Projet IMA201 : Inpainting

Gabriel Françon et Anselme Donato

Le but du projet est de coder un algorithme pour effacer des objets dans une image numérique. Le défi consiste à combler le vide laissé par l'objet de manière plausible par rapport au reste de l'image. Habituellement, le problème est traité de deux façons différentes : certains algorithmes utilisent de la synthèse de texture à partir du reste de l'image alors que d'autres utilisent des techniques dites d' "inpainting", plus efficaces pour reconstruire des structures. L'algorithme proposé mélange les deux méthodes pour obtenir un résultat optimal. L'idée est de choisir à chaque itération un "patch cible" (target_patch) situé à la frontière de la zone à remplir (donc partiellement rempli de pixels de l'image). En comparant les pixels connus du patch aux autres pixels de l'image, on détermine le patch source (source_patch) le plus ressemblant possible, et on vient compléter le target_patch avec les pixels du patch_source. Petit à petit, on peut donc remplir le vide.

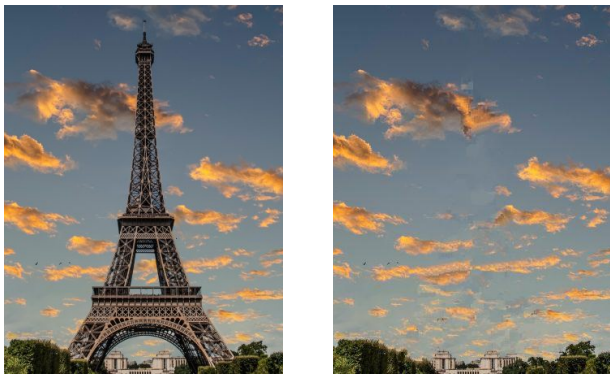


Fig. 1 : ***exemple d'inpainting obtenu avec l'algorithme*** *(Gauche : image originale. Droite : la région correspondant à la tour Eiffel a été sélectionnée et automatiquement enlevée)*

Principe d'exécution

Un des points d'attention majeurs de cette méthode est l'ordre de remplissage. En fonction de l'ordre dans lequel on choisit les patchs cibles, on peut avoir des résultats très différents. On attribue donc une priorité de traitement à chaque pixel.

- détecter le contour de la zone à remplir, en déterminant le front du masque
- mettre à jour la priorité de chacun de ses pixels. Cette priorité est basée sur deux termes : le terme de confiance et le terme de donnée
- Extraire le target_patch centré sur le pixel le plus prioritaire
- trouver, dans une zone de recherche prédéfinie, le source_patch qui lui convient le mieux (c'est à dire celui qui est le moins "loin" visuellement)
- compléter le target_patch avec les donnés du source_patch retenu
- mettre à jour la confiance accordée aux nouveaux pixels du target_patch

Déterminer le front du masque

L'algorithme remplit les patchs en commençant par ceux situés sur le contour de la zone cible, puisque évidemment pour comparer le patch cible au reste de l'image, il faut que ce patch cible soit en partie rempli. On doit donc détecter le contour de la zone à remplir. Pour cela, deux méthodes: soit utiliser le Laplacien, soit faire une dilatation.

Le Laplacien permet de repérer les discontinuités de l'image. Comme dans notre image masque il n'y a qu'une seule discontinuité, à savoir le passage du noir au blanc à la frontière de la zone cible, le Laplacien pourrait en théorie nous permettre de bien repérer cette frontière. Toutefois, en pratique il y a un problème: ce Laplacien n'est pas forcément très précis, et surtout peut aboutir sur une frontière discontinue, ce qui n'est pas idéal. Sur les conseils de notre encadrant, nous avons donc changé d'approche et nous nous sommes basés sur une dilatation (binaire) du masque.

En utilisant un carré 3x3 comme élément structurant, on peut dilater le masque en "épaississant" sa bordure. Ensuite, on n'a plus qu'à retirer à ce masque dilaté le masque d'origine, et on obtient une zone frontière précise et continue. Cette approche est également plus flexible, puisqu'on peut ajuster l'élément structurant si besoin est, alors qu'on n'a pas vraiment de prise sur le Laplacien. A noter qu'avec cette méthode, la frontière est à l'extérieur de la zone à remplir, puisqu'on a épaissit le contour. Les centres des patchs à remplir, qui sont situés sur la frontière, seront donc toujours connus.

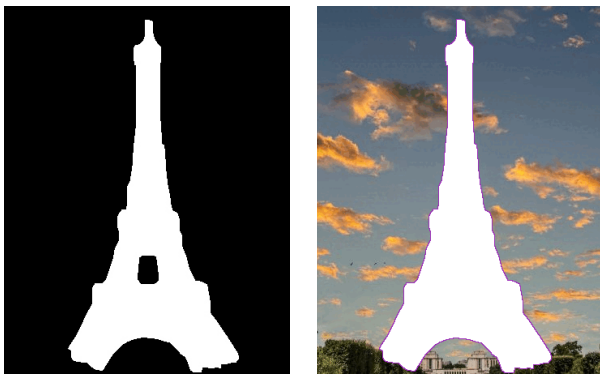


Fig. 2 : ***exemple de masque (blanc) avec son contour (violet)*** *(Gauche : le masque correspondant à l'image de la tour Eiffel de la Fig.1. Droite : le masque et son contour, en violet, superposé à l'image pour plus de visibilité)*

Calcul de la normale

Le calcul des normales au contour de la zone à remplir est nécessaire pour mettre à jour le terme de données D(p) à chaque itération. Une bonne façon de le faire est de calculer le gradient du masque binaire dans les directions x et y (par exemple avec un filtre de Sobel comme c'est le cas dans notre code). On obtient ainsi gradX et gradY les valeurs du gradient dans chaque direction, et on peut déterminer pour chaque pixel du contour le sens de la normale au masque et la normaliser avec la formule suivante :

$$n_p = \left(\frac{-gradY_p}{\|grad_p\|}, \frac{gradX_p}{\|grad_p\|} \right)$$

Calcul de l'isophote et problèmes liés aux zones très texturées

Pour l'isophote également il est nécessaire de calculer un gradient, mais de l'image cette fois-ci. Le gradient étant orthogonal aux contours de l'image, on prend donc en chaque point le perpendiculaire.
Mais on ne peut pas simplement choisir le gradient au niveau du contour de masque puisque sa valeur et sa direction seraient biaisées en raison justement de l'ajout de ce contour artificiel. Il faut donc associer à chaque pixel du fill front un isophote non biaisé et qui reflète quand même la présence de contours dans le voisinage de ce pixel.

Pour ce faire, on effectue les opérations suivantes pour chaque pixel cible du fill_font :

- On commence par créer une dilatation du masque et on met à 0 le gradient des pixels dont la position est contenue dans ce masque (élément structurant de taille 15x15, valeurs obtenues par l'expérience pour s'assurer que les gradients biaisés sont bien annulés).
- Ensuite, on récupère le pixel hors de ce masque qui se situe le plus proche du pixel cible.
- On récupère également le patch associé à celui-ci, et on évalue le gradient de plus grande norme parmi les pixels du patch.
- C'est ce gradient qui servira à constituer l'isophote associé : Ip=(-gradYA(p), gradXA(p)) avec A(p)la fonction qui correspond au processus de sélection décrit ci-dessus.

Ce procédé marche bien sur des images lisses, mais la mesure du gradient est bien moins représentative des contours sur des images plus texturées (les directions partent rapidement dans tous les sens). Pour pallier cela, on fait le choix de pré-traiter l'image avec un filtre passe-bas type gaussien qui adoucit les textures et ne conserve que les contours les plus importants.

Quel terme privilégier dans le calcul de la priorité

L'ordre de remplissage dépend de deux paramètres: le terme de confiance, qui reflète la connaissance qu'on a déjà du contenu du pixel, et le terme de data, qui est là pour préserver les structures géométriques. Les deux termes sont donc complémentaires, et ont chacun un rôle important. Le problème est qu'il y a un risque qu'on favorise d'abord le remplissage des patchs avec un terme de data élevé, sans s'occuper de leur terme de confiance. On peut donc se retrouver à traiter en priorité des patchs sur lesquels on ne sait pratiquement rien, donc des patchs dont le remplissage est hasardeux. Or comme l'algorithme est glouton, un patch mal rempli a des répercussions sur tout le reste du travail. Nous reviendrons sur cette limite dans la suite.
A l'inverse, si on a tendance à ne pas assez considérer les pixels à fort terme de data, on peut se retrouver à remplir d'abord des zones texturées, qui peuvent "déborder" et finir par masquer des structures. Il faut donc réussir à trouver un juste équilibre entre les deux.

L'enjeu de la détermination de la zone de recherche

Contraindre la zone de recherche de patch similaire permet de considérablement réduire le temps de calcul: l'algorithme n'a pas à regarder tous les pixels de l'image mais seulement une petite portion. Beaucoup plus efficace. De plus, il paraît logique de limiter la zone de recherche aux environs de la zone à remplir, qui sont normalement les plus semblables. Il serait bizarre que le programme comble un trou avec un patch copié à l'autre bout de l'image, qui a beaucoup plus de chance de ne pas ressembler à la zone qui l'entoure.

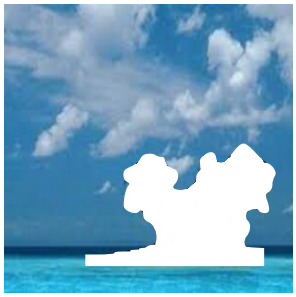


Fig. 3 : ***réduction de la zone de recherche***
(Haut : image originale. Bas gauche : zone de recherche non limitée. On peut utiliser tout ce qui n'est pas dans la zone à remplir, en blanc. Bas droite : zone de recherche limitée au voisinage du masque à l'aide d'une dilatation.)

Choix de la distance

La réussite de l'algorithme repose en partie sur sa capacité à reconnaître les patch qui "se ressemblent" le plus pour choisir au mieux les pixels sources qui serviront à remplir le patch cible en question. Pour ce faire, la technique la plus simple et qui est proposé dans l'article est de choisir comme terme à minimiser la somme des différences au carré entre chaque pixel du patch cible et le pixel à la même position dans le patch source candidat (on ne considère évidemment que les pixels déjà connus du patch cible) :

$$argmin_{q \in zone\ source} \sum_{i=1}^{|N_q|} [(N_p)_i - (N_q)_i]^2$$

*(**Formule du choix du pixel source**, qui est donc celui qui minimise la distance)*

Cette méthode est efficace pour des images en niveau de gris et qui possèdent des zones de texture très lisses. En revanche, lorsqu'on travaille sur des images de couleur, il n'existe plus vraiment de relation d'ordre. Cela revient à calculer l'écart de rouge, de vert et de bleu entre deux pixels. Or la distance entre par exemple deux pixels de canaux R et G identiques mais de canaux B assez éloignés a des chances d'être inférieure à celle entre deux pixels qui diffèrent légèrement sur leurs trois canaux. Et il arrive néanmoins que ce deuxième pixel ressemble visuellement plus au pixel cible, faussant ainsi le choix du meilleur source_patch.
Un première amélioration a été de convertir le target_patch et le source_patch en HSV avant de les comparer, sur conseil de notre encadrant. Les résultats ont effectivement été plus cohérents, bien qu'il soit difficile de bien comprendre pourquoi. On peut néanmoins imaginer que ramener la comparaison de la teinte à celle entre deux termes seulement (au lieu de six) évite de se retrouver avec des teintes opposées patchées côte à côte dans la région cible.

Une deuxième amélioration, en théorie, aurait été de ne pas seulement comparer les couleurs mais aussi les textures qui entourent les deux pixels à comparer. Le deuxième article évoque notamment un opérateur se basant sur le calcul du module du gradient local pour donner une valeur de texture au pixel en question. Pour des textures bien représentées par le module de leur gradient, cet indicateur permettrait ainsi de détecter deux patch correspondant à des textures similaires. En pratique ce deuxième indicateur n'a pas toujours été bénéfique, il avait même tendance à dégrader la qualité du résultat en favorisant des patch de couleur très différente de celle du patch cible. Nous avons donc décidé de ne pas le conserver.

Complexité

L'algorithme est assez lourd à s'exécuter dès que l'on souhaite inpainter une zone importante de l'image, et ce d'autant plus lorsque l'image en question est de haute résolution. Pour mieux cerner l'influence des différents paramètres sur la complexité de l'algorithme, on va établir son expression :

- Image de taille (h, w)
- e épaisseur de la région source (environ 4n.e pixels dans la région source)
- n^2 nombre de pixels de la région cible (approximé comme un rectangle de taille n)
- t nombre de pixels dans un patch (en moyenne ⅕ des pixels non connus pour ⅘ de pixels connus)
- 4n pixels sur le contour du masque, que l'on parcourt pour mettre à jour la confiance (somme pondérée sur ⅘ t pixels) puis le terme de data (2x calcul du gradient sur hxw pixels, puis on prend le max des isophotes dans chaque patch du contour et on calcule le produit scalaire donc encore des calculs sur ⅘ t pixels)
- Recherche du pixel le plus prioritaire, test sur 4n pixels
- Recherche du patch source en itérant sur les m pixels de la région source (on doit à chaque fois calculer la distance au patch cible, soit ⅘ t opérations)
- Affectation des nouvelles valeurs au ⅕ t pixels non connus du patch

- Même processus pendant k itérations avec $k = 5n^2/t$

On obtient donc une complexité de :

$$C = \frac{5n^2}{t}[4n(1 + 2h.w)\,⅘\,t + 4n + ⅘\,4n.e.t + ⅕\,t)]$$

$$\simeq 16n^3(e + 2h.w)$$

Comme on le voit, augmenter la taille des patch ne change pas fondamentalement la complexité de l'algorithme, qui augmente très rapidement en complexité lorsque l'on agrandit la taille du masque (en pratique donc quand on augmente la résolution de l'image puisqu'il faudra plus de pixels à retirer pour un objet de "même taille"). La complexité est polynomiale.

Concernant les patch, on peut donc choisir la taille idéale pour améliorer le résultat :
-une taille trop grande donnera un résultat brouillon et ne pourra pas reproduire les plus petits détails détails
-une taille trop petite risque de ne pas être représentatif du voisinage du pixel cible et donc choisir un patch source qui ne correspond pas assez au voisinage du pixel cible

Problèmes inhérents à l'algorithme

L'algorithme proposé est un algorithme "glouton", c'est à dire qu'il est basé sur le principe qu'un résultat global optimal s'obtient en sommant des résultats locaux optimaux. Dans notre cas, on fait le pari que remplacer de façon réaliste une zone par rapport au reste de l'image peut se faire en remplaçant une poignée de pixels de façon cohérente dans une zone locale de l'image (le patch). Ce principe permet une implémentation facile, mais ne laisse pas la place à la correction d'erreur. En effet, il n'y a pas de boucle de rétroaction : on considère que les décisions qu'on a prises dans le passé sont bonnes. Une erreur faite au début du processus va donc se propager durant toute la suite, et quelques pixels mal remplis peuvent donner lieu à toute une zone incohérente, qui sera d'autant plus étendue si le mauvais remplissage est effectué tôt.

Correction de l'algorithme

L'algorithme utilisé est basé sur le premier article que nous avons eu) lire. Il a cependant ses limitations, et dans le but de l'améliorer nous avons reçu un second article sur le sujet. L'objectif général de ce second article proposé est de "lisser" la fonction shift_map obtenue avec le premier algorithme. On souhaite vérifier si pour chaque pixel cible, la décision aurait pu être meilleure dans le choix du pixel source en se basant sur les décisions prises pour les pixels avoisinant le pixel cible en question.

L'algorithme a été implémenté partiellement (ANN search par PatchMatch et reconstruction) en fin de projet mais n'a pas, à ce jour, prouvé son utilité en tant que "correcteur" du premier remplissage. Il semble en effet que cette correction ait plus de sens quand l'initialisation de la fonction shift_map a été réalisée avec moins de certitudes (aléatoirement, par une interpolation de Laplace ou en onion peel).
Ceci étant, nous n'avons pas passé assez de temps sur ce code pour le rendre fonctionnel et complet, et ne pouvons donc pas en tirer les conclusions.
Toutefois, l'idée de cet article concernant le traitement multi-échelle nous apparaît intéressante à considérer pour de futures améliorations de l'inpainter. De cette façon, on reconstruit l'image de plusieurs façons et selon des critères différents (contours généraux lorsque l'image est très sous-échantillonnée, détails précis lorsqu'elle est sur-échantillonnée) et on risque moins de se retrouver face à de grosses aberrations de reconstruction.

Pour finir, voici quelques exemples de résultats de notre implémentation de l'inpainting (gauche : image originale. Droite : image inpaintée) Les paramètres utilisés sont à la fin.

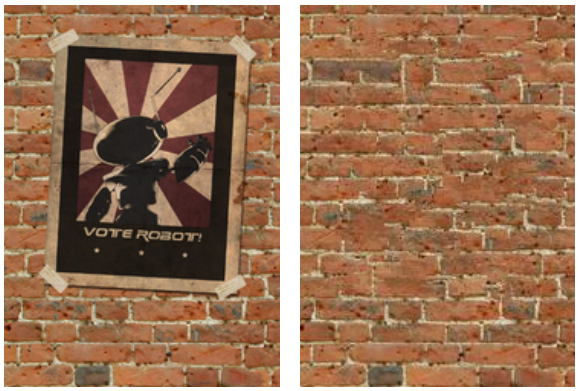
Baseball

Guys

Panel

Bike

Island



Wall

Paramètres utilisés

1. Image // 2. Taille du patch (en pixels) // 3. Epaisseur de la région source (en pixels) // 4. Nombre d'itérations // 5. Temps de calcul en secondes

1	2	3	4	6
Baseball	4x4	4	489	420
Guys	9x9	4	309	223
Panel	10x10	3	299	107
Bike	8x8	4	1867	679
Island	9x9	6	795	790
Wall	9x9	5	1165	903

Remerciements :

Merci à toute l'équipe enseignante d'IMA 201, et tout spécialement à notre encadrant M. Newson.

Référence :

Ce travail est basé sur la publication Region Filling and Object Removal by Exemplar-Based Image Inpainting, de A. Criminisi, P. Pérez et K. Toyama, Microsoft Research, Cambridge (UK) et Redmond (US), publié en 2004.