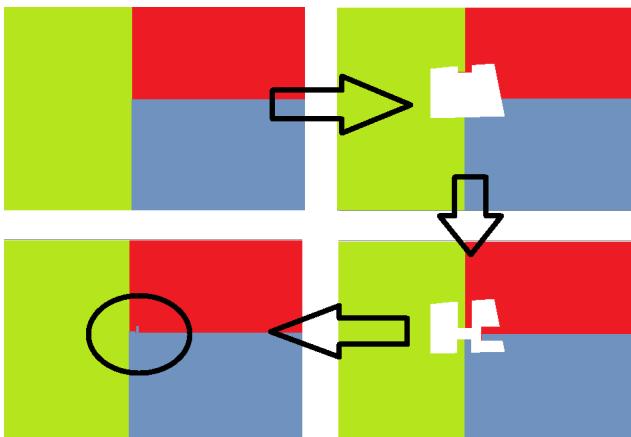


FIGURE 7. Test de l'algorithme sur une image tricolore

On observe cette fois une différence entre l'image initiale en haut à gauche et l'image finale en bas à gauche. Bien que minime, on peut voir que la zone bleue dépasse un peu sur la zone rouge. Au moment de l'intersection des trois zones, l'algorithme se voit contraint de faire un choix dans le complément du dernier patch qui dépend de la direction du gradient. Le gradient le plus grand dépendant des couleurs choisies, c'est finalement le premier qui arrive à l'intersection qui choisit la couleur dominante. Ce choix de gradient implique donc un choix d'une des couleurs dominantes sur une autre, d'où l'incohérence observée.

3.2 Test sur des photos réelles :

Il est maintenant temps de tester l'algorithme avec des photos réelles, pour cela nous avons commencé le test sur une photo de la façade de l'école des Ponts et Chaussées de laquelle nous avons essayé de retirer les panneaux présents devant. Pour le panneau le plus éloigné et donc le plus petit sur l'image on observe un remplacement quasiment invisible à l'œil nu qui nous conforte sur les capacités de notre algorithme.

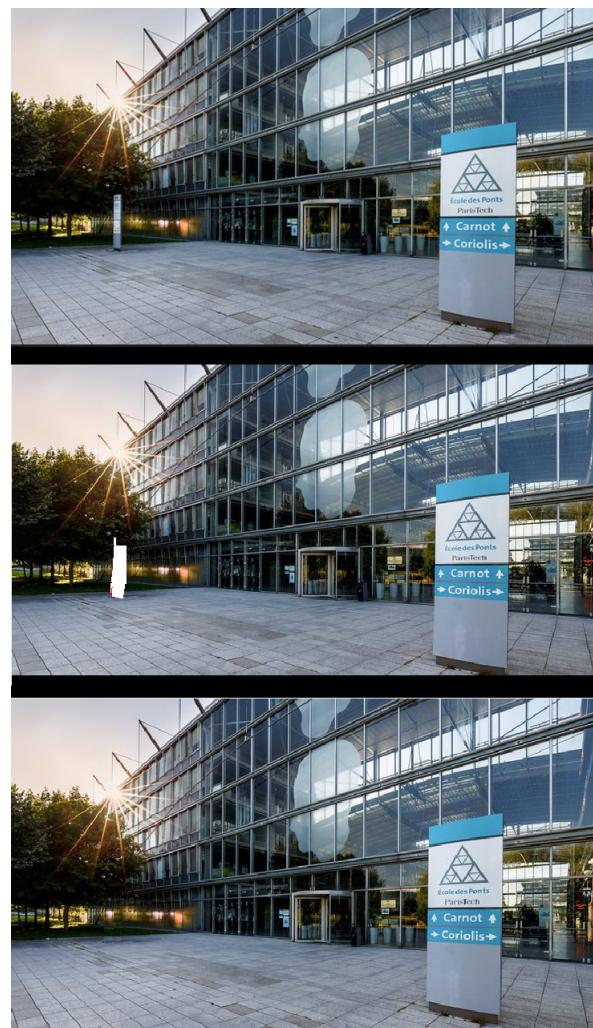


FIGURE 8. Test sur la façade des Ponts et Chaussées : petit panneau

En revanche le test sur le panneau le plus proche et donc bien plus volumineux est bien moins satisfaisant. En effet, des parties de l'image indésirables ont été reproduites par l'algorithme ce qui laisse imaginer la présence du panneau sur l'image modifiée.

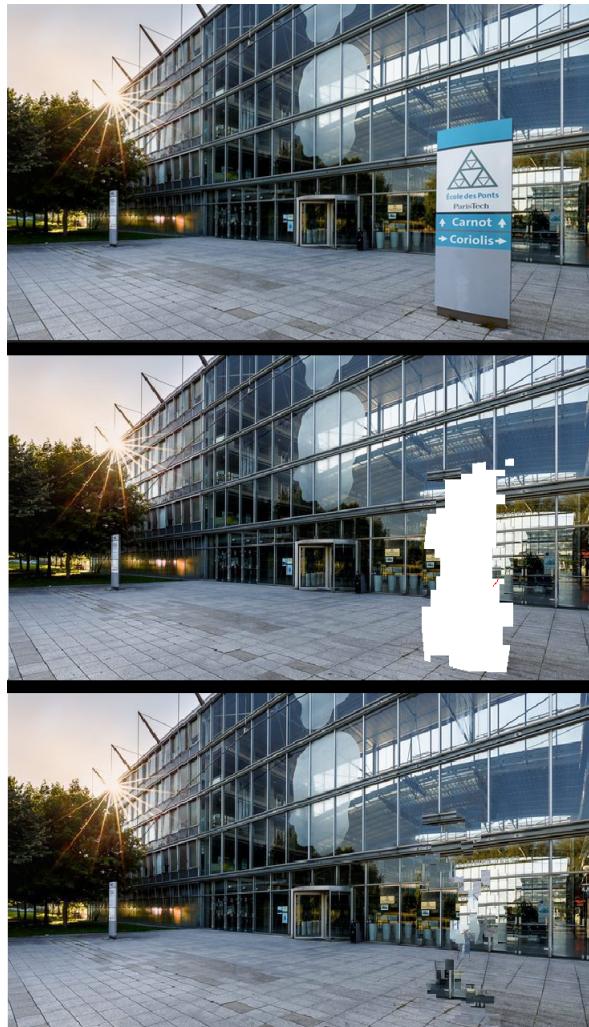


FIGURE 9. Test sur la façade des Ponts et Chaussées : grand panneau

Ceci s'explique assez facilement par le fait que la qualité de l'application de notre algorithme dépend de la quantité d'informations qu'il possède lorsqu'il recherche un patch similaire dans l'image source. Ainsi plus l'objet que l'on souhaite supprimer représente une grande part de l'image moins l'algorithme sera efficace. De la même façon, si aucune partie de l'image source ne colle bien avec la partie à remplacer l'algorithme ne peut que choisir le patch le moins éloigné mais pas forcément celui le plus satisfaisant.



FIGURE 10. Test sur une photo similaire au papier de recherche étudié

On a ensuite voulu tester notre algorithme sur une des images de l'article étudié afin de comparer les résultats. On a pu voir que notre algorithme a été très performant sur cette image en ne laissant quasiment aucune imperfection visible à l'œil nu lors du remplissage. On peut expliquer ce succès grâce au fort gradient qu'il y a entre les différentes parties de l'image et aussi au caractère assez lisse des couleurs de chacune d'entre elle.



FIGURE 11. Test sur une photo avec des textures polychromatiques

On peut de plus voir que l'algorithme est assez performant dans le cas des textures polychromatiques comme au niveau

du mur en pierre. En effet, l'effacement de la personne est quasiment invisible car les motifs répétés dans le mur n'ont pas de rigueur particulière et sont donc assez facile à reproduire sans inquiéter l'œil humain.



FIGURE 12. Autre test

3.3 Impact de la taille du patch

Comme nous l'avons expliqué précédemment, notre algorithme utilise un patch qui lui permet de remplir la zone de travail. Ce patch correspond à la taille du pinceau avec lequel l'algorithme recompose la zone de travail. Plus ce dernier est grand, plus le remplissage est grossier et inversement. En revanche, il existe tout de même une taille de patch trop petite pour laquelle l'algorithme ne possède plus assez d'informations autour de son pixel cible pour remplir la zone de façon cohérente. Cette taille minimale dépend de la taille des motifs qui composent l'image et que l'algorithme doit reproduire.

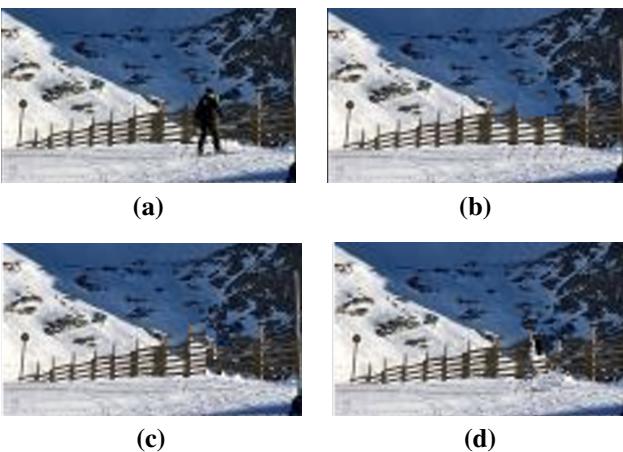


FIGURE 13. Impact de différentes tailles de patch sur une photo de skieur :

- (a) Image originale avec le skieur
- (b) Image sans le skieur avec un patch 21*21 pixels
- (c) Image sans le skieur avec un patch 11*11 pixels
- (d) Image sans le skieur avec un patch 3*3 pixels

Par exemple ici avec cette image de skieur on remarque que la taille de patch idéale doit être au moins de la taille du motif de la barrière en bois derrière le skieur. Pour le patch de 21*21 pixels c'est le cas et on remarque que la barrière est bien reproduite. En revanche pour des patchs plus petits comme celui de 11*11 pixels et 3*3 pixels ce n'est pas le cas et l'algorithme complète la zone de travail avec des imperfections comme une trop forte présence de neige.

4. Annexe

4.1 Cheminement pour la fonction *PointBordomega* :

Pour la mise en place de cette fonction nous sommes passés par plusieurs cheminements de pensée avant d'en arriver à la fonction décrite dans la partie Mise en place. Pour rappel, le but de la fonction est de récupérer le contour d'un polygone tracé par la récupération de ses sommets et la fonction *drawLine* de Imagine++ qui fait la discréétisation d'un segment sur une interface graphique compte-tenu des coordonnées de ses deux extrémités. La difficulté principale rencontrée est que l'on ne connaît pas exactement la méthode de discréétisation utilisée par la cette fameuse fonction *drawLine*, il est donc compliqué de récupérer avec exactitude les pixels composants le segment tracé qui apparaît à l'utilisateur sur l'interface graphique. En effet, le tracé d'une ligne oblique sur une interface graphique connaît un grand nombre de possibilités qui dépend de la conception de la méthode employée. On peut le voir par exemple sur la figure suivante qui discréétise le tracé d'un cercle où deux méthodes plausibles sont représentées.

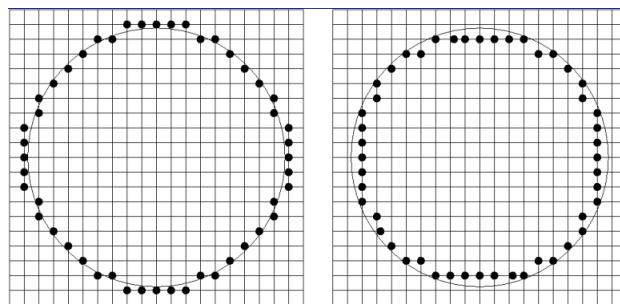


FIGURE 14. Deux discréétisations différentes d'un cercle.

Ainsi une première idée pour obtenir des points qui constituent le contour de notre polygone était de se servir de l'interface graphique. L'idée est d'ouvrir une nouvelle fenêtre graphique en arrière plan sur laquelle tracer en couleur le polygone avec la fonction *drawLine* à partir des coordonnées des sommets, puis de regarder pixel par pixel sur cette interface graphique la couleur d'un pixel, si le pixel est coloré on l'ajoute à la liste des points du contour. Finalement on referme l'interface. Cette méthode fonctionne bien mais n'est pas très satisfaisante dans le sens où elle nécessite l'ouverture puis la fermeture d'une interface graphique supplémentaire. Nous avons donc par la suite opté pour l'implémentation d'une méthode de discréétisation de segment en utilisant le schéma de l'algorithme de Bresenham. Cependant comme décrit précédemment, l'impossibilité d'avoir la même discréétisation que celle de l'algorithme utilisé par la fonction *drawLine* nous a conduit à un écart de quelques pixels avec le contour tracé par *drawLine*. Visuellement ce souci n'était pas important mais représente un problème vis à vis de l'appartenance ou non des points considérés comme étant dans $\partial\Omega$ à l'intérieur de Ω . En effet par la suite notre algorithme doit avoir la certitude de la non appartenance des points du contour à l'ensemble de son espace de travail ce qui n'est pas le cas.

s'il n'y a pas une certaine cohérence entre la discréttisation utilisée pour connaître les points de l'intérieur Ω et celle utilisée pour obtenir les points de son contour $\partial\Omega$. C'est donc en partant des résultats de la discréttisation de l'intérieur de Ω fait par la fonction [EspaceBlanc](#) que finalement nous calculons le contour pour palier à ce problème.

4.2 Cheminement pour la fonction [Findq](#) :

Cette fonction a eu trois versions. La première reprenait le même principe que celle que nous utilisons actuellement, mais n'était pas dotée de l'optimisation de la restriction de recherche. La fonction parcourrait toute l'image et la seule distinction était sur le fait que notre pixel représentant du patch testé appartenait à $I - \Omega$ au début de l'algorithme ou non. Le calcul de la distance induite par la norme 2 sur notre espace $(R^3)^{|\Psi_p \cap (I - \Omega)|}$ étant assez lourd, cette fonction prenait du temps à l'exécution, et il nous arrivait de devoir attendre une heure ou plus avant la fin de notre programme. Il nous fallait donc optimiser.

La première idée était de se restreindre à deux cones dans le sens de l'orthogonal du gradient, délimités à l'aide d'un produit scalaire et d'une tolérance sur celui-ci : il devait être supérieur en valeur absolue à $1 - \varepsilon$, ε étant notre tolérance.

Références

- [1] P. Perez A. Criminisi* and K. Toyama. Region filling and object removal by exemplar-based image inpainting. *IEEE TRANSACTIONS ON IMAGE PROCESSING*, 13 :1–13, 2004.

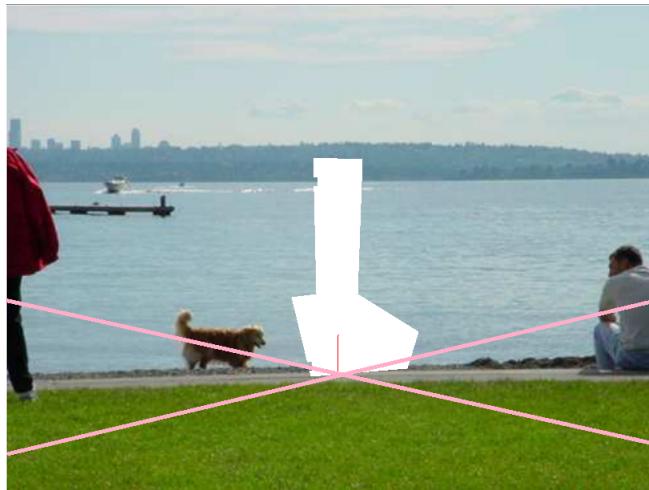


FIGURE 15. exemple de la restriction de recherche avec cônes

Nous avons en rouge la direction du gradient en rose la délimitation des cônes de recherche.

Après plusieurs essais, aucun résultat ne paraissait convaincant. Cette méthode ne marchait donc pas, sauf pour les images bicolores auxquelles cette technique était approprié et aussi la plus rapide.

Nous en sommes donc arrivés à la version qu'on utilise actuellement, où la restriction de recherche est une bande.

Grâce à cette méthode, nous avons des images convaincantes, et nous avons divisé notre temps d'exécution par 10.