Infographie - M1

Chapitre 9 - Anti-crénelage - Couleurs

V.Padovani

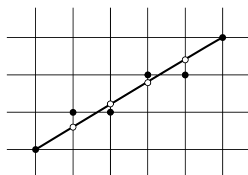
Equipe Preuves, Programmes et Systèmes, Université Paris 7

1 Anti-crénelage

L'*anti-crénelage* (en anglais *antialiasing*) est une technique permettant d'améliorer le rendu des lignes, cercles, bords de forme, etc. Son but est d' atténuer l'effet de discontinuté observable à la jonction de segments de pixels, par l'introduction de dégradés. On se limitera ici au cas de tracés monochromes, optimisés par l'emploi de niveaux de gris.

1.1 Algorithme de Gupta-Sproull

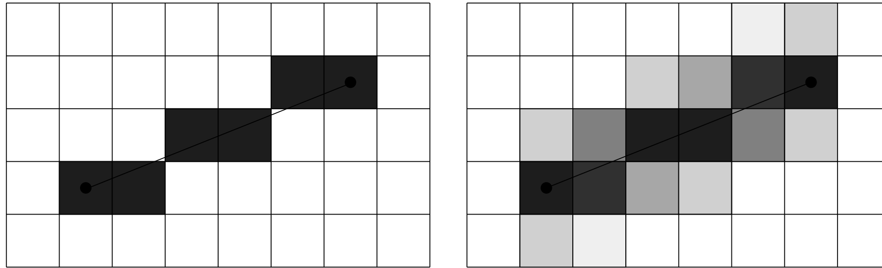
Voici un premier exemple d'algorithme de tracés de segments avec anti-crénelage, dérivé de la version "réelle" de l'algorithme de Bresenham (c.f. Cours n°2). Rappelons que le principe de l' algorithme de Bresenham est de garder en mémoire la différence entre l'ordonnée courante et l'ordonnée du point du segment réel d'abscisse courant. Dès que celle-ci dépasse 1/2, on incrémente l'abscisse courante.



Soi $[AB]$ le segment à afficher. Comme dans Bresenham, on suppose ce segment à coordonnées entières, avec $x_A < x_B$ et une pente comprise entre 0 et 1. On suppose d'autre part donnée une fonction f associant à tout réel d positif un niveau de gris. Cette fonction doit être décroissante, renvoyer un niveau de gris maximal pour $d = 0$, et une valeur nulle dès que d dépasse une certaine borne (1.5 dans la version historique de l'algorithme). On peut prendre par exemple une fonction variant linéairement du niveau de gris maximal à 0 entre 0 et 1.5.

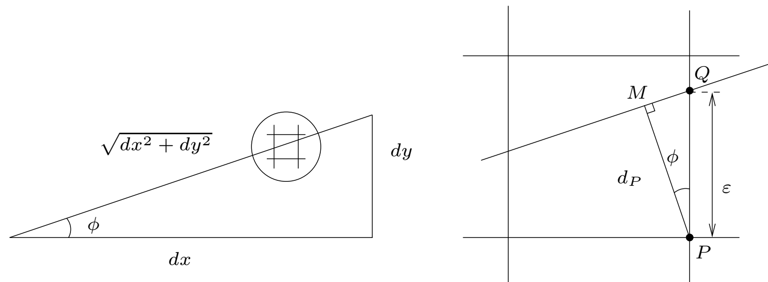
Soit à présent P un pixel qui est soit l'un des pixels affiché par l'algorithme de Bresenham, soit immédiatement au dessus ou en dessous de l'un de ces pixels. Soit d_P la distance entre P et le segment réel. On associe à P le niveau de gris $f(d_p)$.

Avec cette méthode, le niveau de gris associé à un point dépend seulement de sa distance au segment réel. Les pixels les plus sombres seront ceux affichés par l'algorithme de Bresenham. Les pixels au dessus ou en dessous de ces pixels seront affichés avec un niveau de gris moindre, proportionnel à la distance des ces pixels au segment réel.



Le calcul de d_P peut être fait comme suit. Soit $dx = x_B - x_A$. Soit $dy = y_B - y_A$. Soit Q le point du segment réel de même abscisse que P. Soit $\varepsilon = |x_Q - x_P|$. On a $d_P = \varepsilon \times \cos\phi$, où ϕ est l' angle formé par $\vec{PQ}$ et $\vec{PM}$. On a aussi $\cos\phi = dx/\sqrt{dx^2 + dy^2}$, et donc

$$d_P = \varepsilon \times \frac{dx}{\sqrt{dx^2 + dy^2}}$$

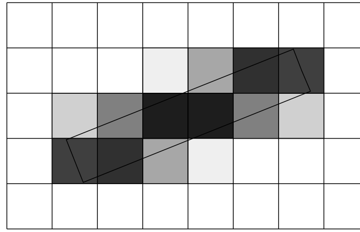


Il est également possible de calculer d_p de manière incrémentale à chaque passage d'une abscisse à la suivante. D'autre part, s'il n'y a qu'un nombre fini de valeurs de niveaux de gris, (ce qui est toujours le cas dans une implémentation réelle), cet algorithme peut être entièrement discrétisé. Le niveau de gris associé à une distance donné peut dans ce cas être extrait d'une simple table.

Noter que cette méthode ne résout pas le problème de l'affichage des extrêmités du segment (pour lesquels il faut d'ailleurs faire un choix : bord carré, bord arrondi, etc).

1.2 Anti-crénelage par calcul d'aire

La méthode suivante est applicable dès que l'on souhaite représenter une forme quelconque, mais dont la description permet, pour chaque pixel, de déterminer l'aire de pixel couverte par cette forme. Soit par exemple R un rectangle. Pour chaque pixel P, soit A_P l'aire de la portion de ce pixel intérieure à R. En supposant chaque pixel de côté 1, cette aire est comprise entre 0 et 1. On attribue à chaque pixel le niveau de gris $G \times (1 - A_P)$, ou G est le niveau de gris maximal. Le résultat aura par exemple la forme suivante :



Cette méthode donne de bons résultats, mais peut, en fonction de la forme traitée, demander des calculs lourds.

1.3 Anti-crénelage par suréchantillonage

La méthode suivante, très générale, est fréquemment employée dans le calcul de rendus d'images. L'idée est de construire, pour calculer le rendu d'une image à une résolution donnée, une image virtuelle, ou "super-image", de résolution plus grande (triple, quintuple, etc.), puis de transformer sa résolution en la résolution-cible en mélangeant les teintes des "super pixels" couvrant un pixel de l'image cible. Le mélange se fait en pondérant les teintes de ces pixels par les valeurs d'une matrice appelée *matrice de filtrage*. Si par exemple chaque pixel est couvert par 3×3 super-pixels, on peut prendre la matrice suivante :

$$F = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

La somme des coefficients est ici égale à 16. Si un pixel p de l'image cible est couvert par les super-pixels $(p^i_j)_{i,j \in [0,\ldots 2]}$ dont les niveaux de gris sont $(g^i_j)_{i,j \in [0,\ldots 2]}$, avec la disposition suivante :

p^0_0	p^0_1	p^0_2
p^1_0	p^1_1	p^1_2
p^2_0	p^2_1	p^2_2

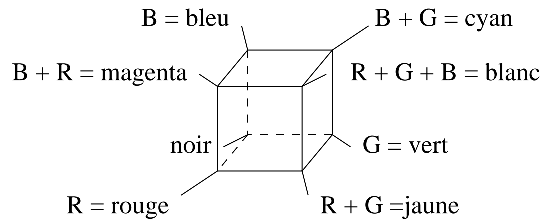
le pixel p de l'image cible sera de niveau

$$\frac{1}{16} \times \Sigma^2_{i=0} \Sigma^2_{j=0} F(i,j) \times g^i_j$$

2 Modèles de couleurs

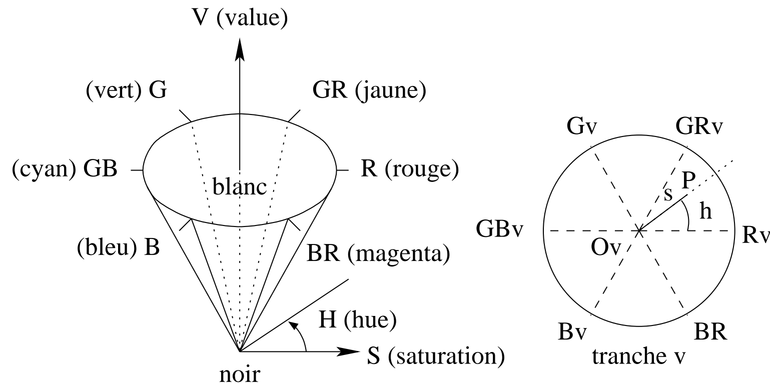
Modèle RGB

Dans le modèle RGB, chaque couleur est représentée par un triplet $(r, g, b) \in [0 \ldots 1] \times [0 \ldots 1] \times [0 \ldots 1]$ spécifiant un certain mélange de rouge (R), vert (G), et bleu (B). A chaque triplet correspond un unique point dans l'espace de couleurs représenté par le cube ci-dessous :



Modèle HSV

Dans le modèle HSV, l'espace des couleurs est représenté par un cône de tranche circulaire (parfois abusivement dessiné comme un cône à tranche hexagonale, mais les calculs se font toujours dans un cône cylindrique). Chaque point du cône est spécifiée par un triplet $(h, s, v) \in [0° \ldots 360°[\times[0 \ldots 1] \times [0 \ldots 1]$ (Hue, Saturation, Value)



- La composante v (value) d'un point est la distance entre le sommet du cône et la tranche du cône orthogonale à l'axe et contenant ce point. Elle spécifie la luminosité (sombre, claire) de la couleur, soit, pour une couleur RGB, le maximum de ses trois composantes. Par convention, si $v = 0$, alors h et s valent 0 et la couleur spécifiée est le noir.

- La composante s (saturation) est la distance entre le point et l'axe du cône (soit la distance entre les point P et O_v dans la figure ci-dessus). Elle spécifie l'écart entre la couleur et un niveau de gris de même luminosité, Si une couleur RGB a pour composantes maximales et minimales *max* et *min*, la saturation associée est $s = 1 - (min/max)$.

- La composante h (hue) est l'angle formé par les vecteurs $\vec{O_v R_v}$ et $\vec{O_v P}$. Par convention, h vaut 0 si $s = 0$ (c'est-à-dire si $P = O_v$). Soit (r, g, b) une couleur RGB, de composantes maximales et minimales *max* et *min*. Soit $\delta = max - min$. L'angle h associé est fonction de la couleur majoritaire, et de la différence entre les deux autres couleurs.

– si $max = r$, la couleur est proche du rouge (R), entre magenta (BR) et jaune (GR). On a $h = 60 \times ((g - b)/\delta) \pmod{360}$.
– si $max = g$, la couleur est proche du vert (G), entre jaune (GR) et cyan (GB). On a $h = 120 + 60 \times ((b - r)/\delta) \pmod{360}$.
– si $max = b$ la couleur est proche du bleu (B), entre cyan (GB) et magenta (BR). On a $h = 240 + 60 \times ((r - g)/\delta) \pmod{360}$.

Noter que dans chaque cas, la différence entre les deux couleurs non majoritaires est comprise entre $-\delta$ et δ, donc, sa valeur divisée par δ est comprise entre -1 et 1.

2.1 Passage d'un modèle à l'autre (en OCaml)

```
(* algorithme classique *)
let rgb_to_hsv (r, g, b) =
  let max_c = max (max r g) b in
  if max_c = 0. then (0.,0.,0.)
  else
    let min_c = min (min r g) b in
    let s = 1. -. min_c/.max_c in
    if  s = 0.0 then (0., 0., max_c)
    else
      let d = max_c -. min_c in
      let h =
        60. *.
          (if max_c = r then (g -.b)/.d else
           if max_c = g then 2.0 +. (b -.r)/.d else
                             4.0 +. (r -.g)/.d) in
      let h = if h < 0. then h +. 360. else h in
         (h, s, max_c)
;;

(* algorithme classique - correct, mais peu lisible. *)
let hsv_to_rgb (h, s, v) =
  if s = 0. then (v, v, v) else
  let h_0 = h /. 60. in
  (* partie entiere de h_0 : [0...5] *)
  let i = int_of_float h_0
  (* partie decimale de h_0 : [0 .. 1[ *)
  and f = h_0 -. (floor h_0) in
  (* valeur de la couleur minimale *)
  let p = v *. (1.0 -. s)
  (* premiere valeur possible de la couleur intermediaire *)
  and q = v *. (1.0 -. s *. f)
  (* seconde valeur possible de la couleur intermediaire   *)
  and t = v *. (1.0 -. s *. (1.0 -. f)) in
```

```
  match i with
      0 -> (v, t, p)
    | 1 -> (q, v, p)
    | 2 -> (p, v, t)
    | 3 -> (p, q, v)
    | 4 -> (t, p, v)
    | 5 -> (v, p, q)
;;

hsv_to_rgb (255., 0.5, 0.8);;

(* autre methode equivalente, plus longue, mais     *)
(* peut-etre plus lisible : on effectue l'inverse   *)
(* exact de rgb_to_hsv                              *)
let hsv_to_rgb (h, s, v) =
  if s = 0. then (v, v, v) else
    (* valeurs deduites de la conversion inverse *)
    let max_c = v in
    let min_c = max_c *. (1.0 -. s) in
    let d = max_c -. min_c in

    (* la valeur du k ci-dessous est :
       - dans [0...1] si h est dans [-60...60] (mod 360)
       - dans [1...2] si h est dans [60 ..180]
       - dans [2...3[ si h est dans [180..300[
       la partie entiere de k vaut donc
       0 si rouge est dominant
       1 si vert  est dominant
       2 si bleu  est dominant
    *)
    let k = ((mod_float (h +.60.) 360.)/. 120.) in

    (* la valeur du f ci-dessous est dans [-1 ... 1],
       et vaut :
        - (g -.b)/.d si rouge est dominant,
             positif si bleu  est minimal,
             negatif si vert  est minimal

        - (b -.r)/.d si vert  est dominant
             positif si rouge est minimal,
             negatif si bleu  est minimal,

        - (r -.g)/.d si bleu  est dominant
             positif si vert  est minimal,
             negatif si rouge est minimal
    *)
```

```
    let f = 2.0 *. ((k -. (floor k)) -. 0.5) in
    (* valeur de la couleur intermediaire selon le signe de f *)
    let int_c =
      if (f > 0.)
      then (f *. d) +. min_c
      else  min_c -. (f *. d)
    (* placement dans le resultat des composantes,    *)
    (* min_c, int_c, max_c en fonction de la partie   *)
    (* entiere de k et du signe de f                  *)
    in match (int_of_float k), f > 0. with
        0, true  -> (max_c, int_c, min_c)
      | 0, false -> (max_c, min_c, int_c)
      | 1, true  -> (min_c, max_c, int_c)
      | 1, false -> (int_c, max_c, min_c)
      | 2, true  -> (int_c, min_c, max_c)
      | 2, false -> (min_c, int_c, max_c)
;;
```