Anselme Donato - Gabriel Françon

Projet Inpainting

Rapport de mi-parcours

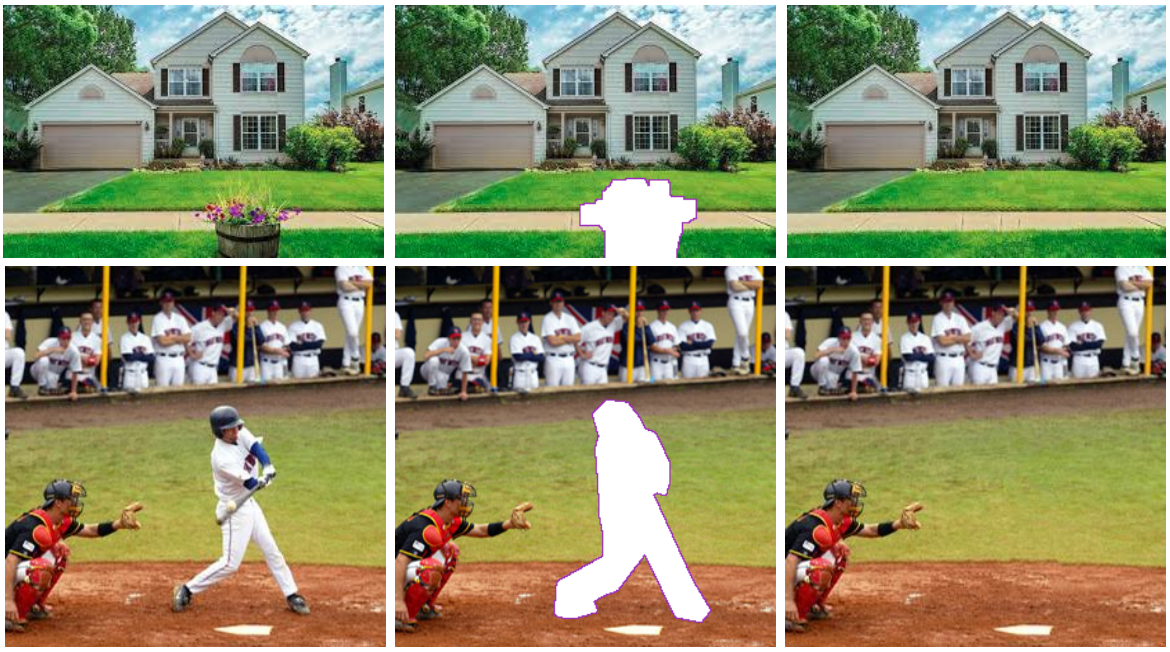
Le but du projet est de coder un algorithme pour effacer des objets dans une image numérique. Le défi consiste à combler le vide laissé par l'objet de manière plausible par rapport au reste de l'image. Habituellement, le problème est traité de deux façons différentes : certains algorithmes utilisent 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 sélectionner 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. 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.

Pour implémenter cet algorithme, le choix du langage **Python** s'est rapidement imposée, tant la librairie **numpy** permet une manipulation facile des tenseurs - en l'occurrence ici des images - et offre des méthodes optimisées pour réduire les temps de calcul, qui deviennent rapidement très grand en traitement d'image. En entrée, on récupère l'image à traiter ainsi qu'une image binaire qui sert de masque de repère pour la zone de l'image à effacer puis remplir. Pour ce faire, l'utilisateur passe en argument leur chemin respectif, qui sont ensuite parsés dans la fonction main. Les fichiers images sont enfin extraits via la librairie ***imageio*** sous forme de matrice de pixels. Il était important de pouvoir accéder et manipuler directement les pixels, l'inpainting se résumant en gros à choisir des pixels dans une zone de l'image pour les recopier dans une autre.

Ensuite, la fonction main appelle la classe « Inpainter » dans laquelle sont implémentées les étapes suivantes que l'on répète jusqu'à avoir totalement rempli l'image :

1. détecter le contour de la zone à remplir
2. mettre à jour la priorité de chacun de ses pixels, et extraire le target_patch centré sur le pixel le plus prioritaire
3. trouver, dans une zone de recherche prédéfinie, le source_patch qui lui convient le mieux
4. compléter le target_patch avec les donnés du source_patch retenu
5. mettre à jour la confiance accordée aux nouveaux pixels du target_patch

A ce jour, nous avons écrit un code fonctionnel qui exécute presque entièrement toutes ces étapes. Les résultats obtenus, dont quelques-uns sont montrés ci-dessous, sont déjà globalement satisfaisants.



L'article laisse des libertés concernant la réalisation de certaines de ces étapes, nous avons donc procédé à plusieurs choix d'implémentation :

- Pour la détection du contour, nous avons d'abord opté pour le calcul du laplacien du masque, mais celui-ci ne permettait pas toujours d'obtenir un contour continu. Nous avons donc plutôt choisi de réaliser une légère dilatation du masque avant de lui soustraire le masque d'origine. (on obtient alors un « masque de contour »).

- Pour mettre à jour la priorité des pixels du contour, il faut calculer deux choses : un terme de confiance C(p) lié à la sûreté que l'on accorde aux valeurs des pixels voisins, et un terme de donnés D(p) qui reflète le fait que le pixel en question se situe ou non sur un contour de l'image (par exemple une ligne qui doit se prolonger dans le masque). Pour le moment et en suivant les directives de notre encadrant, seul le terme de confiance est pris en compte (c'est celui qui a la plus grosse influence). Le calcul de D(p) est plus délicat et fera l'objet de nos prochaines séances de travail.

- L'une des choses qui ralentissait le plus le programme était la recherche du source_patch. Pour pallier cela, il fallait donc réduire la zone de recherche. Tout d'abord, il a rapidement été choisi de ne prendre les pixels que parmi la zone connue initiale pour éviter de favoriser la propagation d'erreurs. Ensuite, plusieurs options ont été étudiées, mais la plus évidente - et la plus concluante - a été de délimiter la zone de recherche comme étant une dilatation de la zone à remplir. Cette restriction n'a pas impacté la qualité du résultat, au contraire : les source_patch choisis se trouvaient déjà presque toujours proche du masque, et on évite ainsi de choisir un patch lointain qui a bien moins de chance de correspondre visuellement. Le calcul du masque de la région source ne s'effectue qu'une fois au début du programme. Ensuite, pour chaque target_patch, on ne s'intéresse qu'aux sources_patch contenus dans ce nouveau masque et se trouvant à une distance inférieure à une valeur fixée (de cette façon on ne va pas chercher un patch de l'autre côté de la zone à remplir). Deux paramètres existent donc pour fixer la région source : son épaisseur et la distance maximale tolérée entre le target_patch et le source_patch. Ces paramètres changent considérablement les résultats obtenus, mais doivent être adaptés à chaque image pour optimiser son traitement (l'épaisseur est généralement de 4 à 6 fois la largeur des patch, et la distance va de 40 à 100 pixels).

- Un autre paramètre déterminant pour la qualité du rendu est la taille des patch. L'article fait mention de patch de 9x9 pixels, ce qui correspond selon leur dire à un peu plus que la plus petite texture visible sur un écran classique. On pourrait croire qu'augmenter la taille des patch permettrait de réduire le temps de calcul. En réalité, cela aurait pour conséquence d'augmenter la complexité du calcul de la différence de texture entre le target_patch et chaque source_patch. Or quelque soit la taille du patch, le nombre de pixels testés et donc de patch reste le même (il est même généralement plus grand car on est obligé d'augmenter l'épaisseur de la région source pour conserver suffisamment de patch différent visuellement). Mais si on réduit trop la taille, on prend moins en compte le voisinage de chaque pixel et on rend le choix du source_patch plus aléatoire. Après plusieurs tests, une taille de 9 semble donc être un bon compromis pour la plupart des images.

- Enfin, il faut s'intéresser à la façon dont on estime numériquement que deux patch sont similaires ou non. Dans l'article, ils précisent qu'il faut calculer la somme des différences au carré pixel à pixel. Cette méthode est efficace pour des images en niveau de gris et qui possédent 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 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. La solution retenue a été de convertir le target_patch et le source_patch en HSV avant de les comparer, sur conseil de notre encadrant. Et les résultats ont effectivement été bien plus cohérents.

Avec la réduction de la zone de recherche et l'utilisation de numpy notamment pour les dilatations de masque, le temps de calcul pour une image de quelques centaines de milliers de pixels (300x300, 720x480, etc) est passé de plus d'une heure à quelques minutes. Il faut compter entre 400 et 1000 itérations selon la taille de la zone de l'image à remplir, et chacune s'effectue en une demi seconde environ. Cette complexité est satisfaisante et nous permet maintenant d'effectuer des tests sur des images de meilleure qualité.