Informatique Graphique II

1- Anti-Aliassage

Michel Beaudouin-Lafon, mbl@lri.fr

source : Foley & van Dam

Introduction

Les tracés de segments et d'autres primitives que nous avons vu en Licence ont un inconvénient : ils font apparaître des "escaliers", surtout lorsque la pente est proche de l'horizontale ou de la verticale. Les techniques d'anti-aliassage (ou anti-aliasing, ou anti-crênelage) on pour but de réduire cet effet d'escalier.

L'effet d'escalier est dû à la nécessité d'appoximer une primitive graphique définie dans le plan réel par un ensemble discret de pixels situés sur une grille. Plus la grille est fine, moins l'effet d'escalier est sensible. Si l'on va au-delà du seuil de la perception humaine, comme sur une imprimante de 1200 points-par-pouce, l'effet d'escalier existe toujours mais n'est plus perceptible.

La figure 1 ci-dessous montre l'empreinte d'un segment d'épaisseur 1 sur la grille de pixels et les pixels allumés par l'algorithme du point médian. Ce schéma montre d'une part que les pixels allumés ne sont pas totalement recouverts par l'empreinte, d'autre part que certains pixels non allumés sont en partie recouverts par l'empreinte. L'idée de base pour diminuer l'effet d'escalier est d'allumer tous les pixels qui intersectent l'empreinte, de façon proportionnelle à la surface recouverte. Cela nécessite bien entendu un écran à niveaux de gris (on se place dans le cas monochrome) : l'anti-aliassage n'est pas possible sur un écran noir et blanc.

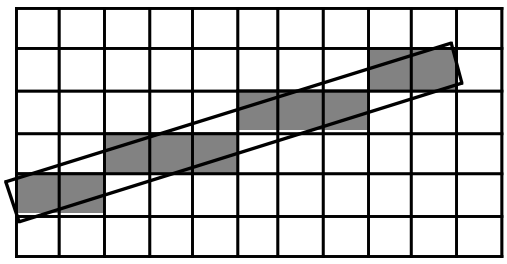


Figure 1. Aliassage des segments

Nous allons voir dans un premier temps un algorithme simple d'anti-aliassage de segments. Ensuite, pour comprendre et traiter le problème de façon plus générale, il va falloir introduire quelques outils mathématiques de traitement du signal, notamment la transformée de Fourier. Un effet incontournable de l'anti-aliassage est de rendre les images moins nettes : pour atténuer l'effet d'escalier, qui est dû à un fort contraste entre des pixels adjacents, on diminue le contraste en allumant des pixels "à moitié", ce qui diminue le contraste mais aussi la netteté.

Anti-aliassage de segments

Au vu de la figure 1 ci-dessus, on peut aisément définir les règles qui décrivent comment allumer chaque pixel lorque l'on trace une primitive :

- si la primitive recouvre complètement le pixel, celui-ci est alllumé avec une intensité maximale ;
- si la primitive a une intersection vide avec le pixel, celui-ci reste éteint ;
- si la primitive intersecte le pixel, l'intensité du pixel est proportionnelle à la surface de l'intersection.

Anti-aliassage non pondéré

Supposons dans un premier temps que les pixels sont carrés. Nous avons observé que, lorsque la résolution de l'affichage augmente, l'effet d'escalier diminue. On peut exploiter cette propriété pour implémenter un algorithme d'anti-aliassage de la façon suivante :

- rastériser la primitive sur une grille n fois plus fine que la grille d'affichage ;
- pour chaque pixel de l'affichage, compter la proportion des n^2 points de la grille correspondant à ce pixel et qui ont été allumés par la rastérisation. Allumer le pixel avec l'intensité correspondante.

Malheureusement, cet algorithme produit des résultats assez décevants, surtout pour les lignes fines (figure 2). La raison en est que la règle de proportionalité ne prend pas en compte la distance entre la zone de recouvrement et le centre du pixel : supposons que la primitive à afficher soit un point plus petit qu'un pixel. Avec l'algorithme ci-dessus, quelle que soit la position du point dans un pixel, celui-ci sera allumé avec la même intensité. Pourtant, il semblerait plus judicieux de faire en sorte que si le point est proche du centre du pixel, celui-ci soit allumé avec une intensité assez forte, tandis que si le point est proche du bord d'un pixel, celui-ci soit allumé avec une intensité plus faible. Afin que l'intensité perçue soit la même, il faudrait alors allumer également le pixel voisin le plus proche. Ainsi, si le point passe du centre d'un pixel au centre de son voisin, ce nouvel algorithme donnerait une animation plus continue (figure 3) :

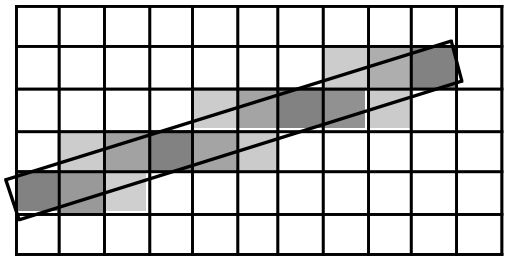


Figure 2. Anti-aliassage non pondéré d'un segment d'épaisseur 1

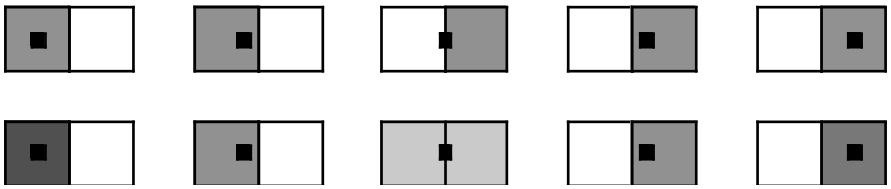


Figure 3. Anti-aliassage non pondéré (ligne du haut) contre anti-aliassage pondéré (ligne du bas) pour l'affichage d'un point

Pour implémenter cette variante dans le cas général du tracé d'une primitive graphique, il faut redéfinir notre notion de la "surface" de l'intersection entre la primitive et le pixel. Pour cela on associe à chaque pixel une fonction qui donne le poids d'un point à distance *d* du centre du pixel. Cette fonction peut être représentée par un volume (figure 4) : à tout point *(x, y)* on associe $z = poids_P (x, y)$. L'intersection entre la primitive et un pixel *P* est alors calculée comme l'intégrale, pour tous les points de la primitive, de la fonction $poids_P$. Cette intégrale s'interprète comme le volume du pixel qui intercepte l'empreinte de la primitive (figure 4).

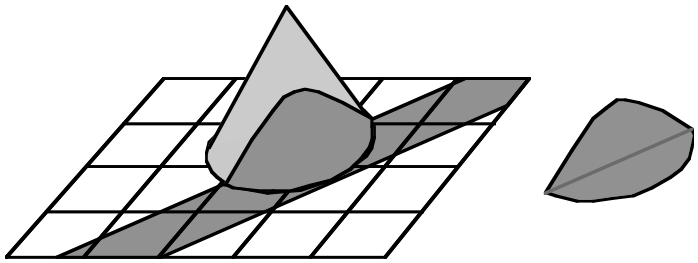


Figure 4. Fonction de pondération

La première méthode d'anti-aliassage vue ci-dessus correspondait à l'usage d'une fonction poids qui vaut *1* sur l'empreinte du pixel et *0* partout ailleurs (figure 5). Pour éviter l'effet de discontinuité que produit cette fonction, il faut d'une part que les supports des fonctions de poids de pixels voisins s'intersectent (pour qu'un point puisse contribuer à l'intensité d'un pixel voisin), et d'autre part que la fonction décroisse lorsque l'on s'éloigne du centre du pixel. Par souci de symétrie, on choisit un cône centré au centre du pixel. L'empreinte d'un pixel est alors un cercle. Une bonne valeur pour le rayon de ce cercle est 1.

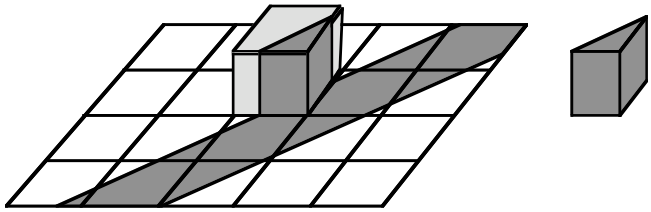


Figure 5. Fonction "boîte" pour l'anti-aliassage non pondéré

La figure 6 montre l'intersection d'une ligne d'épaisseur 1 avec les pixels d'une colonne. Pour un segment du premier octant, le nombre de pixels intersectés varie entre 2 et 5, le cas le plus fréquent étant 3. On peut donc modifier l'algorithme du point médian pour allumer non pas un pixel pour chaque valeur de *x*, mais les 2 à 5 pixels d'abscisse *x* qui intersectent le segment. Le problème se ramène alors à calculer l'intensité de chaque pixel.

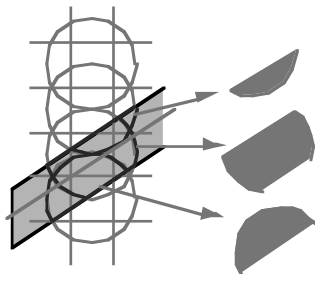


Figure 6. Anti-aliassage pondéré d'un segment d'épaisseur 1

Algorithme de Gupta-Sproull

On peut remarquer que l'intersection entre le segment et un pixel ne dépend que de la distance du segment au centre du pixel et de l'épaisseur du segment, ici fixée à 1. Si l'on suppose l'existance d'une fonction *poids(D)* où *D* est la distance du segment au centre du pixel, notre problème revient à calculer *D* pour les pixels concernés d'une abscisse donnée.

Soit $P\ (x_P, y_P)$ le dernier point calculé par l'algorithme du point médian, *E* et *NE* les points correspondant à un déplacement Est et Nord-Est respectivement. *M* est le point médian (de coordonnées $x_P+1,\ y_P+1/2$) et *I* le point d'intersection du segment avec la droite $x = x_P+1$ (figure 7).

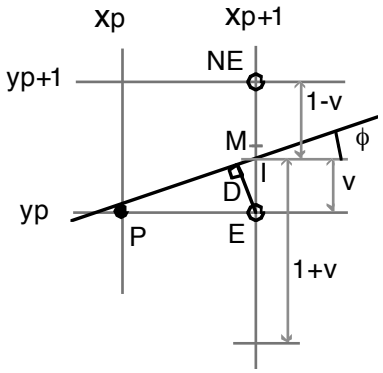


Figure 7. Algorithme de Gupta-Sproull

On considère dans un premier temps un déplacement Est. On se limite au calcul de l'intensité de 3 pixels : le pixel Est $(x_P+1,\ y_P)$ et ses voisins immédiats au-dessus et au-dessous : $(x_P+1,\ y_P+1)$ et $(x_P+1,\ y_P-1)$.

On pose

$$v = y_I - y_P$$

On a alors

$$D = v \cos(\varphi)$$

Par la loi des triangles semblables, φ est l'angle du segment avec l'horizontale, et on a donc

$$\cos(\varphi) = dx / \sqrt{(dx^2 + dy^2)}.$$

Le calcul de *v* peut être réalisé incrémentalement à partir de celui de la variable de décision *d* de l'algorithme du point médian. On rappelle que :

$$d = F(x_P+1, y_P+1/2)$$
$$F(x, y) = 2\ (a\ x + b\ y + c)$$

Pour tout point du segment, on a

$$a\ x + b\ y + c = 0, \text{ soit } y = (a\ x + c) / -b$$

Pour $x = x_P+1$, on obtient :

$$y_I = (a\ (x_P+1) + c) / -b$$
$$v = y_I - y_P = (a\ (x_P+1) + c) / -b - y_P$$
$$= (a\ (x_P+1) + c + b\ y_P) / -b$$

$= 1/2\ F(x_P+1, y_P) / -b$

Comme $b = -dx$, il vient

$2\ v\ dx = F(x_P+1, y_P)$

Cette expression nous permet de calculer D aisément. En effet :

$D = v \cos(\varphi) = v\ dx / \sqrt{(dx^2 + dy^2)} = (2\ v\ dx) / (2\ \sqrt{(dx^2 + dy^2)})$

Le dénominateur peut êre calculé une fois pour toute, et le numérateur peut être calculé de façon incrémentale à partir de la valeur de d. En effet :

$2\ v\ dx = F(x_P+1, y_P) = 2\ (a\ (x_P+1) + c + b\ y_P)$
$= 2\ (a\ (x_P+1) + c + b\ (y_P + 1/2) - b/2)$
$= F(x_P+1, y_P +1/2) - b$
$= d + dx$

Finalement, on obtient comme valeur de D :

$D = (d + dx) / (2\ sqrt(dx^2 + dy^2))$

En précalculant Q, valeur du dénominateur, on obtient :

$Q = 1 / (2\ sqrt(dx^2 + dy^2))$
$D = Q\ (d + dx)$

Cette expression permet de calculer l'intensité du pixel Est, de coordonnées (x_P+1, y_P). Il faut également calculer l'intensité des pixels adjacents de la même colonne : $(x_P+1, y_P +1)$ et $(x_P+1, y_P -1)$, et donc leurs distances respectives D' et D'' au segment. Pour ces points, v est remplacé par $v' = (1 - v)$ et $v'' = (1 + v)$ respectivement. Il vient :

$D' = Q\ (2\ v'\ dx) = Q\ (2\ (1 - v)\ dx) = Q\ (2\ dx - 2\ v\ dx) = 2\ Q\ dx - D$
$D'' = Q\ (2\ v''\ dx) = Q\ (2\ (1 + v)\ dx) = Q\ (2\ dx + 2\ v\ dx) = 2\ Q\ dx + D$

On remarque que $R = 2\ Q\ dx$ est une valeur qui peut également être pré-calculée.

Pour un déplacement Nord-Est, il faut réaliser le même calcul mais cette fois les trois pixels considérés sont le pixel Nord-Est $(x_P+1, y_P +1)$ et ses deux voisins : $(x_P+1, y_P +2)$ et (x_P+1, y_P). On utilise donc

$v = x_I - (y_P +1)$

On obtient alors :

$D = Q\ (d - dx)$
$D' = 2\ Q\ dx - D$
$D'' = 2\ Q\ dx + D$

La version modifiée de l'algorithme du point médian est alors la suivante (pour le premier octant). Elle utilise une fonction *Allumer* qui prend en paramètre les coordonnées du pixel et la distance du segment à ce pixel.

```
TracerLigne (x0, y0, x1, y1) {
      int dx = x1 - x0;
      int dy = y1 - y0;
      /* point courant de l'algorithme */
      int x = x0;
      int y = y0;
      /* variables pour le calcul du point médian */
      int d = 2 * dy - dx;
      int dE = 2 * dy;
      int dNE = 2 * (dy - dx);
      /* variables pour l'anti-aliassage */
      int num = 0;
```

```
        float Q = 1.0 / (2.0 * sqrt (dx+dx + dy*dy));
        float R = 2.0 * Q * dx;
        float D = 0.0;

        /* 1er point. Il est exact donc sa distance est 0 */
        Allumer(x, y, 0.0);
        Allumer(x, y+1, R);
        Allumer(x, y-1, R);

        while (x < x1) {
              /* choix du déplacement */
              if (d < 0) {
                    num = d + dx;
                    d += dE;
                    x++;
              } else {
                    num = d - dx;
                    d += dNE
                    x++, y++;
              }

              /* calcul de la distance et affichage */
              D = num * Q;
              Allumer (x, y, D);
              Allumer (x, y+1, R - D);
              Allumer (x, y-1, R + D);
        }
  }
```

La procédure *Allumer* utilisée ci-dessus prend comme troisième paramètre la distance entre le segment et le centre du pixel. Elle doit calculer l'intensité du pixel en fonction de cette distance et afficher le pixel. Cette intensité est proportionnelle au volume du cône intercepté par la trace du segment (voir plus haut). Elle est nulle lorsque le segment n'intercepte pas le support du pixel, c'est-à-dire lorsque $d > 3/2$. Comme l'intensité à calculer est une valeur entière dans un intervalle correspondant au nombre de niveaux de gris qui peuvent être affichés, il est beaucoup plus efficace de stocker dans une table pré-calculée les valeurs d'intensité en fonction de la distance. Cette table est indépendante du segment, et peut donc être calculée ou chargée au lancement du programme.

Le calcul de la table peut être fait de façon analytique. Une façon plus simple consiste à échantillonner la fonction sur une grille carrée et à calculer le volume pour différentes valeurs de D :

```
/* table a remplir par CalculeTable (n, greys).
   n+n/2 entrées de la table sont utilisées.
   greys est le nombre de niveaux de gris disponibles.
   Soit d la distance entre le segment et un pixel.
   table[d*n] contient l'intensite a affecter au pixel
*/
int table [MAX];

CalculeTable (int n, int greys) {
      float colonne [MAX];
      int nsur2 = n/2
      int i, j;
      float s, total;

      /* calcul du volume interceptant chaque colonne
         i et j parcourent un quart du cône, il faut donc
         doubler les valeurs pour prendre en compte toute
         la hauteur du cône
```

```
         */
        total = 0.0
        for (i = 0; i < n; i++) {
              s = 0.0;
              for (j = 0; j < n; j++) {
                    float d = sqrt ((n-i)*(n-i)+(n-j)*(n-j));
                    if (d > n )
                          d = n ;
                    s += n - d;
              }
              colonne [i] = 2 * s;
              total += 4*s;
        }

        /* calcul de la table : 1ere partie correspondant
           à une intersection d'épaisseur inférieure à 1
        */
        s = 0.0;
        for (i = 0; i < n; i++) {
              s += cumul [i]
              table [n+nsur2  - i] = greys * s / total;
        }

        /* calcul de la table : 2eme partie correspondant
           à une intersection d'épaisseur supérieure à 1
        */
        for (i = 0; i <= nsur2 ; i++) {
              s = s - cumul [i] + cumul [n-i];
              table [nsur2 - i] = greys * s / total;
        }
}
```

La fonction *Allumer* s'écrit alors :

```
Allumer (int x, int y, float d) {
      if (d < 0)
            d = -d;
      /* rien à allumer si le pixel est trop loin */
      if (d > 1.5)
            return;
      /* allumer le pixel.
         Le troisième paramètre est le niveau de gris */
      SetGreyPixel (x, y, table [d*TailleTable]);
}
```

Problèmes de cette technique

L'algorithme ci-dessus a l'avantage d'être simple et efficace. Cependant, il présente plusieurs problèmes qui limitent son usage.

Sous la forme présentée, l'algorithme ne traite que les segments d'épaisseur 1. Il peut être étendu sans difficulté pour traiter des segments plus épais. Pour cela, il faut une table pour chaque valeur de l'épaisseur, et il faut calculer le nombre de pixels à allumer, qui est de l'ordre de $n+3$ pour une épaisseur de n.

L'algorithme ne traite pas les extrémités, qui nécessitent des calculs d'intersection plus complexes. Une extension de l'algorithme existe qui utilise des tables annexes pour les extrémités. Le problème de l'encombrement des tables se pose alors.

L'algorithme ne traite que les segments. Une approche similaire est possible pour le tracé de cercles, mais là encore il faut créer des tables spécifiques. Pour le tracé de primitives plus complexes, comme les courbes de Bézier, l'algorithme est difficile à adapter car même si l'on transforme ces primitives en suite de segments, les raccords entre segments sont difficiles à traiter correctement.

Enfin, l'algorithme ne traite pas correctement le tracé de plusieurs segments qui se chevauchent. En effet, autour de l'intersection, les pixels du dernier segment tracé remplaceront ceux de l'autre segment et l'on aura l'impression qu'un segment "cache" en partie l'autre. Une solution approchée consiste à *ajouter* l'intensité calculée à l'intensité courante de chaque pixel au lieu de remplacer celle-ci. Cette solution est approchée car il faudrait calculer l'*union* des intersections et non leur somme.

Anti-aliassage par filtrage

Pour bien comprendre l'origine de l'aliassage et les remèdes possibles, il est nécessaire d'introduire quelques concepts de base en traitement du signal. L'image que l'on veut dessiner à l'écran est constituée de primitives graphiques. On peut représenter cette image par une fonction de deux variables *s(x, y)* qui associe à tout point de coordonnées réelles son intensité (on se limite ici à des scènes monochromes). Notre problème est de représenter cette fonction par un ensemble de valeurs discrètes, de telle façon que la fonction originale puisse être restituée (on dit *reconstruite*) de façon aussi précise que possible à partir des seules valeurs discrètes.

Une telle fonction est appelée *signal*, au même titre par exemple qu'un signal électrique défini par les variations de son amplitude au cours du temps ou le signal acoustique produit par une source musicale. La seule différence est qu'un signal électrique ou acoustique est une fonction d'une variable temporelle, *a(t)*, tandis que nos images sont des fonctions de deux variables spatiales, *s(x, y)*. Cependant, les problèmes que nous traitons sont indépendants de la dimension : l'aliassage existe aussi en dimension 1, par exemple lorsque l'on veut représenter un signal sonore par un ensemble de valeurs discrètes sur un CD audio. Dans la suite, nous utiliserons souvent des signaux de dimension 1 pour illustrer les concepts introduits, ne serait-ce que parce qu'ils sont plus facile à représenter graphiquement !

Echantillonnage et reconstruction d'un signal

La transformation d'un signal continu en un ensemble de valeurs discrètes est appelée *échantillonnage*. Ces valeurs sont calculées à des intervalles réguliers : on parle de *fréquence d'échantillonnage* pour caractériser le nombre d'échantillons par unité de signal. Par exemple, si on échantillonne un signal audio à raison de 100 échantillons par seconde, la fréquence d'échantillonnage est de 100 points par seconde ou 100Hz. Si on échantillonne une image à raison de 100 échantillons par unité de distance (par exemple le cm), on parle d'une fréquence d'échantillonnage de 100 cm^{-1} (on utilise souvent le dpi ou dots per inch, 1 inch = 2.54cm).

La transformation inverse, d'un ensemble d'échantillons en un signal continu, est appelée *reconstruction*. Nous allons voir que le signal doit avoir des propriétés particulières pour pouvoir être reconstruit correctement à partir des échantillons. En particulier, un résultat fondamental est que l'on ne peut reconstruire correctement un signal à partir d'échantillons prélevés à la fréquence *f* que si le signal ne contient aucune composante de fréquence supérieure à *f/2*. Pour comprendre ce que signifie qu'un signal contient des composantes de fréquences

données, nous allons introduire la notion de spectre et un outil mathématique fondamental du traitement du signal : la transformée de Fourier.

Lorsque l'on échantillonne un signal, il y a deux niveaux d'échantillonnage : d'abord, des échantillons sont prélevés toutes les *n* unités. Ensuite, chaque échantillon est une valeur réelle qui est ensuite représentée (on dit *numérisée*) par une valeur entière dans un intervalle donné. Par exemple, les CD audio ont une fréquence d'échantillonnage de 44kHz et chaque échantillon est représenté sur 16bits, donc dans l'intervalle -32766 à +32767. Une image affichée sur un écran est généralement échantillonnée à 72 points par pouce (dots-per-inch) et l'intensité de chaque point est représentée par une valeur entre 0 et 255. Nous ne tiendrons par compte dans la suite de la numérisation des échantillons. Bien qu'elle pose des problèmes similaires à l'échantillonnage du signal, ils sont en général beaucoup moins sensibles lorsque l'on fait du traitement d'image.

Echantillonnage ponctuel

La méthode la plus évidente pour échantillonner un signal consiste à mesurer sa valeur à chaque échantillon. Ainsi, un signal audio *a(t)* échantillonné à la fréquence *f* donne une famille d'échantillons *a(1/f), a(2/f), ... , a(n/f), ...* (figure 8). Pour une image *s(x, y)*, on a la famille d'échantillons $(s(i/f, j/f))_{i = 1.., j = 1 ..}$.

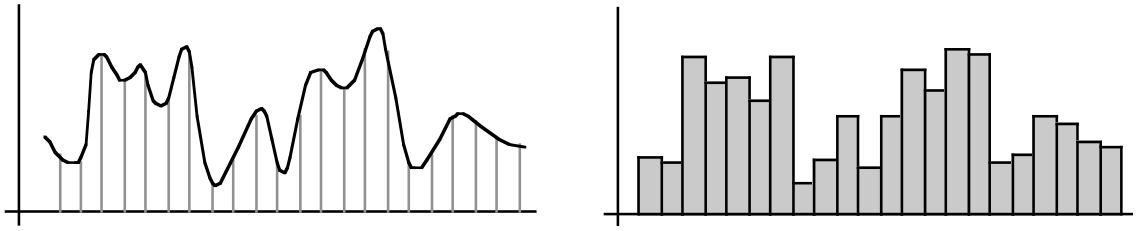


Figure 8. Echantillonnage ponctuel d'un signal

La reconstruction consiste en un "sample-and-hold" qui résulte en une fonction en escalier, ou éventuellement en une interpolation linéaire qui résulte en une fonction affine par morceaux. Dans les deux cas, la différence avec le signal d'origine est facilement perceptible.

Cette méthode présente l'inconvénient majeur que la plus grande partie du signal est ignorée lors de l'échantillonnage. Par exemple, dans une image, certains détails peuvent être complètement ignorés (figure 9).

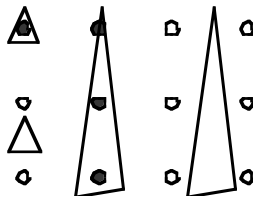


Figure 9. Problèmes de l'échantillonnage ponctuel

Echantillonnage par filtrage

Afin de prendre en compte *tout* le signal lors de l'échantillonnage, l'idée est de calculer la valeur de l'échantillon à partir de la partie du signal qui est au voisinage du point d'échantillonnage. Pour un signal audio, on calcule par exemple la moyenne du signal entre *a((i-1/2)/f)* et *a((i+1/2)/f)* (figure 10). Pour une image, on fait la moyenne sur une zone carrée qui entoure le point d'échantillonnage. Avec cette technique, tout le signal est pris en compte, et chaque point du signal

n'est pris en compte qu'une fois. La technique de reconstruction reste la même que précedemment, mais l'approximation est meilleure car les "détails" du signal ont été pris en compte.

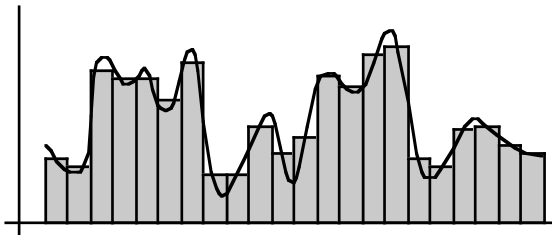


Figure 10. Echantillonnage par filtrage

Cette méthode a cependant un inconvénient qui apparaît si l'on imagine qu'un "détail" du signal se déplace d'un point d'échantillonnage au suivant (figure 11). Tant que le détail est inclus dans un pixel, seul ce pixel est affecté. Lorsque le détail change de pixel, la transition est brusque. Cet effet peut être réduit si l'on considère que la moyenne que l'on calcule pour chaque échantillon est pondérée par la distance au point d'échantillonnage. La fonction de pondération, appellée *filtre d'échantillonnage*, peut être représentée en 3D : dans l'échantillonnage ponctuel, il s'agit d'une impulsion (fonction nulle partout sauf en un point, ici le point d'échantillonnage) ; dans la première méthode ci-dessus, il s'agit d'une boîte (contribution identique de chaque point).

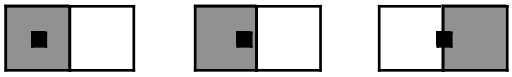


Figure 11. Problème de l'échantillonnage avec un filtre carré

La figure 12 illustre l'effet d'un filtre en forme de pyramide et d'un filtre en forme de cône. Dans ce dernier cas, les cônes de points d'échantillonnage adjacents s'intersectent, c'est-à-dire qu'un point du signal contribue à plusieurs échantillons. Comme le montre la figure, cela donne un résultat plus continu dans le cas du déplacement d'un détail.

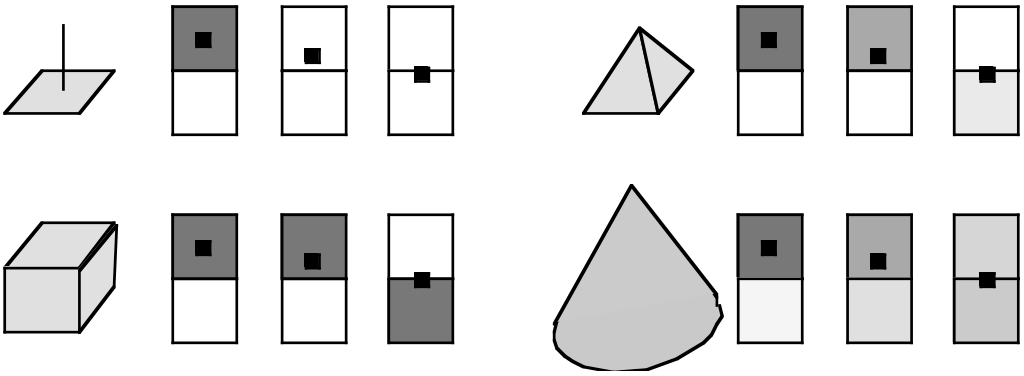


Figure 12. Echantillonnage ponctuel (en haut à gauche), en moyenne ou avec filtre carré (en bas à gauche), avec filtre en pyramide (en haut à droite), avec filtre en cône (en bas à droite).

Lorsque l'on utilise un filtre pour l'échantillonnage, il faut aussi l'utiliser pour la reconstruction. Supposons un signal audio *a(t)* et un filtre *f(t)*. On suppose que le filtre a un support *[-d, +d]* (le support est l'ensemble des valeurs de *t* pour lesquelles *f(t)* n'est pas nul). Soit un échantillon *e(k/f)*. La contribution de cet échantillon au signal reconstruit *r* est alors :

r (k/f + t) = e (k/f) * f (t) pour t dans [-d, +d], 0 ailleurs.

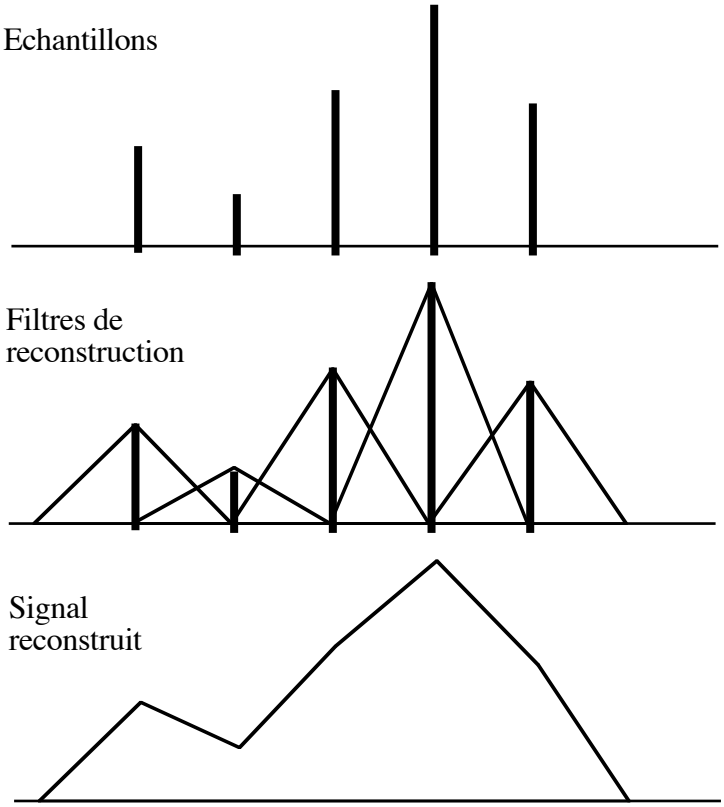


Figure 13. Reconstruction linéaire avec un filtre triangulaire

Pour une image (signal 2D), le principe est le même : soit *s(x, y)* le signal, *f(x, y)* le filtre de support *D* et *e(i/f, j/f)* un échantillon. La contribution de cet échantillon au signal reconstruit r est :

r (i/f + x, j/f + y) = e(i/f, j/f) * f(x, y)

Si l'on utilise un signal en dent de scie (ou en forme de pyramide pour un signal 2D), on retrouve la reconstruction par interpolation linéaire (figure 13). Si on utilise un filtre carré (ou une boîte pour un signal 2D), on retrouve la reconstruction par une fonction en escalier.

La question qui se pose à ce stade est : quel est le "meilleur" filtre ? Est-il possible de constuire un filtre qui permette une reconstruction parfaite du signal d'origine, ou meilleure que toute autre ? C'est maintenant qu'il nous faut introduire un peu de théorie de traitement du signal.

La théorie du traitement du signal va nous apprendre que tou signal ne peut être reconstruit. Il faut (i) qu'il soit périodique et (ii) que son spectre ne contienne pas de fréquence supérieure à la moitié de la fréquence d'échantillonnage.

Représentation fréquencielle d'un signal

Considérons un signal mono-dimensionnel *a(t)*. On dit que ce signal est *périodique* s'il existe une valeur *p*, appelée *période*, telle que, pour tout *t*, *a(t) = a(t+p)*. En d'autres termes, le signal se reproduit égal à lui-même, indéfiniement. Un exemple de fonction périodique est la fonction *sinus* ou *cosinus*.

Un théorème dû à Fourier assure que tout signal périodique peut être décomposé en une somme éventuellement infinie de fonctions sinus dont les périodes sont toutes multiples d'une période dite fondamentale du signal :

$a(t) = \Sigma_k \; a_k \sin (k\, t/f + \phi_k)$

f est la fréquence fondamentale (inverse de la période)

a_k est l'amplitude de la k-ième harmonique

ϕ_k est la phase de la k-ième harmonique

Les figures 14, 15 et 16 pages suivantes montrent quelques exemples de signaux et de leurs décompositions en fondamentale et harmoniques. Cette décomposition est appelée *spectre* du signal.

Transformée de Fourier

Il existe un moyen de calculer le spectre d'un signal. C'est la *transformée de Fourier*, qui est définie de la façon suivante :

$F(u) = \int f(x) \, [\cos(2 \pi u x) - i \sin(2 \pi u x)] \, dx$

l'intégrale porte sur l'intervalle $[-\infty, +\infty]$;

f(x) est un signal mono-dimensionnel ;

u est une variable complexe.

La transformée de Fourier d'un signal est une fonction complexe : pour une fréquence *u*, *F(u)* est de la forme $a + i\, b$. Le module de ce nombre complexe, $\sqrt{(a^2 + b^2)}$, est l'amplitude de la fréquence *u* dans le spectre de *f*. Sa phase est l'angle du nombre complexe dans une représentation polaire, ou $tan^{-1}(b/a)$.

La *transformée de Fourier inverse* permet de passer du spectre *F(u)* au signal *f(x)*. Elle est définie comme suit :

$f(u) = \int F(x) \, [\cos(2 \pi u x) + i \sin(2 \pi u x)] \, du$

On peut donc passer d'un signal (périodique) à son spectre et inversement sans perdre d'information. L'intérêt est que certaines opérations de traitement du signal sont beaucoup plus faciles à réaliser sur le spectre d'un signal que sur le signal lui-même.

Pour les signaux à deux dimensions (donc en particulier les images), l'expression de ces transformations fait intervenir des intégrales doubles :

$F(u, v) = \iint f(x, y) \, [\cos(2 \pi (u x + v y)) - i \sin(2 \pi (u x + v y))] \, dx \, dy$

$f(x, y) = \iint F(u, v) \, [\cos(2 \pi (u x + v y)) + i \sin(2 \pi (u x + v y))] \, du \, dv$

Transformée de Fourier discrète

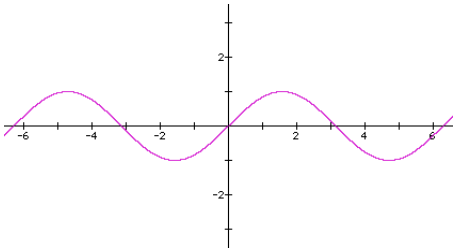
Pour calculer la transformée de Fourier d'un signal avec un ordinateur, il nous faut une expression qui puisse être traduite en un algorithme. La *transformée de Fourier discrète* nous fournit la base d'un tel algorithme :

$F(u) = \Sigma_{0 \le x \le N-1} \, f(x) \, [\cos(2 \pi u x / N) - i \sin(2 \pi u x / N)] \, dx$

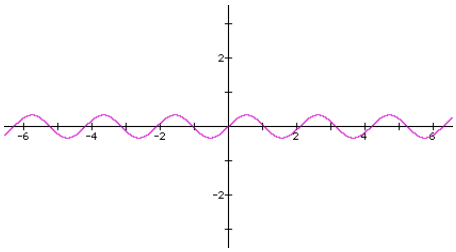
N est le nombre d'échantillons du signal ;

F(u) est calculée pour $0 \le u \le N-1$.

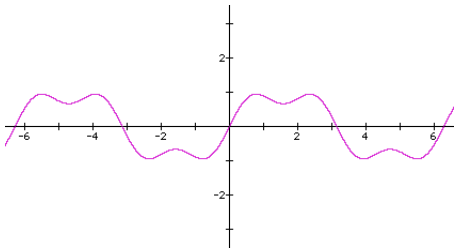
sin x



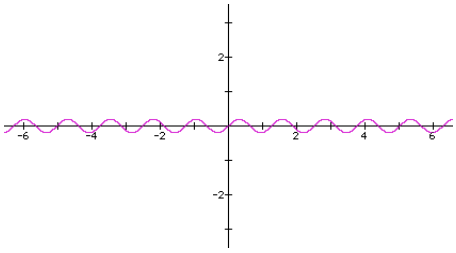
1/3 sin 3x



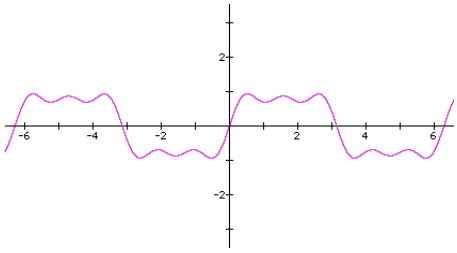
sin x + 1/3 sin 3x



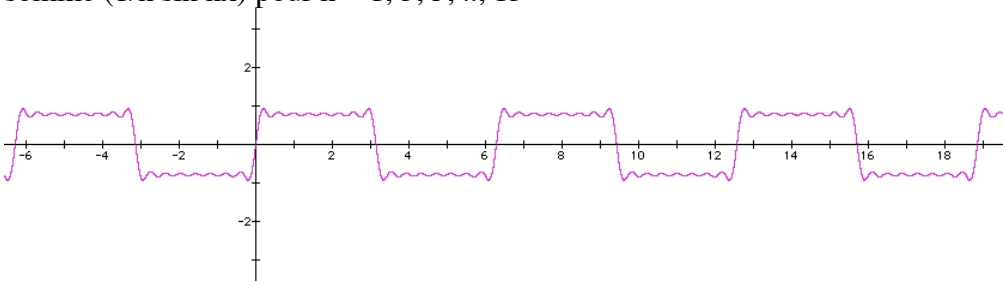
1/5 sin 5x



sin x + 1/3 sin 3x + 1/5 sin 5x



somme (1/n sin nx) pour n = 1, 3, 5, .., 15



sin x - 1/2 sin 2x + 1/3 sin 3x - 1/4 sin 4x

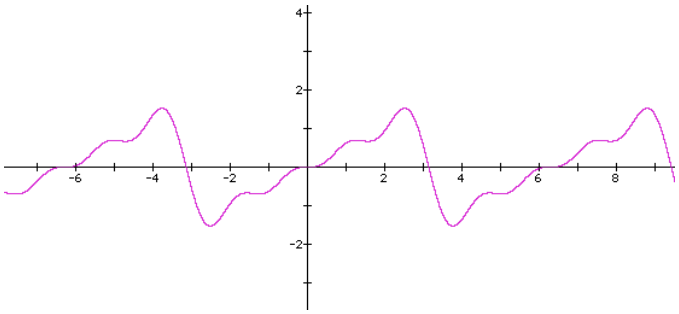
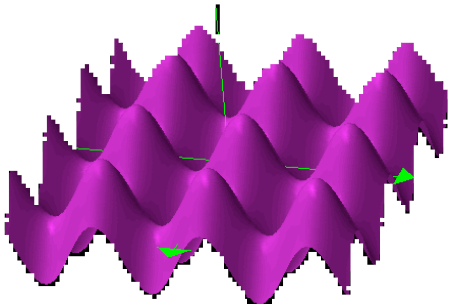
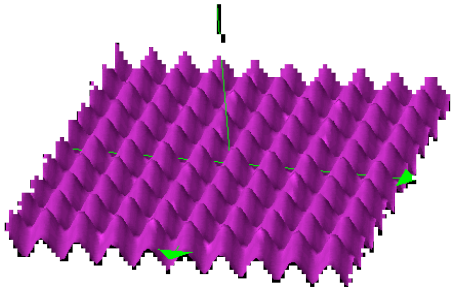


Figure 14. décompositions de fonctions carrées et dent-de-scie.

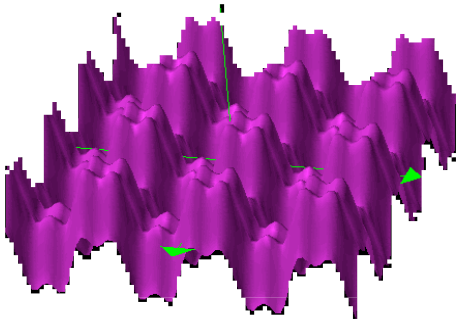
sin x + sin y



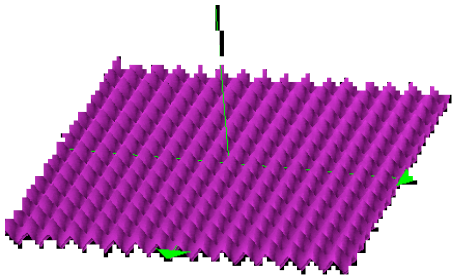
1/3 sin 3x + 1/3 sin 3y



sin x + 1/3 sin 3x + sin y + 1/3 sin 3y



1/5 sin 5x + 1/5 sin 5y



sin x + 1/3 sin 3x + 1/5 sin 5x +
sin y + 1/3 sin 3y + 1/5 sin 5x

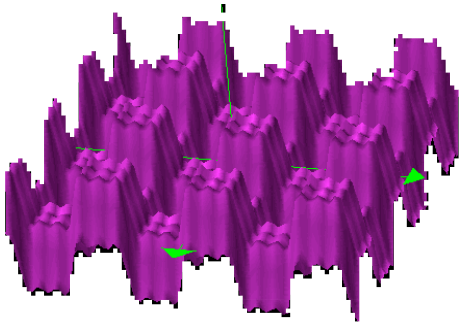


Figure 15. Décomposition de signal carré 2D.

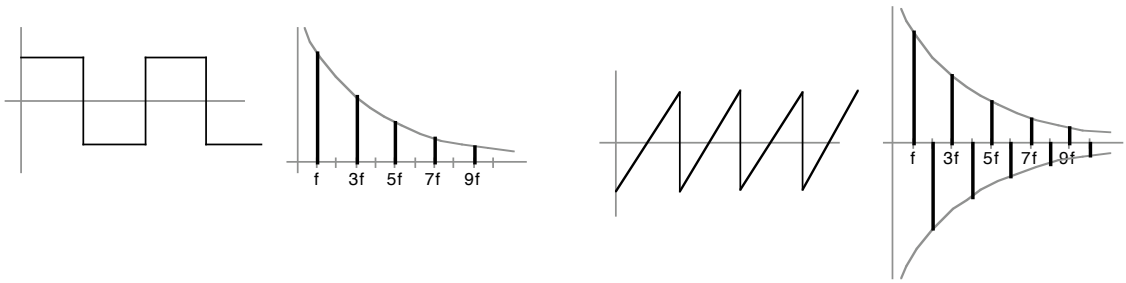


Figure 16. Spectre de signaux carrés et dent-de-scie

Pour pouvoir faire ce calcul, il faut *N* échantillons du signal, qui sont considérés comme une période d'un signal périodique. Le résultat permet de calculer les amplitudes et phases des fréquences *0* à *N-1*. On voit donc d'une part que l'on n'utilise qu'un nombre fini d'échantillons et d'autre part que l'on obtient un spectre fini. Le passage de l'un à l'autre se fait sans perte d'information.

La transformation inverse s'exprime sous la forme suivante. Elle fournit les valeurs de *N* échantillons à partir d'un spectre :

$$f(x) = 1/ N \Sigma_{0 \leq u \leq N-1} F(u) [\cos(2 \pi u x / N) + i \sin(2 \pi u x / N)] dy$$

Le calcul de la transformée de Fourier discrète (ou de son inverse) est en N^2. Cependant, un algorithme astucieux permet de réduire la complexité du calcul à *N.logN* lorsque *N* est une puissance de 2 : c'est la *transformée de Fourier rapide* ou *FFT* (Fast Fourier Transform). Cet algorithme n'est pas détaillé ici.

L'origine de l'aliassage

Le théorème de Fourier nous a permis de définir la notion de spectre d'un signal, et la transformée de Fourier nous permet de passer du signal à son spectre et réciproquement. D'autre part nous avons vu que l'échantillonnage nécessitait la définition d'un filtre, qu'il nous faut caractériser.

Un théorème fondamental du traitement du signal nous apprend qu'un signal périodique ne peut être correctement reconstruit à partir de ses échantillons que s'il ne contient pas de fréquence supérieure à la moitié de la fréquence d'échantillonnage. La fréquence maximale que l'on peut échantillonner est appelée *fréquence de Nyquist*. Pour pouvoir échantillonner correctement un signal, il faut donc enlever les fréquences supérieures à la fréquence de Nyquist. Une façon simple de réaliser cette opération consiste à :

- calculer le spectre du signal (transformée de Fourier) ;
- enlever du spectre les fréquences indésirables ;
- calculer le signal correspondant au nouveau spectre (transformée de Fourier inverse).

(Cette technique est non seulement coûteuse, mais elle est également impraticable, car pour pouvoir calculer le spectre du signal, il faut l'avoir échantillonné ...)

Dans un signal audio, les fréquences élevées correspondent à des sons aigus. A quoi correspondent-elles dans une image ? De façon générale, tout changement brusque d'intensité entre des pixels adjacents produit des fréquences élevées, comme par exemple un tracé de segment noir sur fond blanc. Une autre cause de fréquences élevées dans une image sont des motifs détaillés de petite taille, comme un échiquier dont chaque case ne ferait que quelques pixels.

Si l'on considère une ligne d'une image contenant un segment blanc sur fond blanc, on obtient un signal carré (figure 17, haut). Si l'on enlève les fréquences élevées du signal, on obtient un signal dont les variations brusques sont atténuées (figure 17, bas) :

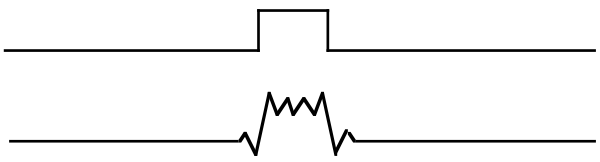


Figure 17. Effet du filtrage passe-bas sur un signal carré

Que se passe-t-il si l'on ne filtre pas les fréquences au-delà de la fréquence de Nyquist ? Comme le montre la figure 18 ces fréquences sont remplacées à la reconstruction par des *alias*, des fréquences parasites qui remplacent les fréquences élevées qui sont incorrectement représentées dans les échantillons. Si f_N est la fréquence de Nyquist et f une fréquence supérieure à f_N présente dans le signal, l'alias qui apparaît lors de la reconstruction a la fréquence $f - f_N$.

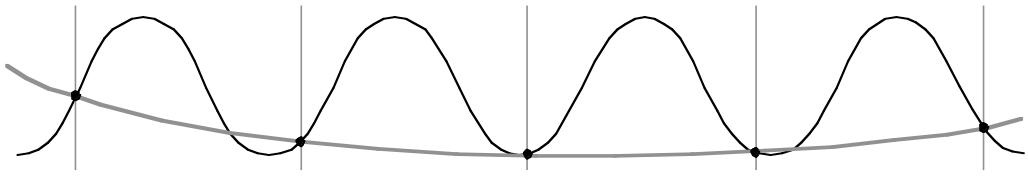


Figure 18. Origine des alias : le signal de fréquence élevée (sinusoïde) ne peut être reconstruit correctement ; les échantillons (points noirs) produisent un signal de fréquence beaucoup plus faible que l'original.

En infographie, ces alias se présentent sous deux formes principales : pour les tracés de primitives graphiques à fort constraste (segments, polygones, etc.), c'est l'effet d'escalier ou de crênelage ; pour les textures trop fines, c'est l'effet de moiré. L'effet du filtrage de ces fréquences élevées est que les transitions entre zones d'intensités très différentes sont plus "douces", moins marquées. Cela est souvent perçu comme une perte de netteté de l'image : les traits sont plus beaux, mais plus flous ; le moiré disparaît, mais le damier n'est plus net. La théorie du traitement du signal nous montre que c'est le mieux que l'on puisse faire.

Anti-aliassage par filtrage

Nous venons de voir que pour permettre un échantillonnage correct d'un signal, il fallait éliminer les fréquences supérieures à la fréquence de Nyquist. Cette opération est facile à réaliser dans le domaine fréquentiel. Il suffit de multiplier le spectre par une fonction carrée *S* :

$S(u) = 1$ si $-k \leq u \leq k$
$S(u) = 0$ sinon

Pour un signal de spectre *F(u)*, on obtient le spectre *F'(u)* suivant :

$F'(u) = F(u) \cdot S(u)$, soit
$F'(u) = F(u)$ si $-k \leq u \leq k$
$F'(u) = 0$ sinon

Comme nos signaux sont dans le domaine spatial, il faut calculer leur transformée de Fourier pour obtenir leur spectre, appliquer le filtre, et calculer la transformée de Fourier inverse pour obtenir le signal filtré. Il serait intéressant de pouvoir faire cette opération de filtrage dans le domaine fréquenciel directement.

Convolution

On définit l'opération de *convolution* de deux fonctions, notée *, de la façon suivante :

$h(x) = f(x) * g(x) = \int f(t)\, g(x - t)\, dt$
l'intégrale porte sur le support de *f*, au pire l'intervalle $[-\infty, +\infty]$.

La convolution peut être vue comme une opération de filtrage. Son interprétation géométrique est présentée figure 19 : le filtre *g(t)* est retourné horizontalement pour représenter *g(-t)*, puis translaté de *x* pour l'amener sur la portion du signal à filtrer *g(x-t)*. On multiplie alors le signal *f(t)* par le filtre translaté *g(x - t)* et l'on

calcule l'intégrale, c'est-à-dire l'aire sous la courbe obtenue. Dans l'exemple de la figure, on voit que la convolution vaut 0 lorsque le filtre translaté ne touche pas le support du signal, puis croît jusqu'à un maximum lorsque le filtre translaté est complètement inclus dans le support, puis redescend vers 0. Le triangle qui en résulte peut être vu comme une version adoucie du carré de départ.

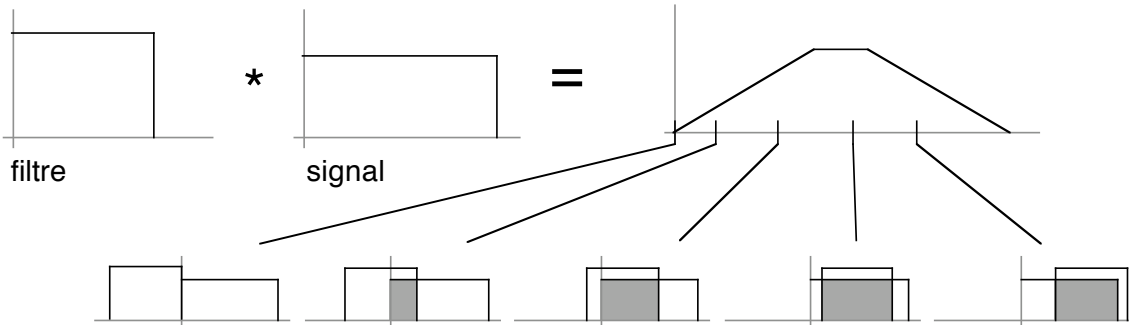


Figure 19. Principe de la convolution : retourner le filtre horizontalement par rapport à l'axe vertical, puis le décaler vers la droite de *x* et calculer l'aire (en grisé sur les images du bas) de l'intersection du filtre et du signal.

La convolution est également définie pour des signaux bidimensionnels :

$$h(x, y) = f(x, y) * g(x, y) = \iint f(s, t)\, g(x - s, y - t)\, ds\, dt$$

Son interprétation géométrique est similaire à celle décrite ci-dessus, sinon que l'on manipule des fonctions de deux variables et qu'il faut donc calculer le volume du résultat de la multiplication et non son aire.

La convolution et la multiplication s'avèrent être des opérations identiques dans des domaines différents :

- la convolution de deux signaux correspond à la multiplication de leurs spectres : $f(x) * g(x) = F(u) \cdot G(u)$;
- la convolution de deux spectres correspond à la multiplication de leurs signaux : $F(u) * G(u) = f(x) \cdot g(x)$.

Cela signifie que si l'on doit multiplier deux spectres, on peut tout aussi bien convoluer les signaux : c'est exactement ce qu'il nous faut pour réaliser le filtrage de notre signal dans le domaine spatial.

Filtrage dans le domaine spatial

Pour éliminer les fréquences supérieures à la fréquence de Nyquist, il faut appliquer un filtre dont le spectre est un carré. Pour réaliser cette opération par convolution, il faut déterminer le signal correspondant à ce spectre. Il s'agit de la fonction *sinc* (figure 20) définie comme suit :

$$\text{sinc}(x) = \sin(\pi x) / \pi x$$

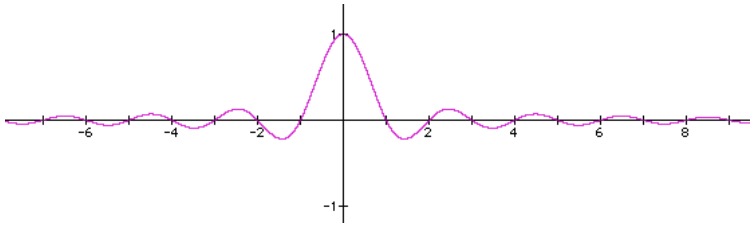


Figure 20 : fonction sinc(x)

Plus précisément, si le spectre du filtre a pour support *[-W, +W]*, son amplitude *A* doit être telle que *A = 2W*. Le signal correspondant est alors *A sinc(A x)*. Pour l'échantillonnage d'images, si l'on considère que l'on calcule un échantillon par pixel et que chaque pixel a des coordonnées entières, la fréquence d'échantillonnage est 1 et donc *W = 1/2*. Il en résulte que *A = 1* et que le filtre a bien pour signal *sinc(x)*.

Bien qu'étant à priori parfait, ce filtre a deux inconvénients. Le premier est que son support est infini, autrement dit toute l'image contribue à la valeur de chaque échantillon. Ceci rend le calcul de la convolution très coûteux (théoriquement infini). Si on tronque le filtre pour lui imposer un support fini (figure 21), on introduit de petites résonnances près de la fréquence de coupure qui peuvent dénaturer l'image. Cet effet peut être atténué en appliquant au filtre une *fenêtre F* :

f(x) = sinc(x) * F(x)

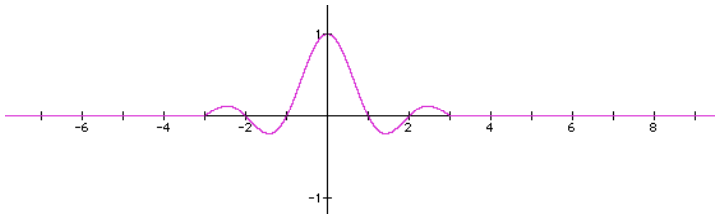


Figure 21. Filtre *sinc* avec une fenêtre carrée

Un autre inconvénient du filtre *sinc* est qu'il a des valeurs négatives. Une fois appliqué à une image, on risque d'obtenir des valeurs d'intensité négative, qu'il faudra ramener à 0. D'autres filtres n'ont pas cet inconvénient. En particulier, le filtre Gaussien a un support relativement petit qui permet d'obtenir de bons résultats de façon efficace. Le filtre Gaussien (figure 22) est de la forme suivante (** représente la fonction puissance) :

f(x) = e ** ($-x^2$ / $2s^2$)
f(x) = e ** ($-(x^2 + y^2)$ / $2s^2$)
avec s = 1 ou 2 selon le support que l'on souhaite.

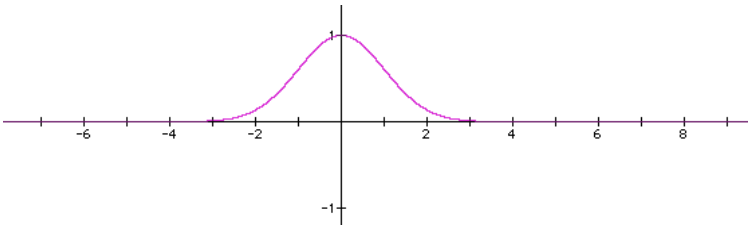


Figure 22. Filtre Gaussien

Suréchantillonnage

Nous avons à présent tous les éléments pour produire une image anti-aliassée : il suffit de calculer chaque échantillon de l'image comme la convolution de l'image et du filtre *sinc* (ou d'un autre filtre) placé au point d'échantillonnage. Pour calculer la convolution, il faut approximer le calcul d'intégrale par un calcul de somme discrète. Etant donné que le support du filtre est de l'ordre d'un pixel, il faut calculer la valeur d'image avec une résolution plus grande (figure 23) : c'est le *sur-échantillonnage*.

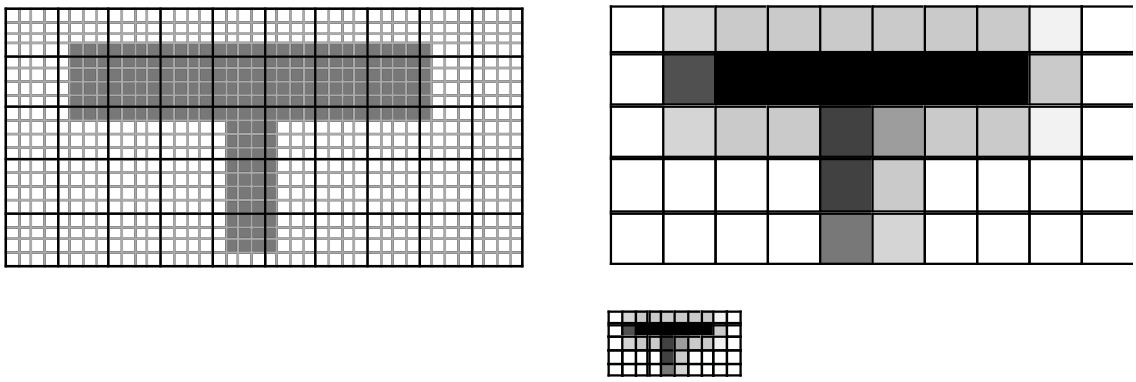


Figure 23. Principe du suréchantillonnage (gauche), image finale agrandie et en taille normale (droite).

Supposons que l'on sur-échantillonne k fois l'image (chaque pixel est décomposé en k^2 "sous-pixels") et que le support du filtre est $[-p, +p]$ en x et en y. La valeur $i(x, y)$ de l'échantillon au point de coordonnées entières (x, y) est alors :

$$i(x, y) = \Sigma_{i=-p..p}\Sigma_{j=-p..p} f(i / k, j / k)\ s(x - i / k, y - j / k)$$

f est le filtre
s est le signal
i est l'image échantillonnée

Une façon efficace mais coûteuse en mémoire de calculer cet échantillonnage consiste à calculer l'ensemble de l'image à la résolution k fois supérieure à la résolution finale, puis d'appliquer la formule ci-dessus tous les k pixels. Cette technique est efficace dans la mesure où $p > k$, car alors la plupart des sous-pixels contribuent à plusieurs pixels et n'ont cependant besoin d'être calculés qu'une fois.

<ecrire l'algorithme>

Comparaison avec les autres techniques

On peut comparer la techniques d'échantillonnage par filtrage avec les deux techniques présentées au début de ce chapitre.

L'anti-aliassage non pondéré consiste à appliquer un filtre carré à chaque pixel. Le calcul d'intersection, tel qu'il a été présenté, consiste à réaliser un sur-échantillonnage. On peut comprendre pourquoi cet algorithme donne de mauvais résultats : la transformée de Fourier d'un carré n'est autre que la fonction *sinc*, qui est très loin de filtrer les bonnes fréquences. En particulier, ce filtre renforce des fréquences au-delà de la fréquence de Nyquist, et produit donc ses propres alias.

L'anti-aliassage pondéré donne de meilleurs résultats. Le filtre appliqué est ici un triangle qui, comme le montre la figure 24, a un spectre $sinc^2$ qui s'amortit beaucoup plus vite que celui de *sinc*.

L'avantage principal de l'échantillonnage par filtrage est sa généralité : il s'applique à toute image, quelle que soit la façon dont elle est produite. Son inconvénient principal est son coût : l'image doit être calculée à une résolution plus grande que l'écran (un sur-échantillonnage de 4 donne en général de bons résultats), et tous les pixels sont traités, même si des zones de l'image ne contiennent rien. Certaines méthodes adaptent le taux de sur-échantillonnage en fonction des caractéristiques de l'image pour éviter ce problème.

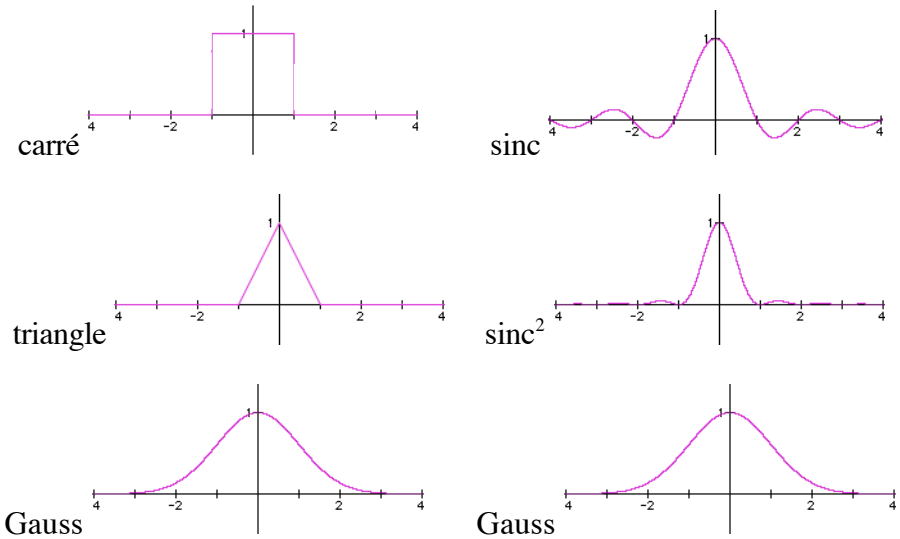


Figure 24. Principaux filtres et leurs spectres

Autres techniques

Au lieu de faire le filtrage pixel par pixel, on peut grandement améliorer les performances calculant l'image en plusieurs passes. A chaque passe on effectue un échantillonnage ponctuel, en changeant le point de vue d'une fraction de pixel entre chaque passe et en pondérant les contributions à l'image finale en fonction du décalage. Sous OpenGL, on peut utiliser le tampon d'accumulation pour faire ce calcul, voire utiliser les extensions de traitement d'image qui incluent le filtrage et qui sont implantées en hardware sur certaines cartes graphiques.

Afin d'améliorer le rendu sans avoir à augmenter le sur-échantillonnage, on utilise l'échantillonnage stochastique. Cette méthode consiste à faire varier aléatoirement les positions des points échantillonnés : en ajoutant du bruit à la grille utilisée pour calculer les valeurs de décalage, on réduit les artefacts introduits par la régularité de la grille. L'échantillonnage stochastique est facile à implémenter avec l'algorithme multipasse décrit ci-dessus (en utilisant le tampon d'accumulation sous OpenGL) : il suffit de rajouter le bruit stochastique aux déplacements du point de vue entre chaque passe.