Projet 18 : Trucage photographique par inpainting

Clément Allain, Thomas Bekalarek, Thomas Gerbeaud,
Louis Lesueur, Clément Norodom, Maxime Poli

Encadrant : Pascal Monasse

2019

FIGURE 1 – Exemple d'utilisation de l'algorithme (originale au-dessus et photo truquée dessous)

Résumé

Nous avons étudié la méthode d'"inpainting" par correspondance d'imagettes. Cet algorithme permet de supprimer des zones d'une image et de combler harmonieusement les vides laissés. L'utilisateur sélectionne la zone à supprimer et l'algorithme va alors la remplir à partir d'imagettes issues du reste de l'image de manière à masquer le trucage le mieux possible. Nous présentons notre implémentation ainsi que les résultats obtenus et les difficultés rencontrées.

Sommaire

Introduction **3**

1 Principe de l'algorithme **3**
1.1 Calculer les priorités des imagettes 3
1.2 Propagation de l'information 4
1.3 Actualisation des valeurs de confiance 4
1.4 L'algorithme 4

2 Implémentation **5**
2.1 Le code 5
2.2 L'interface graphique 5
2.3 La parallélisation 6

3 Résultats obtenus **7**
3.1 Exemples 7
3.2 Difficultés rencontrées 8
3.3 Comparaison avec Photoshop 10

Conclusion **11**

Introduction

Il arrive souvent de vouloir enlever des éléments de photos, que ce soient des fils électriques, des poteaux ou des touristes qui gâchent le paysage. L'inpainting permet de résoudre ces problèmes : cet algorithme supprime des zones d'une image et les comble de manière à ce que le trucage soit difficile à remarquer. Les applications sont nombreuses : restauration de photographies anciennes, photomontages, effets spéciaux, réalisations de panoramas...

Nous nous sommes intéressés à l'algorithme utilisant la correspondance d'imagettes, qui est la forme d'inpainting la plus utilisée aujourd'hui ; et nous nous sommes basés sur l'article de référence [1] sur cet algorithme, celui qui a introduit cette méthode. Celle-ci a remplacé celle se basant sur la résolution d'équations aux dérivées partielles [2]. Nous l'avons implémentée en C++ et l'avons ensuite testée sur différentes images.

1 Principe de l'algorithme

L'idée générale de l'algorithme est que pour combler un trou dans une image on peut uniquement se servir des pixels présents dans le reste de l'image. On compare les pixels à la frontière du trou à ceux du reste de l'image, et on les remplit par ceux qui correspondent le plus. L'ordre de ce remplissage est déterminant pour la qualité de l'image en sortie.

L'utilisateur sélectionne la frontière $\delta\Omega$ de la zone Ω à supprimer puis à remplir.
Une imagette est une petite zone de l'image, typiquement de dimension 9×9 pixels mais l'essentiel est qu'elle soit légèrement supérieur à la dimension maximale des éléments de texture distinguables de l'image. On note $\Psi_{\mathbf{p}}$ l'imagette centrée sur le pixel $\mathbf{p}$.

La région source Φ dans laquelle on va chercher les imagettes peut être définie soit comme l'image entière privée de Ω ou comme une portion de l'image ne contenant pas Ω.

À chaque pixel $\mathbf{p}$ est associé une couleur (qui peut être "vide" s'il n'est pas rempli) et une valeur de confiance $C(\mathbf{p})$ qui reflète la confiance que l'on a dans la couleur de notre pixel. Au cours de l'algorithme, on lui associe également une priorité $P(\mathbf{p})$ qui détermine l'ordre de remplissage. L'algorithme supprime les couleurs des pixels dans Ω, puis itère les trois étapes suivantes jusqu'à ce que toute l'image soit remplie.

1.1 Calculer les priorités des imagettes

Le résultat obtenu par l'algorithme dépend grandement de l'ordre de remplissage : il faut donc définir une priorité à associer à chaque pixel. La priorité est plus importante pour les pixels qui sont sur la continuité de lignes droites et qui sont entourés de pixels dont la confiance est haute. Pour toute imagette $\Psi_{\mathbf{p}}$ centrée en $\mathbf{p} \in \delta\Omega$, on définit sa priorité $P(\mathbf{p})$ comme étant :

$$P(\mathbf{p}) = C(\mathbf{p})D(\mathbf{p}) \tag{1}$$

où $C(\mathbf{p})$ est le terme de confiance et $D(\mathbf{p})$ est le terme de gradient.

Ils sont définis comme ceci :

$$C(\mathbf{p}) = \frac{\sum_{\mathbf{q} \in \Psi_\mathbf{p} \cap \Phi} C(\mathbf{q})}{|\Psi_\mathbf{p}|}, \quad D(\mathbf{p}) = \frac{|\nabla I_\mathbf{p}^\perp \cdot \mathbf{n}_\mathbf{p}|}{\alpha} \tag{2}$$

$|\Psi_\mathbf{p}|$ est l'aire de $\Psi_\mathbf{p}$. On initialise C de la manière suivante : $C(\mathbf{p}) = 0$ si $\mathbf{p} \in \Omega$ et $C(\mathbf{p}) = 1$ sinon. Le terme de confiance est une mesure de la proportion d'information sûr autour de $\mathbf{p}$. Le but est de remplir d'abord les imagettes qui ont une grande part de leurs pixels déjà remplis, et encore plus s'ils ont été remplis tôt ou qu'ils n'étaient pas dans la région à supprimer.

En ce qui concerne le terme de gradient, $\mathbf{n}_\mathbf{p}$ désigne la normale à la frontière $\delta\Omega$ et α est un facteur de normalisation. $\nabla I_\mathbf{p}^\perp$ est le vecteur indiquant la direction et l'intensité de l'isophote au point $\mathbf{p}$. Une isophote est une courbe qui relie les points de même luminosité. Ce terme favorise donc la priorité des imagettes par lesquelles des isopohotes passent, et encourage le prolongement des lignes.

1.2 Propagation de l'information

Une fois que toutes les priorités ont été actualisées, on considère l'imagette $\Psi_{\hat{\mathbf{p}}}$ de priorité maximale. On cherche maintenant à remplir ses pixels qui sont dans Ω par ceux de l'imagette extérieur à Ω la plus ressemblante. Formellement, on considère :

$$\Psi_{\hat{\mathbf{q}}} = \arg \min_{\Psi_\mathbf{q} \subset \Phi} d(\Psi_{\hat{\mathbf{p}}}, \Psi_\mathbf{q})$$

La distance $d(\Psi_\mathbf{b}, \Psi_\mathbf{b})$ entre les imagettes $\Psi_\mathbf{b}$ et $\Psi_\mathbf{b}$ est définie comme la somme des distances euclidiennes entre les pixels remplis correspondants des deux imagettes. Ces distances euclidiennes sont calculées dans un espace de couleur. Nous avons suivi les indications de l'article de référence [1] nous avons représenté les pixels dans l'espace des couleurs CIE LAB, ce qui permet d'avoir de meilleurs propriétés que dans l'espace RGB. Intuitivement, plus la distance entre deux imagettes est faible, plus celles-ci se ressemblent visuellement.

Maintenant que l'on dispose de $\Psi_{\hat{\mathbf{q}}}$, on remplit chaque pixel vide $\mathbf{p} \in \Psi_{\hat{\mathbf{p}}} \cap \Omega$ par le pixel correspondant dans $\Psi_{\hat{\mathbf{q}}}$.

1.3 Actualisation des valeurs de confiance

L'imagette centrée en $\hat{\mathbf{p}}$ est entièrement traitée ; il faut maintenant actualiser les valeurs de confiance des pixels sur la frontière de Ω qui viennent d'être remplis. Intuitivement, ils ont été modifiés à la même itération que $\hat{\mathbf{p}}$ et leur données proviennent de la même imagette source : il paraît cohérent de leur donner la même confiance qu'à $\hat{\mathbf{p}}$. On effectue alors :

$$\forall \mathbf{p} \in \Psi_{\hat{\mathbf{p}}} \cap \Omega, C(\mathbf{p}) = C(\hat{\mathbf{p}})$$

Les valeurs de confiance diminue donc au fil des itérations, indiquant que nous sommes moins sûrs des couleurs des pixels au centre de la région à remplir initialement.

1.4 L'algorithme

Les étapes précédentes sont répétées jusqu'à ce que toute l'image soit remplie. On obtient donc le pseudo-code de l'algorithme d'inpainting :

Ω^0 : **domaine initial à supprimer, de frontière $\delta\Omega^0$ sélectionnée par l'utilisateur.**
$t = 0$: **itération en cours.**
Répéter
1. Identifier la frontière $\delta\Omega^t$ et calculer les priorités $P(\mathbf{p})$ pour tout $\mathbf{p} \in \delta\Omega^t$.
2. Trouver l'imagette $\Psi_{\hat{\mathbf{p}}}$ qui maximise la priorité, puis trouver l'imagette $\Psi_{\hat{\mathbf{q}}}$ de distance minimale avec $\Psi_{\hat{\mathbf{p}}}$. Copier les données de $\Psi_{\hat{\mathbf{q}}}$ vers $\Psi_{\hat{\mathbf{p}}}$, aux pixels $\mathbf{p} \in \Omega^t \cap \Psi_{\hat{\mathbf{p}}}$.
3. Actualiser $C(\mathbf{p})$ pour tout $\mathbf{p} \in \delta\Omega^t \cap \Psi_{\hat{\mathbf{p}}}$.
4. Actualiser le domaine à considérer avant de passer à l'itération $t+1$: $\Omega^{t+1} = \Omega^t \setminus \Psi_{\hat{\mathbf{p}}}$.
jusqu'à ce que $(\Omega^t = \emptyset)$

Algorithme 1: Algorithme d'inpainting

2 Implémentation

2.1 Le code

Nous avons implémenté cet algorithme en C++ en suivant les étapes décrites précédemment. Nous avons défini la zone source Φ comme une certaine proportion de l'image autour de Ω. Nous avons également implémenté un "balayage" qui permet de réduire le temps de calcul. Au lieu de regarder les imagettes centrées en tous les pixels, on saut de pixel en pixel avec un pas fixé. Plus ce pas est grand, plus le trucage est grossier.

2.2 L'interface graphique

Nous avons élaboré et amélioré le code en utilisant une méthode de sélection assez simple : l'utilisateur sélectionne des points pour former un polygone qui délimite le domaine Ω. En parallèle nous avons implémenté une interface plus complète, avec plus d'options et d'outils de sélection. Il est en effet plus intéressant de disposer d'outils particuliers pour certaines images. De plus, plutôt que d'effectuer plusieurs fois des opérations d'inpainting, il est plus logique de supprimer toutes les zones en même temps : ainsi, on évitera de reconstituer la nouvelle image à partir d'imagettes que l'on supprimera par la suite.

Différentes fonctions ont été implémentées. La fenêtre dispose d'un zoom dynamique, qui permet de cibler des zones de l'image précisément en appuyant sur les touches + ou - du pavé numérique. En plus de cela, des barres de défilement ont été ajoutées afin de déplacer l'image sur l'écran si celle-ci a des dimensions supérieures à celles de la zone d'affichage.

Pour la sélection à proprement parler plusieurs outils sont disponibles, leur description s'affichant lorsque l'on passe sur le bouton associé avec la souris. On dispose tout d'abord d'une sélection rectangulaire, mais également d'une sélection circulaire, et enfin d'une sélection polygonale. Une gomme est également disponible. On peut faire varier sa taille à l'aide du curseur situé sous les boutons. Elle permet de désélectionner des zones déjà sélectionnées au préalable. Bien sûr, il est possible d'utiliser ces fonctions successivement.

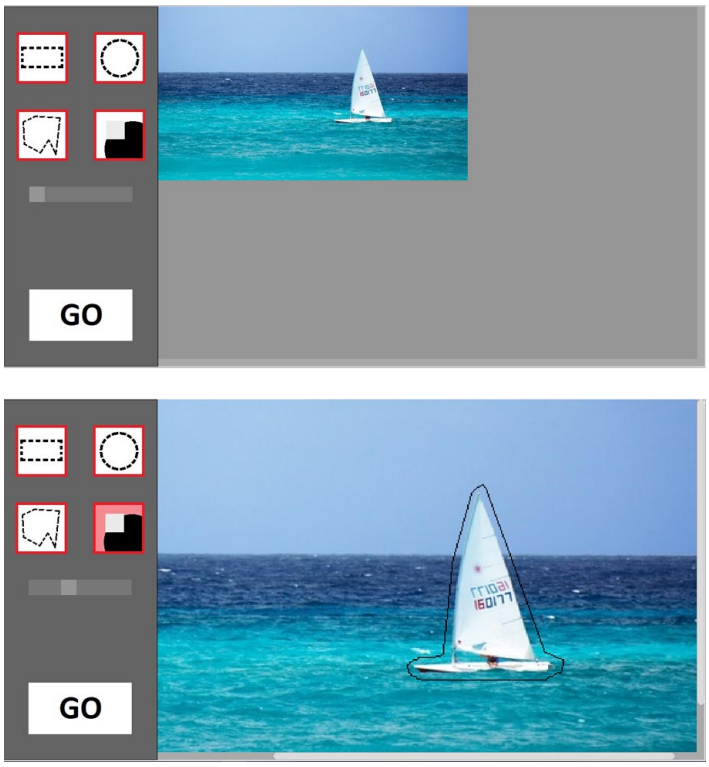

FIGURE 2 – Au-dessus, interface au chargement de l'image. En-dessous l'image a été zoomée, déplacée, le bateau a été sélectionné à l'aide des différents outils et la gomme est en cours d'utilisation.

La zone sélectionnée s'affiche directement sur l'écran, la frontière étant représentée par un trait noir. Lorsque la sélection est terminée, il suffit d'appuyer sur le bouton "GO" pour que l'algorithme d'inpainting démarre et modifie la zone sélectionnée.

2.3 La parallélisation

Afin de réduire le plus possible le temps d'exécution, nous avons utilisé du calcul parallèle. Cette technique permet de répartir les calculs à effectuer sur les différents cœurs du processeur et ainsi diviser par deux ou par quatre le temps d'exécution. Nous avons parallélisé les fonctions permettant de calculer les priorités et déterminer l'imagette de priorité maximale, celle permettant de trouver l'imagette de distance minimale avec celle à remplir, et celle permettant déterminer le plus petit rectangle contenant Ω^0 à l'initialisation (qui permet de réduire la taille des domaines à parcourir ensuite).

Nous avons choisi ces fonctions car ce sont celles qui effectuent le plus de boucles, et de loin. Les autres fonctions n'agissent que sur une imagette à la fois. On ne peut pas non plus paralléliser les boucles de plus haut niveau qui appellent ces fonctions, car les données sont actualisées à chaque itération et il faut que les tâches puissent être effectuées de manière indépendante pour paralléliser. En effet, dans le cas contraire les "threads" pourraient rentrer en conflit et les opérations ne seraient plus effectuées dans l'ordre souhaité.

On observe bien avec la parallélisation que le temps d'exécution est divisé par deux ou par quatre, suivant la machine utilisée. On arrive finalement à un temps d'exécution autour de 30 secondes typiquement pour des images de moyenne définition, ce qui est satisfaisant.

3 Résultats obtenus

3.1 Exemples

Nous avons testé notre implémentation de l'algorithme sur différentes images. Pour des images de moyenne définition, le temps de calcul est satisfaisant. Pour des images en haute définition, le temps de calcul est par contre beaucoup plus long même pour des zones Ω de petite taille, et le code peut prendre plusieurs minutes à calculer l'image en sortie.

Les images originales sont à gauche et les images obtenues avec l'algorithme sont à droite.

FIGURE 3 – Grande zone à supprimer sur fond uni

Comme on pouvait s'y attendre, l'algorithme est particulièrement efficace pour éliminer des zones sur un arrière-plan uni. Les frontières entre les éléments de l'image sont claires, et même avec un balayage élevé le trucage est bien réussi.

FIGURE 4 – Zone à supprimer passant par-dessus un autre élément

Sur cet exemple, le panneau de signalisation passe devant le chien. Mais la méthode de correspondance d'imagettes permet de reconstituer la partie cachée du chien.

FIGURE 5 – Suppression de logo

L'algorithme d'inpainting permet également de supprimer les écritures et les logos sur les images de manière efficace, ce qui se voit bien sur cet exemple. Pour obtenir ce résultat, nous avons utilisé plusieurs fois l'algorithme d'affiler pour effacer une lettre après l'autre, ce qui est donne de meilleurs résultats lorsque l'on cherche à supprimer des éléments disjoints dans l'image.

FIGURE 6 – Plusieurs utilisations successives de l'algorithme

C'est ce que nous avons également fait sur cette photo pour supprimer les bateaux. On peut remarquer que lorsque le fond n'est pas clairement uni, on obtient plus facilement des artefacts non désirés et le trucage se voit plus.

FIGURE 7 – Photo haute définition

Pour cet exemple nous avons utilisé une photographie en haute définition. L'algorithme fonctionne de manière très satisfaisante, mais n'est vraiment utilisable qu'en sélectionnant des petites zones de l'image. Il a déjà mis plusieurs minutes pour obtenir l'image en sortie dans ce cas, et le temps de calcul est bien plus long en sélectionnant une grande zone au départ.

3.2 Difficultés rencontrées

Notre code permet d'obtenir des résultats satisfaisants la plupart du temps, mais il possède quelques limites. Tout d'abord au niveau du temps de calcul. Comme dit précédemment, pour des images haute définition le temps de calcul est trop long pour une utilisation confortable. Par ailleurs, notre code donne de moins bons résultats dans certaines situations.

FIGURE 8 – Bon résultat pour la partie sur fond bleu, mais trucage visible au niveau de la base

Sur cet exemple on voit que même si la tour Eiffel a bien été supprimée et remplacée par du ciel bleu, le trucage est visible au niveau de la base. Dans notre implémentation, la zone Φ est fixée comme une certaine proportion de l'image autour de Ω. Ainsi, au niveau de la base l'algorithme va chercher les imagettes les plus ressemblantes autour de la base et pas seulement dans le ciel bleu. Comme il n'y a pas de démarcation claire à favoriser, l'algorithme va prolonger les arbres ou les bâtiments derrière ce qui donne ce résultat.

FIGURE 9 – Prolongement de la manche

Cela se voit encore plus sur cet exemple. Le champ de fleurs en arrière-plan n'a pas été un problème pour notre code. Ce qui a causé des difficultés est, comme précédemment, le fait que l'on cherche les imagettes dans une portion définie de l'image. On cherche à avoir une cassure brute au niveau de la manche droite de la femme, mais en fait l'algorithme va plutôt avoir tendance à prolonger cette manche ce qui donne ce résultat.

3.3 Comparaison avec Photoshop

Les logiciels de retouche numériques comme Photoshop possèdent des outils de correction par inpainting. Nous avons alors comparé nos résultats à ceux obtenus en utilisant Photoshop, afin d'estimer l'efficacité de notre implémentation.

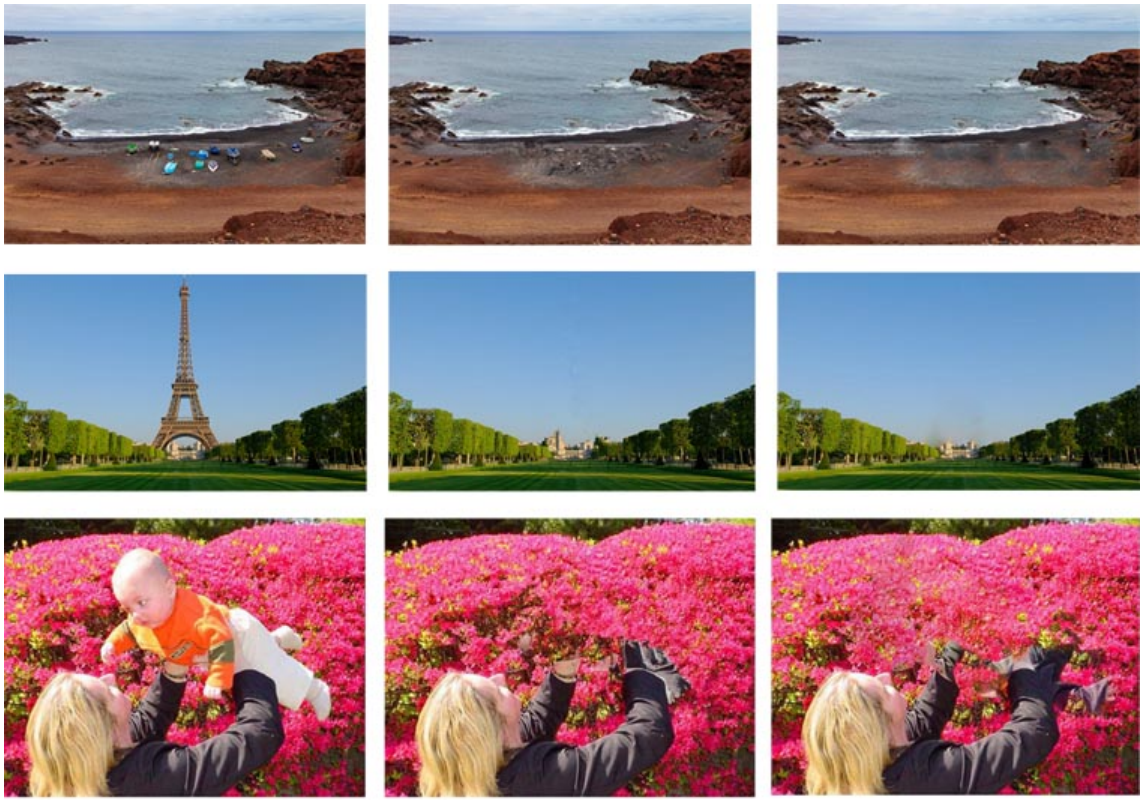

FIGURE 10 – Comparaison entre l'image originale, le trucage obtenu avec notre code et le trucage obtenu avec Photoshop (de gauche à droite)

Tout d'abord la vitesse d'exécution est bien plus faible en utilisant Photoshop qu'en utilisant notre code. Même pour des images haute définition, l'outil de Photoshop ne prend que quelques secondes pour modifier l'image. Mais le contraire aurait été étonnant pour un logiciel commercial, d'autant plus pour un logiciel aussi populaire que Photoshop.

Par ailleurs les résultats obtenus ne diffèrent pas grandement : notre code permet d'obtenir des trucages de qualité comparable à ceux obtenus avec Photoshop. L'effet de flou que l'on trouve dans les trucages par Photoshop suggère que ce logiciel utilise d'autre techniques de retouches en plus de l'inpainting. Quand on utilise l'outil dédié, la zone modifiée n'est pas uniquement constituée de pixels de l'image : d'autres effets semblent avoir été appliqués.

De plus, le prolongement de la manche est aussi là avec Photoshop ce qui suggère que ce problème est inhérent à l'algorithme et n'est pas uniquement du à notre implémentation.

Conclusion

Nous avons ainsi implémenté l'algorithme d'inpainting avec succès. Les résultats que l'on obtient sont très acceptables, même si ce n'est pas une méthode miracle et le trucage peut être flagrant suivant les cas. La principale différence avec les logiciels commerciaux est le temps d'exécution qui reste long pour des images haute définition, même en utilisant la parallélisation.

Pour améliorer notre code, on pourrait laisser le choix à l'utilisateur de sélectionner la zone Φ en plus de Ω, ainsi que trouver une méthode pour adapter la taille des imagettes automatiquement suivant l'image en entrée.

Références

[1] A. Criminisi, P. Perez, et K. Toyama. Region Filling and Object Removal by Exemplar-Based Image Inpainting. In *IEEE Transactions on Image Processing*, 13(9), 2004.

[2] M. Bertalmio, L. Vese, G. Sapiro, et S. Osher. Simultaneous structure and texture image inpainting. In *Proc. Conf. Comp. Vision Pattern Rec.*, Madison, WI, 2003.