Les algorithmes de base du graphisme

Table des matières

1 Traçage **2**
- 1.1 Segments de droites 2
  - 1.1.1 Algorithmes simples 3
  - 1.1.2 Algorithmes de Bresenham (1965) 4
  - 1.1.3 Généralisation aux huit octants 7
- 1.2 Cercles 8

2 Remplissage **11**
- 2.1 Test de l'appartenance d'un point à polygone 11
- 2.2 Algorithme de balayage de lignes 12
  - 2.2.1 Recherche des intersections 12
  - 2.2.2 Structures de données 13
  - 2.2.3 Algorithme 16
- 2.3 Remplissage de régions 17
  - 2.3.1 Connexité 17
  - 2.3.2 Algorithmes 18

3 Antialiasing **18**
- 3.1 Algorithmes de tracé de segments corrigés 19
- 3.2 Sur-échantillonnage de l'image 20

1 Traçage

Le traçage de primitives 2D (segments, cercle, ...) est une des fonctionnalités de base du graphisme 2D qui est utilisée par la plupart des applications en synthèse d'images. Le problème qui se pose est la représentation discrète (pixels) des primitives 2D. Les aspects fondamentaux des algorithmes correspondants sont la rapidité d'exécution et la précision de l'approximation discrète réalisée.

Les solutions pour résoudre le problème du tracer de primitives sont :

1. Méthodes brutes (naïve) : discrétisation de la primitive en n points, puis approximation au pixel le plus proche pour chacuns des n points. Peu efficace et peu précise.
2. Méthodes de l'analyseur différentiel numérique : utilisation de la tangente à la courbe tracée pour déterminer à partir d'un point tracé la position du point suivant. Méthode plus efficace pour l'implantation matérielle.
3. Méthodes incrémentales : basé sur l'analyseur différentiel numérique. La position du point suivant est choisie de façon à minimiser l'erreur d'approximation. Méthode optimale pour le tracé de segments de droites.

1.1 Segments de droites

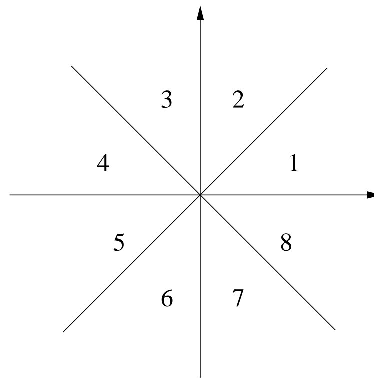


FIG. 1 – Les huit octants.

On réduit la complexité du problème en traitant tout d'abord le tracer de segment appartenant au premier octant. Les autres se déduisant ensuite par symétries.

1.1.1 Algorithmes simples

Une droite est définie par l'équation :

$$y = mx + B,$$

avec $m = \Delta y / \Delta x$. En partant du point le plus à gauche et en incrémentant la valeur de x_i de 1 à chaque itération, on calcule :

$$y_i = mx_i + B,$$

puis on affiche $(x_i, E(y_i + 0.5)$, le pixel le plus proche en x_i du segment traité. Cette méthode est inefficace en raison du nombre d'opérations nécessaires à chaque itération.

En remarquant que :

$$y_{i+1} = mx_{i+1} + B = m(x_i + \Delta x) + B = y_i + m\Delta x,$$

on peut donc à partir de la position (x_i, y_i) déterminer la position du point suivant de manière incrémentale :

$$\begin{cases} x_{i+1} = x_i + 1, \\ y_{i+1} = y_i + m, \end{cases}$$

L'algorithme correspondant :

```
void segment(x0, y0,                /* pt de depart gauche */
                  x1, y1,              /* pt d'arrivee droit  */
                  valeur)              /* valeur des pixels   */
{
 int x;
 double dx = x1 - x0;
 double dy = y1 - y0;
 double m = dy / dx ;
 double y = y0;

 for(x = x0; x <= x1 ; x++){
  AfficherPixel(x, (int) (y +0.5), valeur);
  y += m
 }
} /* fin segment */
```

dit de l'analyseur différentiel numérique. L'analyseur différentiel numérique est un système utilisé pour résoudre les équations différentiels par des méthodes numériques. Les valeurs successives de x et y sont déterminées en incrémentant simultanément x et y par de petites valeurs proportionnelles aux premières dérivées en x et y (dans notre cas 1 et m).

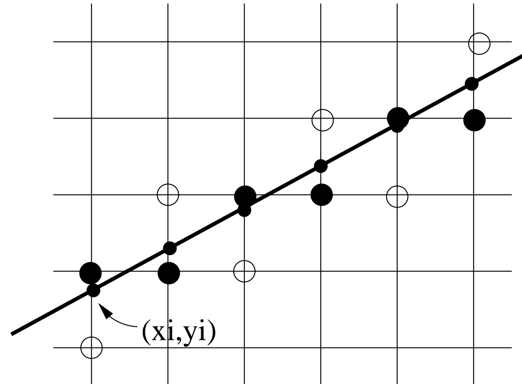


FIG. 2 – Segment dessiné avec la méthode de l'analyseur différentiel numérique.

Les inconvénients de cette approche sont :

1. Calcul de la partie entière,
2. y et m sont réels.

1.1.2 Algorithmes de Bresenham (1965)

L'idée est de mesurer l'erreur à chaque itération (distance du pixel à la droite) et de choisir la meilleure approximation. On teste pour cela le signe de $d_1 - d_2$.

$$\begin{cases} e = d_2 - d_1 \geq 0 & NE, \\ e < 0 & E. \end{cases}$$

Au premier point $(x_0 + 1, y_0 + m)$:

$$e_1 = (d_2 - d_1)/2 = d_2 - 0.5 = m - 0.5,$$

Ensuite le calcul de l'erreur se fait de manière incrémentale, à chaque itération :

$$\begin{cases} NE : e = e + m - 1, \\ E : e = e + m. \end{cases}$$

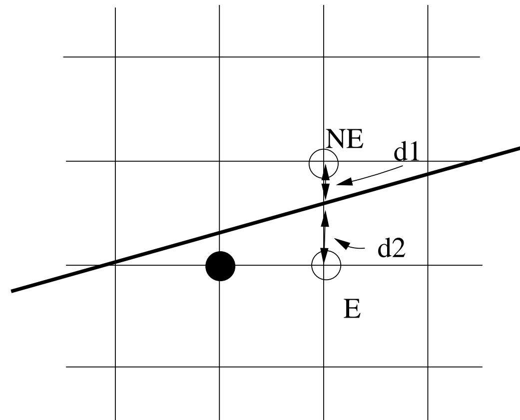


FIG. 3 – Algorithme de Bresenham : le choix se fait entre le point Est et le point Nord-est à chaque itération en fonction de l'erreur $d_2 - d_1$.

```
void Bresenham(x0, y0,               /* pt de depart gauche */
                    x1, y1,               /* pt d'arrivee droit  */
                    valeur)               /* valeur des pixels   */
{
 int x;
 double m = y1 - y0 / x1 - x0 ;
 double e = -0.5;
 int y = y0;

 for(x = x0; x <= x1 ; x++){
  e += m ;
  AfficherPixel(x, y, valeur);
  if(e >= 0){
    y += 1;
    e -= 1;
  }
 }
} /* fin Bresenham */
```

Dans l'approche précédente seule l'erreur et le coefficient directeur de la droite sont réels. Une amélioration possible consiste à effectuer toutes les opérations sur

des entiers. Pour cela, il suffit de remarquer que :

$$\begin{cases} NE : e = e + \Delta y/\Delta x - 1, \\ E : e = e + \Delta y/\Delta x. \end{cases}$$

$$\begin{cases} NE : \Delta x * e = e + \Delta y - \Delta x, \\ E : \Delta x * e = e + \Delta y; \end{cases}$$

$$\begin{cases} NE : e' = e' + \Delta y - \Delta x, \\ E : e' = e' + \Delta y; \end{cases}$$

avec : $e'_1 = e_1 * \Delta x = \Delta y - \Delta x/2$.
D'où la procédure :

```
void BresenhamEntier(x0, y0,              /* pt de depart gauche */
                         x1, y1,              /* pt d'arrivee droit
                         valeur)              /* valeur des pixels
{
 int x;
 int y = y0;
 int dx = x1 - x0;
 int dy = y1 - y0;
 int incrE = 2 * dy ;
 int incrNE = 2 * (dy - dx);
 int e = 2 * dy - dx;

 for(x = x0; x <= x1 ; x++){
  AfficherPixel(x, y, valeur);
  if(e >= 0){
    y += 1;
    e += incrNE;                                /* pixel Nord-Est *
  }
  else
    e += incrE;                                 /* pixel Est */
 }
} /* fin BresenhamEntier */
```

Bresenham a montré que cet algorithme correspond à la meilleure approximation de la droite (erreur minimum).

1.1.3 Généralisation aux huit octants

La généralisation se fait en choisissant différentes valeurs d'incrémentation pour x et y et en itérant sur x ou y en fonction de la valeur du coefficient directeur (supérieur ou inférieur à 1).

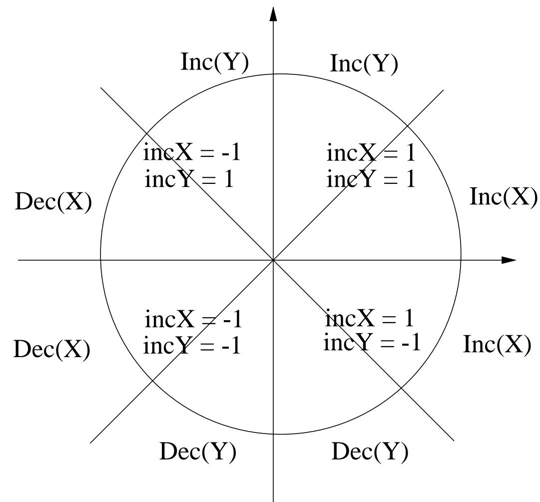


FIG. 4 – Généralisation de l'algorithme de Bresenham aux 8 octants.

```
void BresenhamGeneral(x0, y0, x1, y1, valeur)
{
 int dx = x1 - x0;
 int dy = y1 - y0;
 int IncX, IncY;

 if (dx > 0)
   IncX = 1;
 else{
   IncX = -1;
   dx = -dx;
 }
 if(dy > 0)
   IncY = 1;
 else{
   IncY = -1;
   dy = -dy;
 }

 if(dx > dy)
  /* algorithme normal */
 else
  /* inverser x et y dans l'algorithme */

} /* fin BresenhamGeneral */
```

Exercice : implanter la fonction Bresenham et la tester sur des images PNM, ou avec la fonction DrawPixel sous MESA.

1.2 Cercles

La procédure de tracer d'un cercle s'effectue suivant les principes décrit précédemment, à savoir :

- procédure incrémentale,
- découpage de l'espace en octants.

Nous traitons cette fois-ci les pixels du deuxième octant. L'erreur est calculée en évaluant la valeur de la fonction $F(x, y, R) = x^2 + y^2 - R^2$ au point milieu M entre deux pixels. Cette valeur étant positive à l'extérieur du cercle et négative

à l'intérieur. Si le point milieu M est à l'extérieur du cercle, alors le pixel SE est le plus proche du cercle. Dans le cas contraire, le pixel E est le plus proche (voir figure 5).

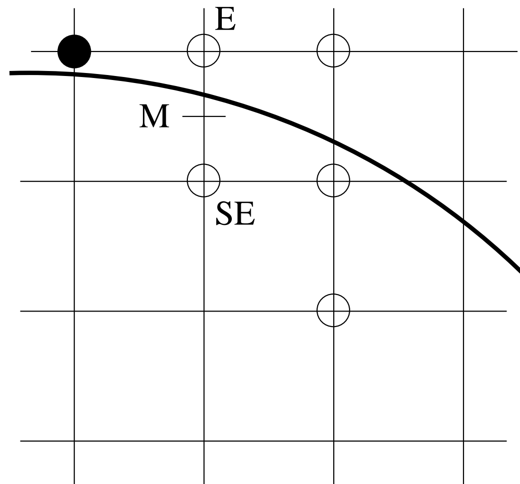


FIG. 5 – Tracer de cercle

De la même manière que pour les segments de droites, le calcul de l'erreur se fait de manière incrémentale. Au pixel affiché (x_i, y_i), la position du point M (pour le point suivant) dépend du signe de e_i :

$$e_i = F(x_i + 1, y_i - 1/2, R)$$

, si $e_i > 0$ (M à l'extérieur) alors le pixel à afficher est SE est le prochain point milieu sera $(x_i + 2, y_i - 3/2)$. La valeur e_{i+1} s'écrit :

$$e_{i+1} = F(x_i + 2, y_i - 3/2, R) = e_i + 2x_i - 2y_i + 5.$$

si $e_i < 0$ (M à l'intérieur) alors le pixel à afficher est E est le prochain point milieu sera $(x_i + 2, y_i - 1/2)$. La valeur e_{i+1} s'écrit :

$$e_{i+1} = F(x_i + 2, y_i - 1/2, R) = e_i + 2x_i + 3.$$

L'algorithme de tracer de cercle s'écrit donc :

```
void MidpointCircle(int rayon, int valeur)  /* le centre du cercle es
```

```
 {                                                    /* a l'origine  */
  int x = 0;
  int y = rayon;
  double d = 5.0/4.0 - rayon;

  AfficherPixel(x,y,valeur);

  while(y > x){       /* octant 2 */
    if(d < 0)                                 /* pixel suivant E */
      d += 2.0*x+3.0;
    else{                                     /* pixel suivant SE */
      d += 2*x-2*y+5.0;
      y -= 1;
    }
   AfficherPixel(x,y,valeur);
   x += 1;
  }
 } /* fin MidpointCircle */
```

Cette approche nécessite des opérations réelles pour le calcul de d. En remarquant que le rayon a une valeur entière et que les incréments sur d ont aussi des valeurs entières, on peut alors substituer, comme valeur initiale de d, la valeur $1 - R$ et effectuer des opérations strictement entières. Une autre optimisation de l'algorithme consiste à gérer deux variables de décision e_E et e_{SE} au lieu d'une seule basées sur des différences d'ordre 2 (exercice).

La généralisation aux huit octants se fait en remplaçant la procédure AfficherPixel() de l'algorithme par la procédure suivante :

```
void AfficherPixelCercle(int x, int y, int valeur)
{
  AfficherPixel(x,y,valeur);
  AfficherPixel(y,x,valeur);
  AfficherPixel(y,-x,valeur);
  AfficherPixel(x,-y,valeur);
  AfficherPixel(-x,-y,valeur);
  AfficherPixel(-y,-x,valeur);
  AfficherPixel(-y,x,valeur);
```

```
  AfficherPixel(-x,y,valeur);
}
```

Exercice : développer une approche similaire pour l'ellipse.

2 Remplissage

Les écrans matriciels permettent la représentation des surfaces et non plus uniquement des segments (écrans vectoriels). Le réalisme en est bien sur augmenté et cela est beaucoup utilisé en graphisme en particulier dans le domaine de la CAO. Le coût de la génération de surfaces devient donc un aspect fondamental des algorithmes et nécessite des méthodes performantes pour effectuer le remplissage.

Nous allons voir ici deux approches pour effectuer le remplissage, la première s'applique aux polygones et est basée sur le test de l'appartenance d'un point au polygone. La deuxième traite du remplissage de régions définies par des contours fermés et est basée sur des approches récursives.

2.1 Test de l'appartenance d'un point à polygone

Une méthode naïve consiste à faire la somme des angles définis par les segments de droites connectant le point à traiter et les sommets ordonnés du polygone. Cette somme est nulle pour un point extérieur et égale à $2 \times \pi$ pour un point intérieur au polygone (voir figure 6).Cette approche est peu efficace en raison du nombre d'opérations nécessaire au calcul des angles.

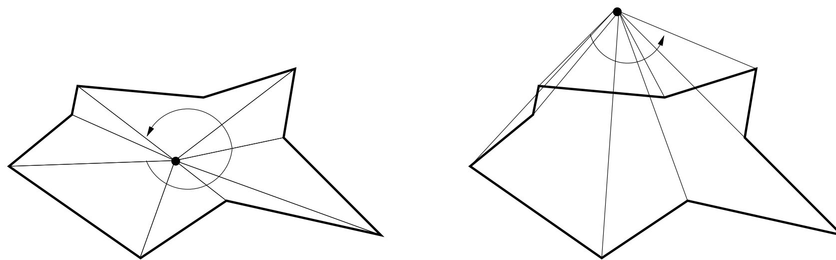


FIG. 6 – Points intérieurs : $\Sigma \ angles = 2\pi$; points extérieurs : $\Sigma \ angles = 0$.

Une autre approche consiste à calculer l'intersection d'une demi-droite infinie partant du point à traiter. Dans le cas d'un point intérieur, le nombre d'intersections entre toute demi-droite infinie issue de ce point et les cotés du polygone sera impair alors que dans le cas d'un point extérieur, le nombre d'intersection sera pair (voir figure 7).

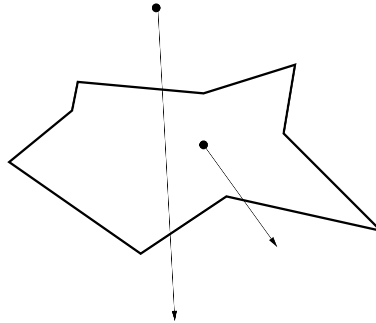


FIG. 7 – Points intérieurs : nombre d'intersections impair, extérieur : nombre d'intersections pair.

Les sommets du polygone constituent un cas particulier d'intersection. En effet, suivant le cas l'intersection est simple : les deux cotés du polygone au sommet sont de part et d'autre de la demi-droite, ou double : les deux cotés du polygone sont du même coté de la demi-droite.

2.2 Algorithme de balayage de lignes

Ce type d'algorithme consiste à balayer les lignes horizontales appartenant au rectangle englobant le polygone à traiter, et à afficher les pixels intérieurs au polygone sur chaque ligne.

2.2.1 Recherche des intersections

La recherche des intersections des lignes de balayage avec le polygone se fait en calculant l'approximation des segments à l'aide d'un algorithme de traçage et en mémorisant les point écrans correspondants.

À chaque itération, et pour chaque coté du polygone :

$$\begin{cases} y_{i+1} = y_i + 1 \\ x_{i+1} = x_i + 1/m. \end{cases}$$

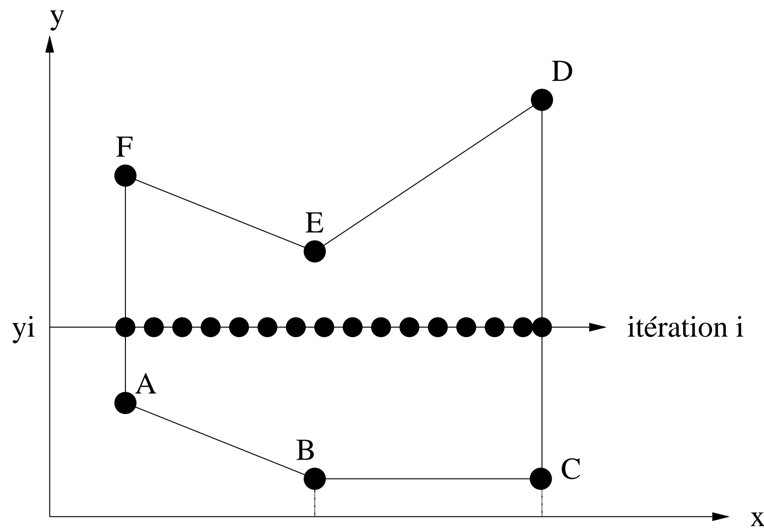
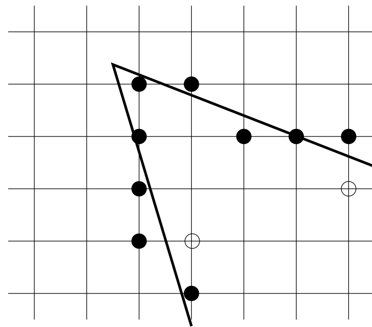


FIG. 8 – Balayage de lignes.

☞ Pour chaque coté impliqué, le calcul nécessite uniquement les valeurs de x_i et de $1/m$.



En raison de l'approximation de l'algorithme de traçage, les points d'intersections peuvent être à l'extérieur du polygone.
$\longrightarrow$ Approximation au pixel intérieur.

2.2.2 Structures de données

L'algorithme fait usage du calcul des intersections présenté et maintient à jour une table active de ces intersections.

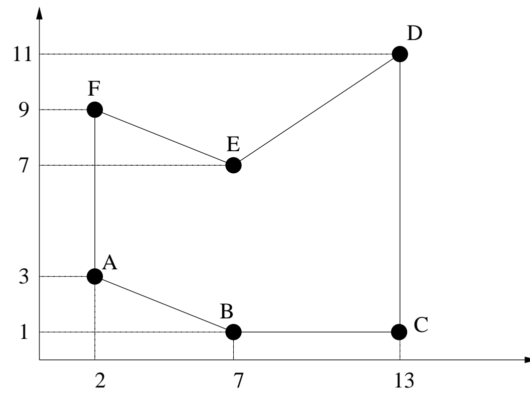


FIG. 9 – Exemple de polygone traité.

Structures de données : 2 tables.

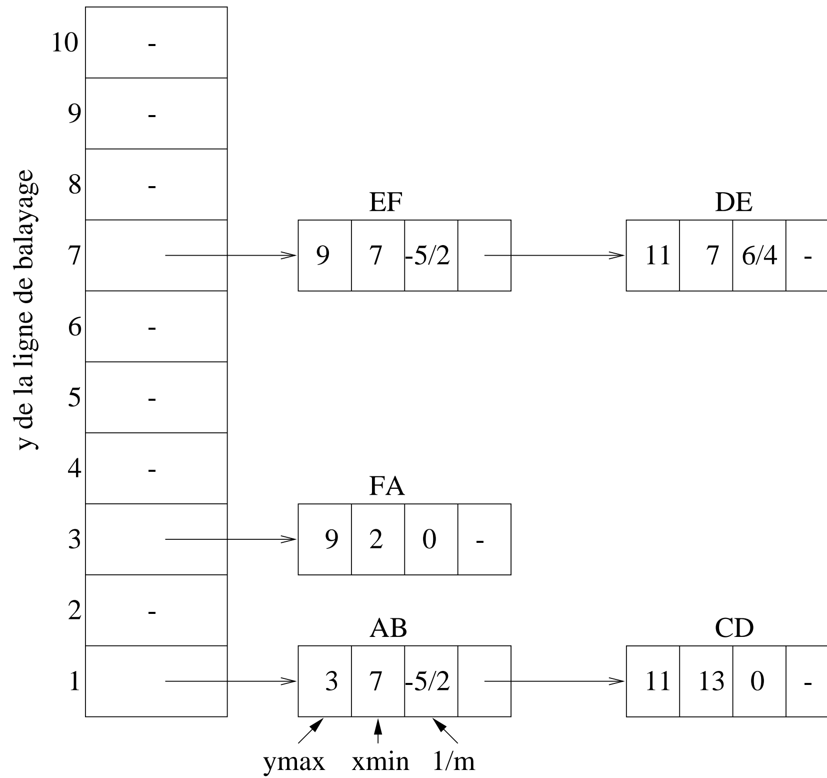


FIG. 10 – La table des cotés : contient les cotés du polygone triés suivant les y croissants.

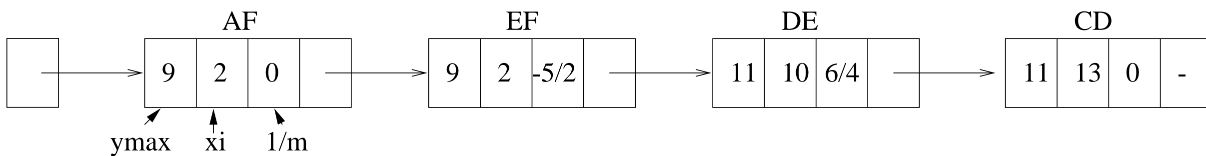


FIG. 11 – La table active : contient les cotés actifs, ici à l'itération $i = 9$ ($y_i = 9$).

2.2.3 Algorithme

1. y est initialisé à la valeur de la première entrée de la table des cotés
2. répéter jusqu'à que la table active soit vide

   2.1 mettre dans la table active les entrées de la table des cotés dont le ymin est égal à y.
   2.2 enlever de la table active les entrées dont le ymax est égal à y.
   2.3 trier la table active suivant les xi croissants.
   2.4 remplir la ligne suivant la règle de parité en utilisant les xi de la table active.
   2.5 incrémenter y de 1.
   2.6 mettre à jour les xi dans la table active

2.3 Remplissage de régions

Ce type d'algorithme traite les régions définies par une frontière. À partir d'un pixel intérieur à la région, on propage récursivement la couleur de remplissage au voisinage de ce pixel jusqu'à atteindre la frontière.

2.3.1 Connexité

Les régions et les frontières peuvent être de 2 types différents :

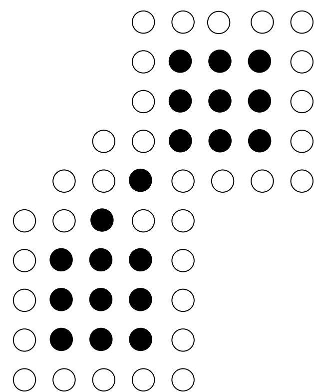
1. Régions (resp. frontières) de connexité 4 : deux pixels de la région peuvent être joints par une séquence de déplacement dans les 4 directions : haut, bas, gauche, droite.
2. Régions (resp. frontières) de connexité 8 : deux pixels de la région peuvent être joints par une séquence de déplacement dans les 8 directions : haut, bas, gauche, droite, haut-gauche, haut-droit, bas-gauche, bas-droit.

Remarques :

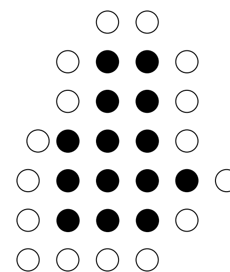
☞ La connexité 4 implique la connexité 8.

☞ Une région de connexité 8 est définie par une frontière de connexité 4.

☞ Une région de connexité 4 est définie par une frontière de connexité 8.



région 8, frontière 4



région 4, frontière 8

FIG. 12 – Connexité.

2.3.2 Algorithmes

```
void RemplissageRegion4(int x, int y        /* pixel interieur initial
                       Couleur cfrontiere   /* couleur de la frontiere
                       Couleur cregion)     /* couleur de la region */

{
  Couleur c;

  c = CouleurPixel(x,y);
  if(c != cfrontiere && c != cregion){
    AffichePixel(x,y,cregion);
    RemplissageRegion4(x,y-1,cfrontiere,cregion);
    RemplissageRegion4(x-1,y,cfrontiere,cregion);
    RemplissageRegion4(x+1,y,cfrontiere,cregion);
    RemplissageRegion4(x,y+1,cfrontiere,cregion);
  }
} /* fin  RemplissageRegion4 */
```

RemplissageRegion8 se déduit aisément de cet algorithme.

Remarque :
Algorithme hautement récursif et donc coûteux. Une amélioration significative consiste à traiter le remplissage par ligne :

```
1. calculer les pixels frontières droits pour la ligne d'ordonnée y.
2. empiler les coordonnées de ces pixels.
3. tant que la pile est non vide
 3.1. dépiler un point droit (xc,yc).
 3.2. remplir la ligne vers la gauche jusqu'au prochain point frontièr
 3.3. calculer les pixels intersections droits pour les lignes yc+1 et
 3.4. empiler les valeurs.
```

3 Antialiasing

Les algorithmes de tracé présentés précédemment ont un problème commun de précision. En effet, la discrétisation implique une perte d'informations qui se

traduit, dans notre cas, par des contours en forme de *marches d'escaliers*. Cette perte d'informations est bien connu en traitement du signal (analyse de Fourier) et le terme *aliasing* qui en est issu est utilisé en graphisme pour caractériser ces artefacts dans les images produites. L'*antialiasing* consiste alors à corriger (diminuer) ces effets.

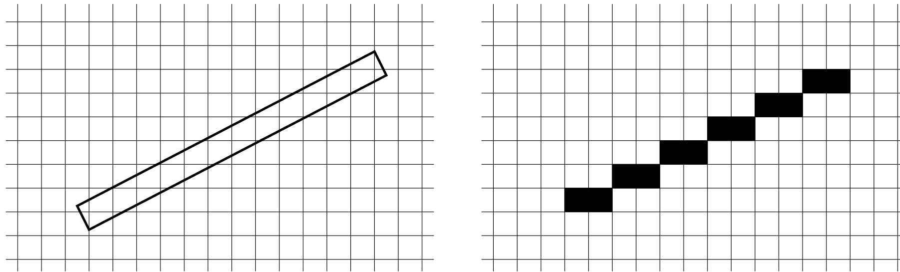


FIG. 13 – L'effet marches d'escaliers.

En dehors de la solution simple, mais rarement utilisable, qui consiste à augmenter la résolution de l'image, il existe plusieurs méthodes pour traiter l'aliasing. Nous présentons ici deux classes d'approches représentatives de ces méthodes. L'une, spécifique, consiste à améliorer les algorithmes de tracé étudiés précédemment, l'autre, plus générale, consiste à effectuer un *sur-échantillonnage* de l'image.

3.1 Algorithmes de tracé de segments corrigés

Une première solution spécifique aux algorithmes de tracé consiste à réduire l'effet de *crénelage* en allumant non pas un seul pixel à chaque itération, mais plusieurs, avec des intensités différentes suivant la proximité du pixel considéré et de la primitive. Les approches diffèrent sur les critères mis en œuvre pour déterminer l'intensité d'un pixel.

Une première approche consiste à considérer un segment d'epaisseur non nulle (1 par exemple) et d'allumer chaque pixel couvert par le segment en fonction de l'aire de recouvrement du pixel (et en considérant les pixels comme rectangulaires). La figure 14 illustre ce principe.

Cette approche est celle proposée par la librairie OpenGL. Un reproche qui lui est fait est qu'un pixel contribue sur une zone rectangulaire et non de manière isotrope. La conséquence est que des pixels voisins peuvent présenter des contrastes

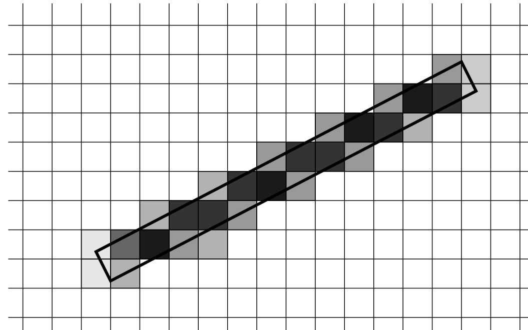


FIG. 14 – Tous les pixels intersecté par le segment ont une intensité non nulle fonction de l'aire couverte par le segment.

d'intensité importants. Une solution pour remédier à cela consiste à considérer la distance du pixel à la primitive pour calculer l'intensité.

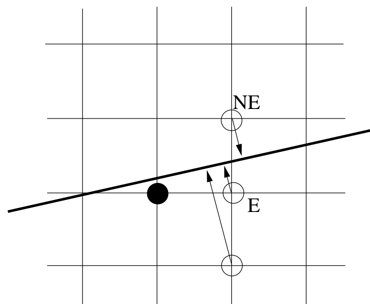


FIG. 15 – L'intensité des pixels est déterminée en fonction de leurs distances du à la primitive.

Il est facile de modifier l'algorithme de trace de segments vu precedemment de manière à prendre en compte cet aspect. La question qui se pose ensuite est comment passer d'une distance à une intensité. Classiquement, on applique une fonction de filtrage Gaussienne (voir la figure 16)

3.2 Sur-échantillonnage de l'image

Une autre approche plus générale du problème de l'aliasing consiste à calculer, dans un premier temps, une image virtuelle à une résolution plus importante que celle de l'image désirée, puis de de réduire cette image par moyennage :

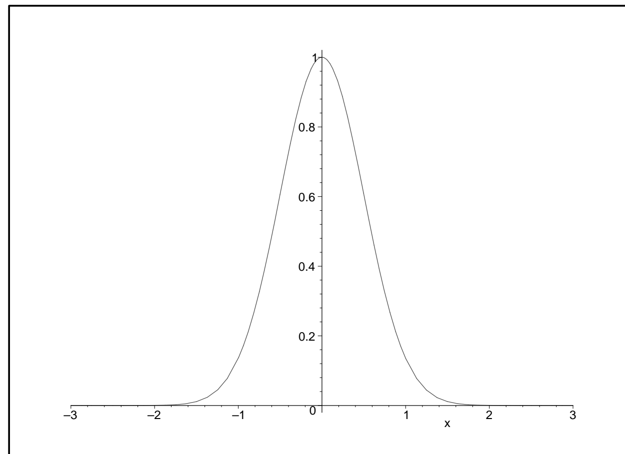


FIG. 16 – Exemple de fonction pour déterminer l'intensité des pixels suivant la distance à la primitive.

⟶ L'intensité de chaque pixel de l'image finale est déterminée par moyennage de plusieurs pixels de l'image sur-échantillonnée.

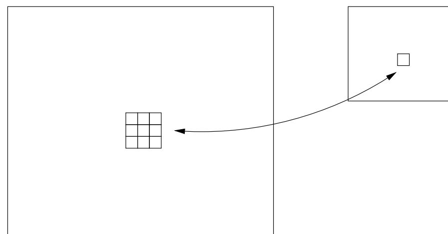


FIG. 17 – Sur-échantillonnage : l'image finale est calculée par moyennage d'une image virtuelle de résolution supèrieure.

La question est alors : quel type de moyennage appliquer ?

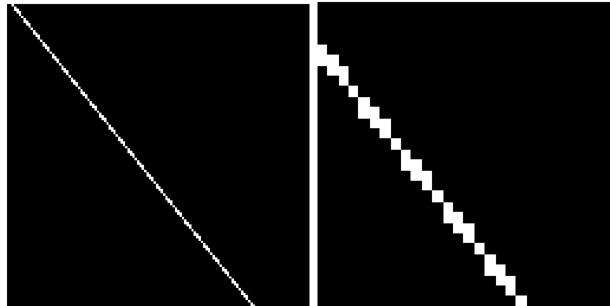


FIG. 18 – Tracé sans antialiasing.

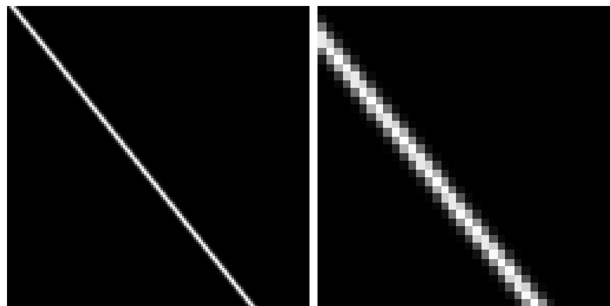


FIG. 19 – Tracé avec antialiasing (méthode spécifique basée sur les distances des pixels au segment.

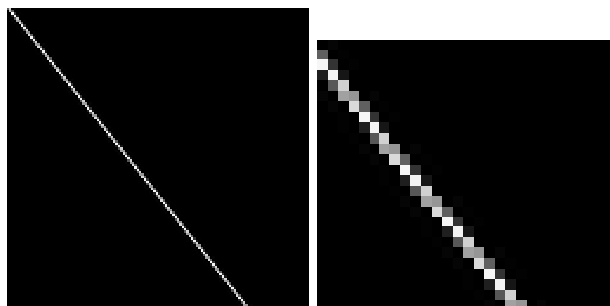


FIG. 20 – Tracé avec antialiasing (méthode générale basée le moyennage (Gaussien) d'une image de plus grande résolution.