Informatique Graphique II

2- Traitement d'images

Michel Beaudouin-Lafon, mbl@lri.fr

source : Foley & van Dam pp 815-843, p753, Gonzalez & Woods

Introduction

Jusqu'à présent, nous avons traité exclusivement d'images de synthèse, c'est-à-dire d'image produites à partir d'une description abstraite et dont on peut (théoriquement) connaître la valeur en tout point (de coordonnées réelles). Les algorithmes de rastérisation pemettent de calculer des valeurs échantillonnées qui approchent l'image "réelle". Dans le cours précédent, on a vu l'utilisation de techniques de traitement du signal (filtrage par convolution) pour réaliser l'anti-aliassage, c'est-à-dire une meilleure approximation de l'image échantillonnée que celle donnée par les algorithmes de rastérisation simples.

Les images numérisées sont aussi très utilisées en infographie. Une image numérisée est une image dont on connaît les valeurs de chaque pixel. Ces images sont obtenues soit par numérisation (appareil photo numérique, scanner), soit par synthèse. Dans tous les cas, on ne connaît de l'image que ses pixels (échantillons) et on ne peut la calculer en tout point (de coordonnées réelles) que par reconstruction, sans garantie que ces valeurs reconstruites sont correctes : si l'image avait des fréquences élevées, elles se traduisent par des alias que l'on est incapable d'identifier.

Il existe de nombreuses manipulations que l'on peut faire sur les images, que l'on peut classe en 3 grandes catégories :

- composition : assembler différentes images pour constituer une nouvelle image avec par exemple des effets de transparence ou de masquage ;
- traitement : modification d'une image donnée soit par une transformation géométrique (rotation par exemple), soit pour modifier son contenu (élimination du bruit ou renforcement des contours par exemple) ;
- analyse : extractions d'informations structurelles ou sémantiques (contours, objets 3D, etc.).

Ces trois catégories sont loin d'être indépendantes : la composition nécessite des techniques de traitement, par exemple pour changer la taille d'une image avant de la composer, et l'analyse fait la plupart du temps appel à des techniques de traitement, notamment par filtrage. Le domaine de l'analyse d'image étant particulièrement vaste, il ne sera pas abordé ici. De même ne sont pas abordées les techniques de stockage et de compression d'images.

Dans la suite, on présente les techniques de base de traitement et de composition d'images : amélioration d'images, transformations géométriques, composition par transparence.

Amélioration d'images

[cette section est basée sur Digital Image Processing, R.C. Gonzalez and R.E. Woods, Addison Wesley, 1992]

Nous avons vu au chapitre précédent que l'on pouvait représenter une image (et plus généralement tout signal) dans le domaine fréquentiel grâce à la transformée de Fourier, et que la multiplication de deux spectres étaient équivalente à la convolution de leurs signaux (et réciproquement). Le filtrage par composition a été utilisé pour produire des images anti-aliassées. Notons que cette technique nécessite une suréchantillonnage de l'image, c'est-à-dire qu'il faut connaître la valeur de l'image en plus de points que le nombre de point d'échantillonnage.

Supposons que l'on dispose d'une image numérisée qui contient des escaliers. Peut-on améliorer cette image par filtrage ? Remarquons d'abord que toute image numérisée correspond à un signal : il se peut que l'effet d'escalier corresponde à la réalité du signal, et non pas à un effet d'aliassage. Toute technique de filtrage qui travaille à partir de l'image numérisée peut améliorer l'aspect visuel de l'image ou le déteriorer, selon les caractéristiques de l'image.

Traitements point par point

La technique de traitement la plus simple consiste à appliquer à chaque pixel de l'image une fonction qui transforme son intensité. Cette fonction peut être représentée par une courbe de transfert (fig. 1) qui donne en ordonnée l'image de chaque niveau d'intensité en abscisse :

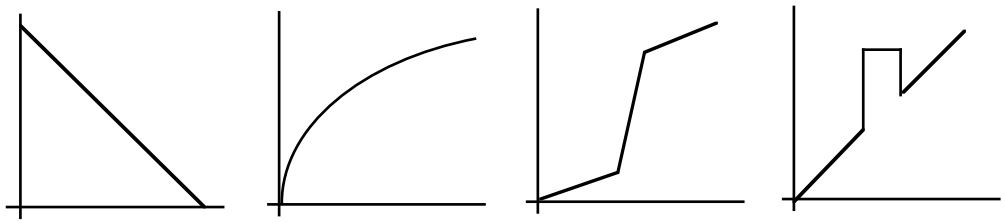


Image en négatif | Compression de l'intensité | Renforcement du contraste | Extraction de niveaux

Figure 1 - exemples de courbes de transfert

La compression de l'intensité est utile pour des images qui ont une très large plage d'intensité, comme par exemple une transformée de Fourier. Le renforcement du contraste permet d'augmenter artificiellement le contraste d'une image terne. L'extraction de niveau permet de mettre en évidence des parties d'une image ayant une intensité donnée (par exemple en analyse de données satellites).

Equalisation d'histogramme

Pour déterminer la courbe de transfert la mieux adaptée à un traitement donné, il est utile de calculer l'histogramme de l'image, qui donne le nombre de pixels en fonction de l'intensité de l'image (figure 2).

Une technique simple consiste à équaliser l'histogramme, c'est-à-dire à transformer l'image de telle sorte que l'histogramme transformé soit le plus "plat" possible. En

d'autres termes, il s'agit de rendre la fonction de densité de probabilité aussi uniforme que possible. Pour obtenir ce résultat, il suffit de prendre comme fonction de transfert la fonction *f* suivante (*n* est le nombre total de pixels, *H(i)* le nombre de pixels d'intensité *i*, fourni par l'histogramme) :

$f(i) = (1/n)\ \text{Somme}_{i=0..i}\ H(i)$

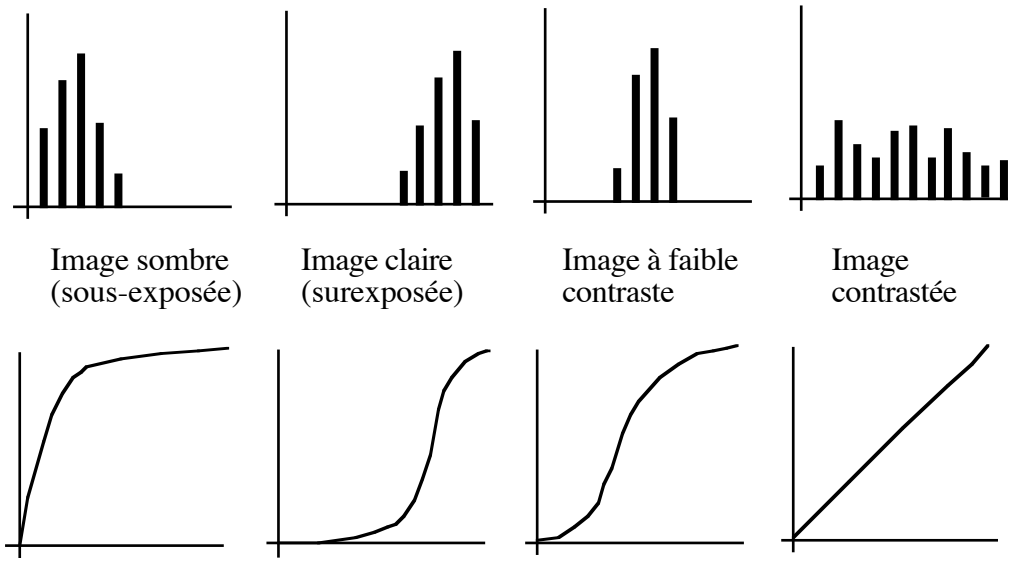


Figure 2 - exemple d'histogrammes (nb de points en fonction de l'intensité)

La fonction *f* ci-dessus a la propriété d'être monotone croissante, c'est-à-dire qu'elle conserve l'ordre des niveaux de gris. En pratique, cette méthode a l'avantage d'être automatique. C'est aussi parfois un inconvénient. On peut alors l'améliorer en spécifiant une fonction de densité de probabilité souhaitée pour l'image résultante, ce qui permet d'indiquer par exemple que l'on souhaite plus de détails, et donc plus de contraste, dans une zone donnée de l'histogramme. (Nous ne détaillons pas cette technique ici).

Méthodes locales

Les techniques point à point ont l'inconvénient d'agir globalement sur l'image. Les méthodes locales utilisent des techniques voisines des méthodes globales, mais en le appliquant à un voisinage de chaque pixel.

Histogramme local

Ainsi, pour améliorer le contraste de l'image localement, on peut appliquer la technique d'équalisation d'histogramme à chaque pixel en utilisant les valeurs des pixels voisins.

Pour chaque pixel de l'image
    calculer l'histogramme sur un voisinage (par exemple 7x7)
    calcule la fonction de transfert par équalisation de l'histogramme
    appliquer cette fonction au pixel central

Le calcul de l'histogramme peut être optimisé le long d'une ligne puisqu'à chaque déplacement, 7 pixels sortent du voisinage et 7 pixels entrent. De plus, il suffit de connaitre la valeur de la fonction de transfert pour le pixel central et donc calculer la somme des 7x7/2 premiers histogrammes.

Amélioration locale du contraste

Une autre façon d'améliorer le contraste est de calculer la moyenne *m* et l'écart-type *s* de la zone de l'image autour de chaque pixel et d'appliquer la transformation suivante (*p* est l'intensité du pixel, *p'* l'intensité après transformation, *M* la moyenne globale des intensités et *k* un réel entre 0 et 1) :

```
p' = A (p - m) + m
A = k M / s
```

A étant inversement proportionnel à la variance autour du pixel considéré, il est plus grand dans les zones de faible contraste. Le terme *A (p-m)* renforce donc le contraste dans ces zones. En pratique, il est parfois nécessaire de borner *A* entre deux valeurs *Amin* et *Amax*.

Filtrage spatial

Une technique très répandue pour améliorer les images est le filtrage spatial : on se donne un filtre sous forme d'une matrice carrée (3x3 ou 5x5 le plus souvent). On centre le filtre sur chaque pixel, et on effectue la somme des produits des intensités des pixels par les valeurs du filtre.

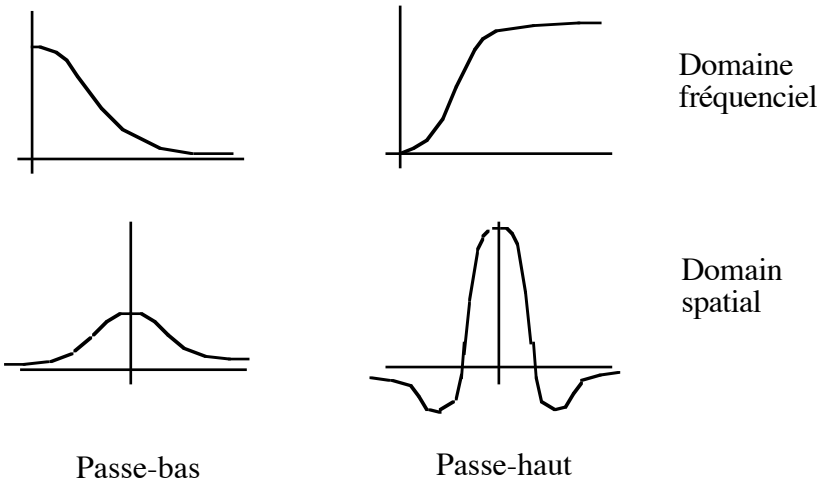


Figure 3 - filtres passe-haut et passe-bas

Lissage

Le lissage peut servir à rendre l'image un peu floue avant d'autres traitements, ou pour enlever un bruit de fond ("neige"). Un filtre *nxn* dont toutes les valeur sont $1/n^2$ effectue une moyenne de chaque pixel avec ses voisins, qui rend l'image globalement plus floue. En fait ce filtrage correspond à un mauvais filtre passe-bas comme le montre la figure 3 ci-dessus (le spectre d'une fonction carrée est la fonction sinc). Un filtre gaussien donne en général de meilleurs résultats.

Une technique qui donne souvent de meilleurs résultats est le filtrage médian: on trie les valeurs des intensités des pixels voisins et on affecte au pixel central la médiane des intensités (la médiane de n valeurs v_1, .., v_n, triées par ordre croissantes est $v_{E(n/2)}$).

Renforcement des détails

L'inverse du lissage consiste à utiliser un filtre passe-haut qui conserve les fréquences élevées. D'après la figure 3, on voit que le filtre a une forte valeur positive au centre, et des valeurs négatives autour. Un filtre 3x3 aurait pour valeurs 8 au centre et -1 ailleurs. La somme des coefficients étant nulle, le filtre assombrit les zones de même intensité et éclaircit celles où l'intensité change.

Une variante consiste à calculer la différence entre l'image initiale et l'image transformée par un filtre passe-bas. En effet:

PasseHaut = Original - PasseBas

Si l'on amplifie l'image avec un facteur *A* on obtient un filtre *Boost* qui renforce les fréquences élevées, tout en conservant les fréquences faibles :

Boost = A*Original - PasseBas
= (A - 1) * Original + Original - PasseBas
= (A - 1) * Original + PasseHaut

Sur une grille 3x3, le filtre correspondant a pour matrice :

-1/9	-1/9	-1/9
-1/9	9A-1	-1/9
-1/9	-1/9	-1/9

Pour A=1, on retrouve le filtre passe-haut. Pour A>1, on ajoute l'image originale, ce qui donne une image souvent plus lisible. Le filtre passe-haut a cependant tendance à faire apparaître du bruit, car souvent le bruit dans les images numérisées est un signal haute fréquence.

Enfin, une autre technique qui permet de renforcer les arêtes consiste à utiliser des filtres qui calculent une approximation du gradient de l'image : lorsque le gradient est élevé, les changements d'intensité sont importants. Les masques suivants donnent de bons résultats (le filtre de gauche extrait le gradient vertical, celui de droite le gradient horizontal ; on peut combiner les deux filtres en les passant en parallèle sur l'image et en additionnant les images résultantes).

Filtres de Prewitt :

-1	-1	-1	1	0	1
0	0	0	-1	0	1
1	1	1	-1	0	1

Filtres de Sobel (module du gradient) :

-1	-2	-1	1	0	1
0	0	0	-2	0	2
1	2	1	-1	0	1

Transformations géométriques

Une autre catégories de méthodes de traitement d'images concerne les transformations géométriques : translation, changement d'échelle, rotation, etc.

La translation ne pose a priori pas de problème, à moins que le vecteur de translation n'ait pas des coordonnées entières. Dans ce cas, il faut reconstruire l'image de départ à partir de ses échantillons pour pouvoir calculer l'antécédent d'un pixel de l'image translatée. Cette technique s'applique à d'autres transformations (rotation, etc.) et sera traitée par une technique générale de filtrage. Cependant, pour certains opérations, il existe des algorithmes plus efficaces que le filtrage et qui donnent de bons résultats.

Expansion d'image

Une opération très fréquente est le changement de taille d'une image. L'algorithme de Weiman permet de changer l'échelle horizontale ou verticale d'une image d'un facteur rationnel *p/q*. Si l'on veut transformer une image de taille *NxM* en une image *N'xM'*, il suffit de lui appliquer une expansion horizontale de facteur *M'/M* et une expansion verticale de *N'/N*.

Dans le cas d'une expansion horizontale, l'algorithme de Weiman fonctionne par recopie de colonne. On suppose d'abord que le rapport d'expansion *p/q* est plus grand que 1. On commence par calculer le code de Rothstein du nombre *p/q*. C'est une séquence de *p* bits tels que exactement *q* bits sont à 1, et ces *q* bits sont équitablement répartis parmi les *p* bits.

Ce code peut très facilement être calculé par l'algorithme de tracé de segments du point médian. En effet, si l'on considère les mouvements E ou NE produits par le tracé du segment *(0, 0)-(p, q)* (qui est dans le premier octant), on note qu'il y a *p* mouvements en tout (puisqu'on trace *p* pixels), et qu'il y a exactement *q* mouvements NE, puisque ce sont les seuls mouvements où *y* augmente. On peut donc calculer le code de Rothstein de *p/q* en produisant un 0 à chaque mouvement E et un 1 à chaque mouvement NE.

Supposons ce code stocké dans un tableau de bits *roth* de *p* éléments. Ce tableau nous permet de décider, pour chaque colonne de l'image de départ, sa colonne de destination dans l'image d'arrivée (on suppose ici que la largeur de l'image de départ est *q* et celle de l'image d'arrivée est *p*) :

```
sourceCol = 0;
for (j = 0; j < p; j++) {
      if (roth [j] == 1) {
            CopyColumn (j, sourceCol);
            sourceCol++;
      }
}
```

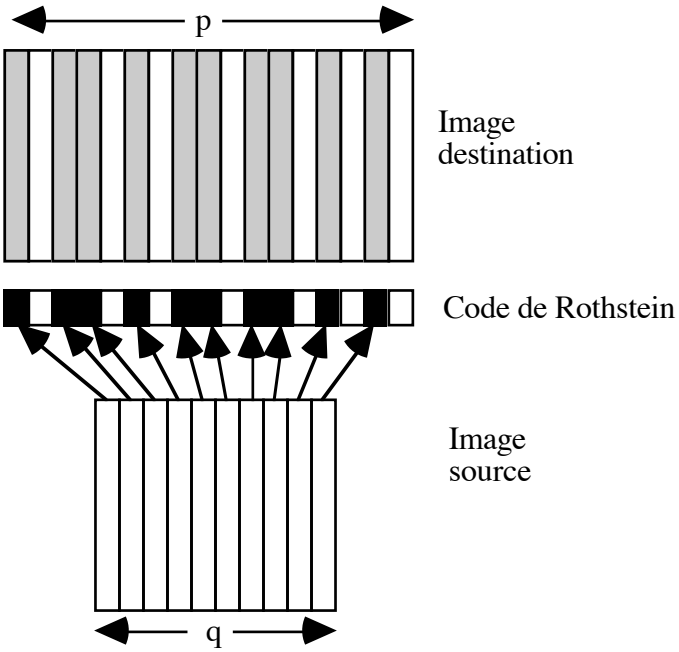


Figure 4 - Expansion d'image par l'algorithme de Weiman

A l'issue de cette phase, certaines colonnes de l'image d'arrivée n'ont pas été affectées, ce qui crée des discontinuités dans l'image. On itère donc l'algorithme avec un nouveau code de Rothstein, calculé comme le décalage circulaire du code précédent. Chaque nouvelle image est ajoutée à l'image destination. On effectue ce calcul pour tous les décalages (soit p ici), et comme l'image a été recopiée en tout q fois dans l'image de destination, on divise chaque pixel de l'image destination par q: On peut réduire le coût de l'algorithme en calculant le code de Rothstein de p'/q' tels que p' et q' soient premiers entre eux et $p'/q' = p/q$.

Si $p/q < 1$, l'expansion est en fait une contraction. On applique la même méthode mais en inversant les rôles des images sources et destination : cette fois-ci le code de Rothstein indique quelle colonne de l'image source il faut recopier dans l'image destination.

Transformations multipasses

L'algorithme de Weiman utilise deux transformations, l'une en x, l'autre en y, pour réaliser la mise à l'échelle d'une image. On peut utiliser un principe similaire pour réaliser un très large gamme de transformations, appelées transformations multipasses. L'idée de base est de ramener une transformation 2D en une séquence de transformations 1D, d'abord sur les colonnes de l'image de départ, puis sur les lignes de l'image ainsi transformée. Plus précisément, la première transformation préserve les colonnes et la seconde préserve les lignes : toutes les images des pixels d'une colonne (resp. ligne) se retrouvent sur cette colonne (resp. ligne). Dans le cas de l'algorithme de Weiman, l'ordre des transformations n'a pas d'importance. Dans le cas général, les deux transformations ne commutent pas.

Appelons (x, y) les coordonnées dans l'image de départ, (u, v) celles à l'issue de la première et (r, s) les coordonnées de l'image finale. Soit T la matrice de la transformation que l'on cherche à décomposer, A la matrice de la première transformation et B celle de la seconde. On a (on note $(x\ y)^t$ le transposé de $(x\ y)$, c'est-à-dire le vecteur colonne) :

$$(u\ v)^t = A\ (x\ y)^t$$
$$(r\ s)^t = B\ (u\ v)^t = B\ (\ A\ (x\ y)^t) = T\ (x\ y)^t$$

La première transformation préserve les colonnes, sa matice est donc telle que:

$$(u\ v)^t = A\ (x\ y)^t = (x\ f(x, y))^t$$

De même, la seconde matrice préserve les lignes, sa matrice est donc telle que:

$$(r\ s)^t = B\ (u\ v)^t = (g(u, v)\ v)^t$$

Il vient donc en composant ces transformations :

$$(r\ s)^t = B\ (u\ v)^t = B\ (\ A\ (x\ y)^t) = B\ (x\ f(x, y))^t = (g(x, f(x, y))\ f(x, y))^t$$

Soit :

$$r = g\ (x, f\ (x, y)$$
$$s = f\ (x, y)$$

Etant donnée une transformation T, il faut calculer analytiquement les fonctions f et g. On étudie le cas des rotations et de la transformation d'un carré en trapèze, qui se produit lors de la transformation perspective 3D->2D.

Rotation

La décomposition que l'on cherche est illustrée figure 5 : le rectangle est déformé verticalement, puis horizontalement.

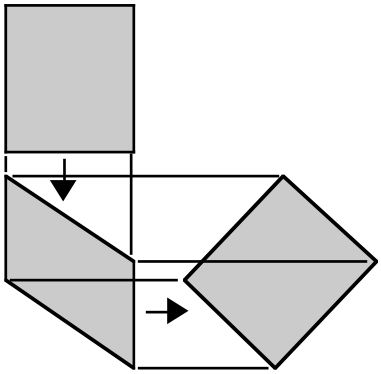


Figure 5 - décomposition d'une rotation : transformation qui préserve les colonnes suivie d'une transformation qui préserve les lignes.

La matrice d'une rotation d'angle *a* est :

cos(a) -sin(a)
sin(a) cos(a)

Soit :

r = x cos(a) - y sin(a)
s = x sin(a) + y cos(a)

Il vient donc immédiatement la définition de *f* :

f(x, y) = x sin(a) + y cos(a)

Le calcul de *g* est un peu plus délicat. Il faut exprimer *g* en fonction de *u* et *v* . On sait que :

g(u, v) = x cos(a) - y sin(a)
u = x
v = f(x, y) = x sin(a) + y cos(a)

On peut donc calculer *y* en fonction de *u* et *v* :

y = (v - x sin(a)) / cos(a) = (v - u sin(a)) / cos(a)

et remplacer dans la définition de *g* (on pose *sec(a) = 1/cos(a)*) :

g(u, v) = u cos(a) - ((v - u sin(a)) / cos(a)) sin (a)
= u (cos(a) + $\sin^2$(a) / cos(a)) - v sin(a) / cos(a)
= u ($\cos^2$(a) + $\sin^2$(a)) / cos(a) - v tan (a)
= u / cos(a) - v tan(a)
= u sec(a) - v tan(a)

Ainsi on obtient la matrice A suivante :

1	0
sin(a)	cos(a)

et la matrice B suivante :

sec(a)	-tan(a)
0	1

On peut vérifier que BA = T, matrice de la rotation. Notons que dans la dérivation, nous avons divisé par *cos(a)*, qui est nul lorsque a vaut 90º. Cela ne pose pas de problème car une rotation d'angle droit est une simple transposition de pixels.

En fait, lorsque *a* s'approche de l'angle droit, la transformation se comporte mal car la matrice *B* a deux termes en *1/cos(a)* qui deviennent très grands et de signes opposés. En les additionnant, on perd en précision à cause des arrondis du calcul.

Il est donc préférable, pour de petites valeurs de *a*, de faire une rotation de 90° en transposant les lignes et les colonnes de l'image suivie d'une rotation d'un angle (90 - *a*).

Le problème du "bottleneck" (goulot d'étranglement)

La figure 6 illustre un problème différent, et plus général : selon que l'on effectue d'abord la transformation verticale puis horizontale ou l'inverse (les matrices ne sont pas les mêmes dans les deux cas), l'image intermédiaire peut se trouver très compressée (cas de gauche dans la figure 6). Bien que mathématiquement, les deux possibilités soient exactement équivalentes, il n'en est pas de même sur le plan informatique. En effet, l'image intermédiaire est calculée avec la même résolution que l'image de départ, et l'on perd donc de l'information.

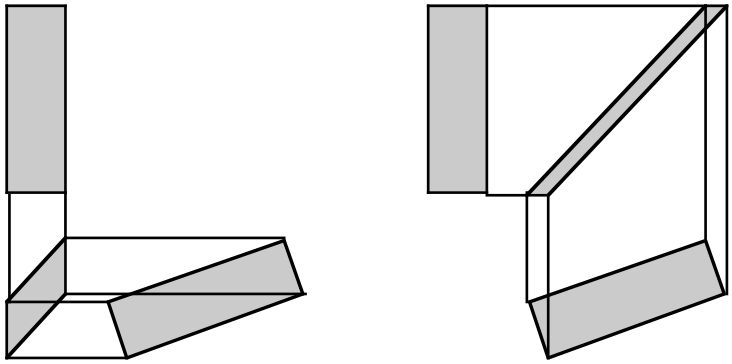


Figure 6 - le problème du "bottleneck"

On a donc en général 4 possibilités pour faire une transformation multipasses : rotation de 90° éventuelle de l'image suivie de la transformation horizontale puis verticale, ou verticale puis horizontale. Bien qu'on ne sache pas le prouver formellement, il existe toujours une solution qui marche mieux que les autres et l'on ne connaît pas de cas où toutes les solutions soient mauvaises.

Enfin, dans le cas des rotations, on peut utiliser une décomposition en trois étapes (horizontal, vertical, horizontal), qui marche quel que soit l'angle :

R = A B C

avec A = $\begin{matrix} 1 & -\tan(a/2) \\ 0 & 1 \end{matrix}$

B = $\begin{matrix} 1 & 0 \\ \sin(a) & 1 \end{matrix}$

et C = $\begin{matrix} 1 & -\tan(a/2) \\ 0 & 1 \end{matrix}$

Transformation d'un carré en trapèze

Cette transformation est utile lors de la projection perspective d'une facette carrée. Cette opération est appelée *placage de texture* car elle correspond à l'affichage de la projection d'une image qui serait plaquée sur la facette (figure 7). Soit la projection :

$r = x / (y + 1)$

$s = y / (y + 1)$

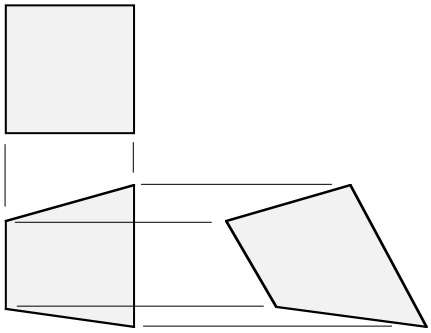


Figure 7 - placage de texture

On a donc :

$u = x$
$v = f(x, y) = y / (y + 1)$
$\Rightarrow y = v (y + 1) \Rightarrow (1 - v) y = v \Rightarrow y = v / (1 - v)$
$g (u, v) = x / (y+1) = u / ((v / (1 - v)) + 1) = u / (1 / (1 - v)) = u (1 - v)$

On obtient donc les transformations suivantes, que l'on peut représenter par des matrices en coordonnées homogènes :

Verticalement : $u = x; v = y / (y + 1)$
Horizontalement : $r = u (1 - v); s = v$

Cette technique peut être appliquée pour transformer une image par une transformation de la forme

$$T(x, y) = S(m(x)h_1(y), m(x)h_2(y))$$

dans laquelle *S* est n'importe quelle transformation "standard" en infographie (translation, mise à l'échelle, rotation, perspective), *m*, h_1 et h_2 sont quelconques. Le placage de texture sur des surfaces courbes peut être réalisé avec la même méthode en prenant une transformation de la forme

$$T(x, y) = (t_1(x, y), t_2(x, y))$$

dans laquelle t_1 et t_2 sont deux formes bicubiques telles que T est injective (c'est-à-dire que deux points distincts ont toujours des images distinctes). (Voir "Planar 2-pass texture mapping and warping", Alvy Ray Smith, SIGGRAPH'87, pp.263-272.)

Anti-aliasage des transformations

Lorsque l'on utilise ces transformations multipasses, chaque ligne ou colonne est transformée en une ligne ou une colonne, mais les pixels ne sont pas transformés en des pixels pour autant : l'image d'un pixel par une transformation horizontale par exemple a de fortes chances d'avoir une abscisse non-entière. Pour calculer la valeur du pixel destination, il faut combiner les valeurs de plusieurs pixels sources.

Idéalement, il faudrait calculer la valeur d'un pixel destination en ré-échantillonnant l'image source au point antécédent du pixel. Pour faire ce ré-échantillonnage, il faut reconstruire l'image en appliquant un filtre *sinc* à la position du pixel destination. Dans la pratique, ce calcul est très complexe et on le simplifie de deux façons :

- d'une part on ne considère que les pixels voisins sur la même ligne (ou colonne pour une transformation verticale) que le pixel considéré ;

- d'autre part on ne calcule pas l'antécédent du support du filtre, mais on utilise un support fixe. En fait, le plus souvent on utilise un simple filtre carré sur les pixels source.

Ces deux approximations donnent de bons résultats si l'image n'est pas trop déformée lors de la première transformation. C'est une raison supplémentaire pour éviter le "bottleneck". A l'issue des deux transformations, un pixel source aura contribué à un certain nombre de pixels destination (voir figure 8). Il en résulte qu'à chaque transformation, l'image perd de la netteté : si l'on tourne une image d'un angle *a*, puis le résultat d'un angle *-a*, on obtient une image de moins bonne qualité que l'image originale.

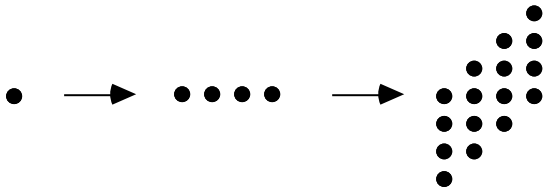


Figure 8 - pixels affectés par un pixel source

Transformation par filtrage

Au lieu de décomposer une transformation en plusieurs passes, on peut faire le traitement en une passe. Cette approche n'est pas plus efficace que la transformation multipasse mais elle donne un résultat plus exact. L'algorithme présenté est celui de Feibush, Levoy et Cook.

Le principe est le suivant : pour chaque pixel de la destination, on détermine le support du filtre qui permettra de calculer la valeur du pixel. On calcule les antécédents des positions des pixels qui sont dans le support du filtre. On calcule la valeur de l'image source à ces positions et on les multiplie par les valeurs du filtre pour déterminer la valeur du pixel. L'algorithme est compliqué par la prise en compte du contour de l'image : lorsque le support du filtre sort du contour de l'image, on ne prend en compte que son intersection avec le support.

La première étape, calcul des antécédents des positions, est essentiellement algébrique. Dans certains cas, un algorithme incrémental peut être utilisé. L'étape de filtrage est plus complexe (figure 9) . Pour chaque pixel *P* de l'image destination :

- on calcule le rectangle englobant le support du pixel dans l'image destination ;
- on calcule l'antécédent de ce rectangle dans l'image source (c'est un quadrilatère) (étape 1, figure 9) ;
- pour chaque pixel *p* de l'image source qui est dans le rectangle englobant ce quadrilatère (étape 2) *et* dans le contour de l'image, on calcule la position *p'* du pixel dans l'image destination (étape 3) ;
- si le support du pixel *p'* n'est pas totalement intérieur au contour, on calcule l'intersection du contour et du support du pixel et on détermine (grâce à une table pré-calculée) la pondération du pixel ;
- on multiplie la valeur du pixel *p'* par la valeur du filtre en ce point, en prenant en compte la pondération éventuelle, et on accumule cette valeur dans l'intensité du pixel *P*.

Cet algorithme donne de bons résultats mais peut être très coûteux sur des petites parties de l'image destination qui ont des antécédents de grande taille. Par exemple, un objet distant d'une scène tridimensionnelle apparaît, après projection

perspective, petit à l'écran. Un pixel de l'écran peut avoir dans l'image source un antécédent couvrant plusieurs centaines de pixels : on passe beaucoup de temps à calculer la valeur d'un seul pixel.

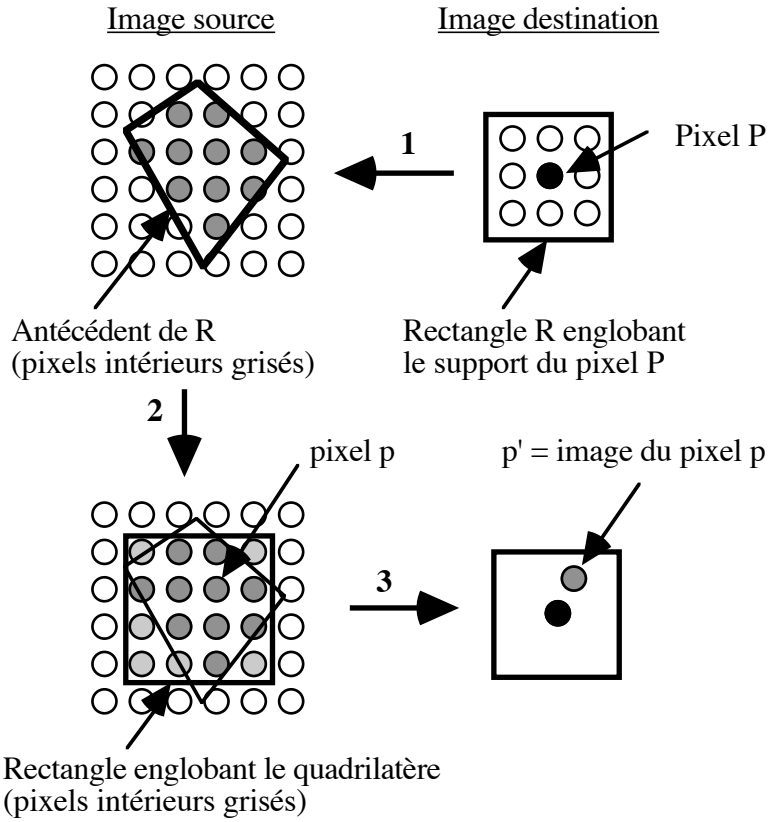


Figure 9 - transformation par filtrage

La solution la plus fréquemment employée (et implémentée en standard dans un système comme OpenGL) est l'utilisation de plusieurs résolutions de l'image source, appellées "MIP maps" (multum in parvo) : lorsque l'antécédent d'un pixel est trop grand, on remplace l'image source par une image de plus faible résolution, réduisant ainsi le nombre de pixels à considérer (figure 10).

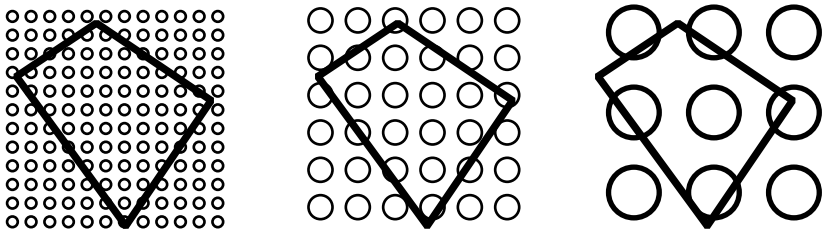


Figure 10 - Comparaison des nombres de pixels couverts par le même quadrilatère dans 3 "MIP maps". Le bon choix est la version qui s'approche le mieux du support du filtre dans l'image destination (ici celle du milieu).

Transparence

L'une des principales opérations de composition d'images consiste à superposer deux images. Même si une image doit être affichée au-dessus de l'autre, des

problèmes se posent le long des contours de l'image : si l'on ne prend pas en compte le fait que le contour de l'image traverses certains pixels, la superposition laissera apparaîtres des effets d'escalier.

Le traitement correct de la composition d'images nécessite d'associer à chaque pixel une valeur, indépendant de son intensité, qui indique la proportion du pixel couverte par l'image en ce point. Cette valeur, appelée α ("alpha-value"), est souvent stockée dans des plans spéciaux des tampons d'image, appelés "alpha-channel". En plus de servir à décrire précisément les contours d'une image, cette valeur permet également de gérer la transparence des images : dans ce cas, la valeur alpha est interprétée comme la proportion de lumière que le pixel laisse passer, et donc la proportion de l'image sous-jacente qui est visible à travers le pixel.

Considérons pour l'instant les couvertures partielles et évaluons la façon dont se combinent les valeurs-alpha de deux pixels lorsque l'on compose deux images. Si une image recouvre 1/3 d'un pixel avec la couleur rouge et une autre image la moitié de ce même pixel avec la couleur bleue. Quelle doit être la couleur finale du pixel ? Si l'on considère que le recouvrement du pixel est aléatoire, en moyenne, les deux pixels se recouvrent sur 1/3 * 1/2 = 1/6 de leur surface (figure 11).

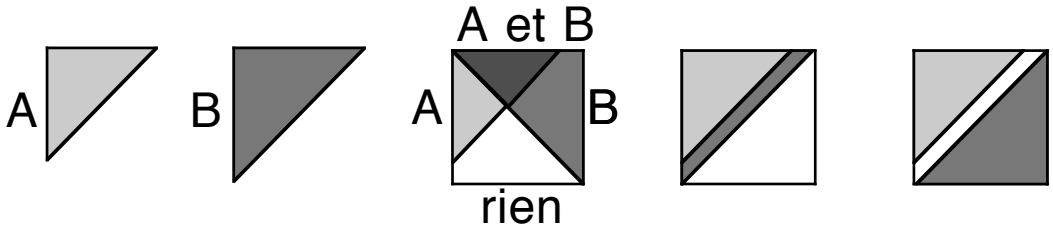


Figure 11 - recouvrement d'un pixel par deux images.
Les deux situations de droite ne sont pas considérées :
on considère toujours une intersection proportionnelle.

Si l'on veut faire un mélange des deux images, avec 60% de *A* et 40% de *B*, on calcule la valeur de pixel en mélangeant les couleurs de *A* et *B* avec les proportions 1/3*60% de rouge et 1/2*40% de bleu, soit, en notation RVB (Rouge, Vert, Bleu) : (0.2, 0, 0.2).

Un tel mélange suppose que les deux images sont transparentes. Parfois, on veut mettre l'image A devant l'image *B* ou ne considérer que les parties de *A* qui sont dans *B*, etc. On peut identifier toutes les combinaisons possibles en remarquant que chaque pixel peut avoir 4 régions (figure 11) : celle recouverte par A et B, celle recouverte seulement par *A*, celle recouverte seulement par *B* et celle recouverte ni par *A*, ni par *B*. Si l'on appelle α_A et α_B les valeurs-alpha des deux images au pixel considéré, on peut calculer les proportions couvertes par ces régions :

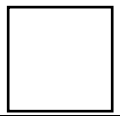
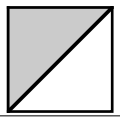
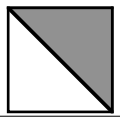
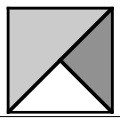
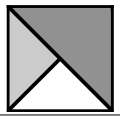
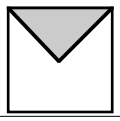
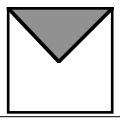
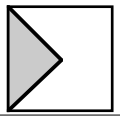
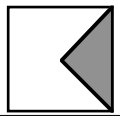
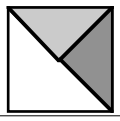
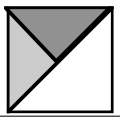
- la proportion couverte par les deux images est $\alpha_A \ \alpha_B$;
- la proportion couverte par l'image A seulement est $\alpha_A \ (1 - \alpha_B)$;
- la proportion couverte par l'image B seulement est $\alpha_B \ (1 - \alpha_A)$;
- la proportion couverte par aucune des images A et B est $(1 - \alpha_A) \ (1 - \alpha_B)$.

Selon le mode de superposition choisi, on peut choisir la couleur qui doit apparaître dans chacune de ces régions :

- la région couverte ni par *A*, ni par *B* est toujours noire ;
- la région couverte seulement par *A* peut être de la couleur de *A* ou noire ;
- la région couverte seulement par *B* peut être de la couleur de *B* ou noire ;

- la région couverte par *A* et par *B* peut être noire, de la couleur de *A* ou de la couleur de *B*.

On obtient 12 possibilités (le quadruplet de la description représente les couleurs choisies pour les 4 zones identifiées ci-dessus) :

opération	**description**	**schéma**	$\mathbf{F_A}$	$\mathbf{F_B}$
Effacer	0, 0, 0, 0		0	0
A	0, A, 0, A		1	0
B	0, 0, B, B		0	1
A au-dessus de B	0, A, B, A		1	$1 - \alpha_A$
B au-dessus de A	0, A, B, B		$1 - \alpha_B$	1
A inclus dans B	0, 0, 0, A		α_B	0
B inclus dans A	0, 0, 0, B		0	α_A
A caché par B	0, A, 0, 0		$1 - \alpha_B$	0
B caché par A	0, 0, B, 0		0	$a - \alpha_A$
A si B	0, 0, B, A		α_B	$1 - \alpha_A$
B si A	0, A, 0, B		$1 - \alpha_B$	α_A
A xor B	0, A, B, 0		$1 - \alpha_B$	$1 - \alpha_A$

Si l'on choisit par exemple l'opération de composition "*A* au-dessus de *B*", il faut mélanger la couleur de *A* avec $(1 - \alpha_A)$ la couleur de *B*. La couleur de *A* est C_A et la couleur de *B* est C_B, la couleur finale du pixel est :

$$\alpha_A \, C_A + (1 - \alpha_A) \, \alpha_B \, C_B$$

De façon générale, si l'on appelle F_A la fraction du pixel de l'image A qui reste visible en fonction de l'opération choisie et de même pour F_B (colonnes de droite de la table ci-dessus), l'opération de composition s'écrit :

$$F_A\ \alpha_A\ C_A + F_B\ \alpha_B\ C_B$$

On voit que cette équation fait systématiquement apparaître les termes $\alpha_A\ C_A$ et $\alpha_B\ C_B$. Il est donc judicieux de stocker une valeur de pixel (R, V, B, α) sous la forme (αR, αV, αB, α). Dans ce cas, il faut s'assurer que les composantes de couleurs ne soient pas supérieures à la valeur-alpha du pixel. L'équation de composition s'écrit alors :

$$C = F_A\ C_A + F_B\ C_B$$
$$\alpha = F_A\ \alpha_A + F_B\ \alpha_B$$

Le terme C définit la couleur du pixel et le terme α sa valeur-alpha.

Enfin, quelques opérations unaires et une nouvelle opération binaire sont intéressantes pour la composition d'images (f, t et q sont compris entre 0 et 1) :

foncer (A, f) = (f R_A, f V_A, f B_A, α_A)
transparent (A, t) = (t R_A, t V_A, t B_A, t α_A)
opaque (A, q) = (R_A, V_A, B_A, q α_A)
A **plus** B = ($R_A + R_B$, $V_A + V_B$, $B_A + B_B$, $\alpha_A + \alpha_B$)

Pour réaliser un fondu enchaîné entre deux images, il suffit de calculer, pour plusieurs valeurs de t :

transparent (A, t) **plus transparent** (B, 1 - t)

Il faut remarquer que la fonction **opaque** n'agit que sur la valeur-alpha. Il se peut donc que celle-ci devienne inférieure aux composantes de couleur du pixel, ce que l'on a pourtant interdit plus haut. Si cela se produit, le pixel est dit "lumineux" car il ajoute de la lumière plutôt que de seulement la filtrer. Lorsque les pixels sont finalement restitués à l'écran, il faut :

- diviser chaque composante par la valeur-alpha si l'on avait stocké l'image avec la valeur-alpha pré-multipliée ;
- assurer que chaque composante de couleur est dans l'intervalle [0, 1], en la ramenant dans cet intervalle si ce n'est pas le cas.

Enfin, on peut réaliser des effets de composition d'image avec la table de couleurs (voir le début du cours de licence...). Dans ce cas, on stocke les images dans des plans de la mémoire d'écran, et on utilise des manipulations de la table de couleur pour réaliser des effets similaires à ceux que l'on vient de voir. L'avantage est la rapidité des effets d'animation comme le fondu enchaîné puisqu'il suffit de changer la table de couleurs, pas de recalculer l'image. L'inconvénient est la perte de résolution en couleurs, puisque les images ont un faible nombre de bits par pixel pour pouvoir être affectées à des plans de la mémoire d'écran.