Algorithmique de base de l'Infographie

RETOUR

Dessin 2D des objets de base

Algorithmes de tracé de segments

Algorithme de base pour beaucoup de traitements de l'Informatique Graphique:

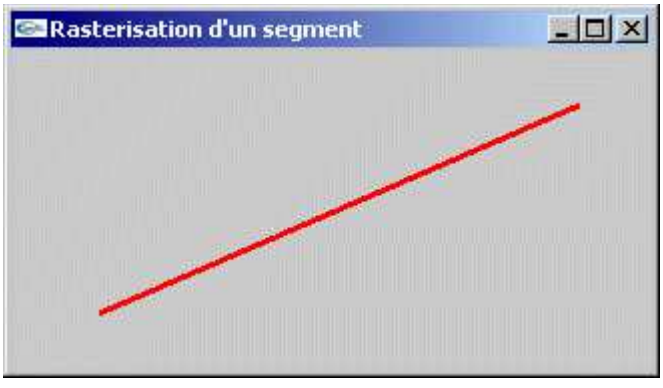
- Dessin en fil de fer
- Remplissage
- Elimination des parties cachées
- ...

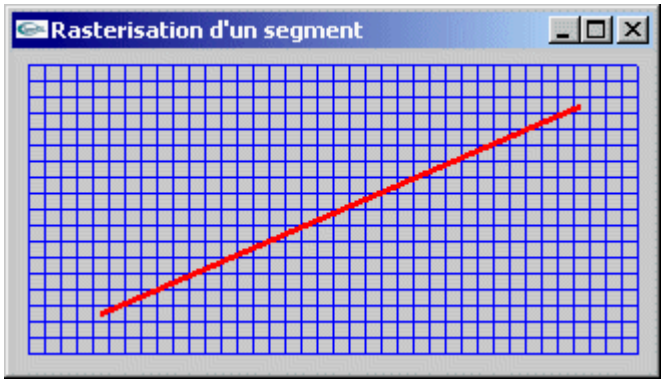
Cet algorithme doit être efficace car "mis à toutes les sauces".

On désire tracer un segment entre deux points (xi,yi) et (xf,yf) de R^2.

Ce tracé est effectué sur un écran bitmap (composé d'une matrice de n x m pixels carrés).

-> Le segment doit être discrétisé (rasterisé).





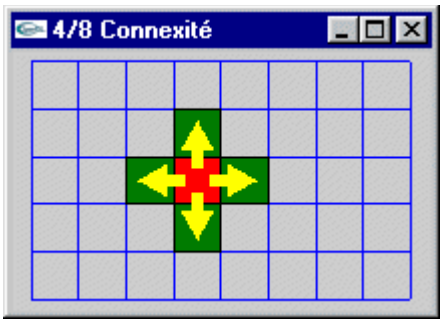
Le programme GLUt

Exécutable GLUt

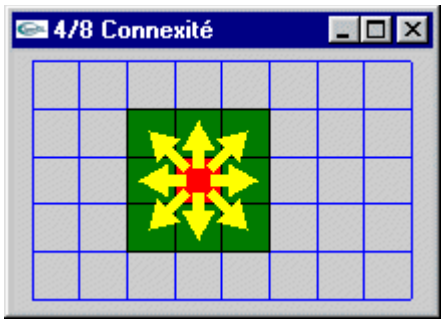
Problème: Quels pixels doit-on tracer?

Trois impératifs:

- Tout pixel du segment discret doit être traversé par le segment continu.
- Tout pixel du segment discret doit toucher au moins un autre pixel soit par l'un de ses cotés, soit par un de ses sommets (8-connexité).



4-connexité



8-Connexité

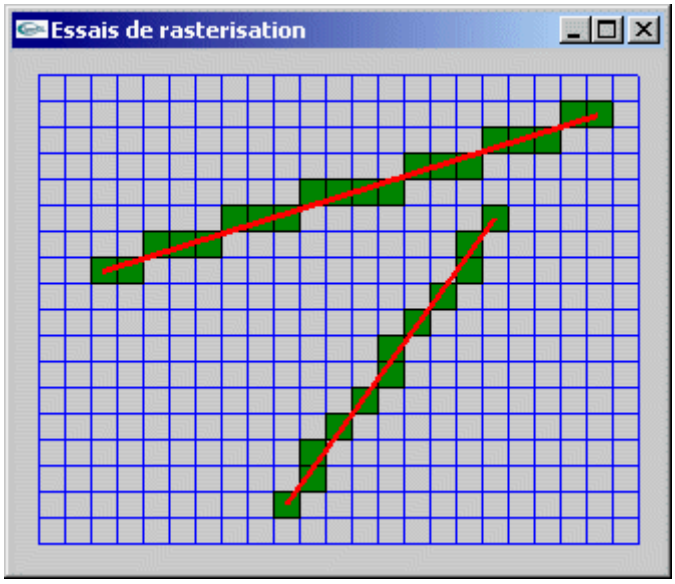
Le programme GLUt

Exécutable GLUt

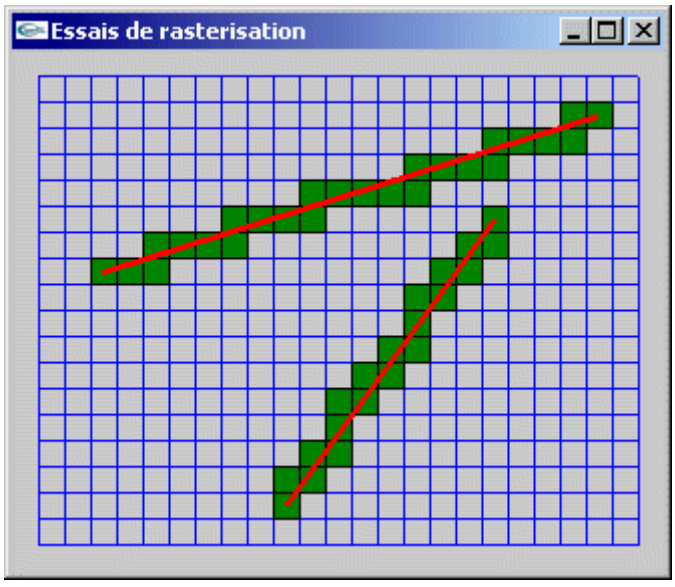
- On doit tracer le moins de pixels possible.

On trace 1+max(abs(xi-xf),abs(yi-yf)) pixels.

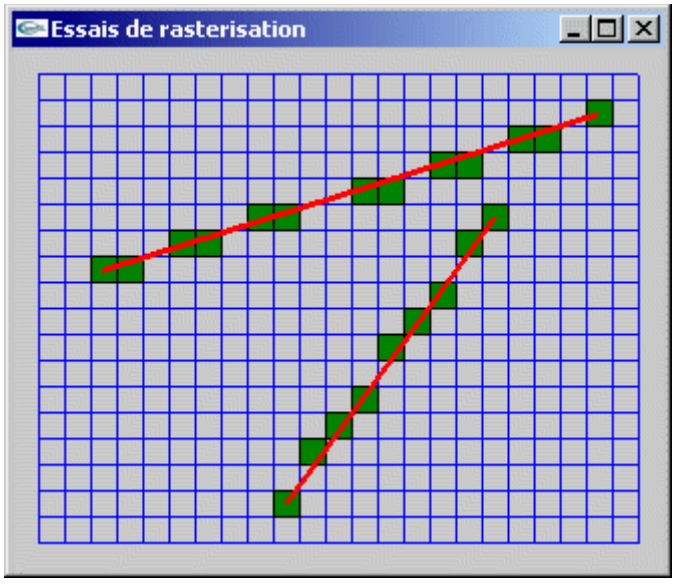
Si abs(xi-xf) < abs(yi-yf), on trace un pixel et un seul par ligne interceptant le segment, sinon on trace un pixel et un seul par colonne interceptant le segment.



Bons tracés



Mauvais tracés



Mauvais tracés

Le programme GLUt
Exécutable GLUt

Tracé de segments par l'équation cartésienne

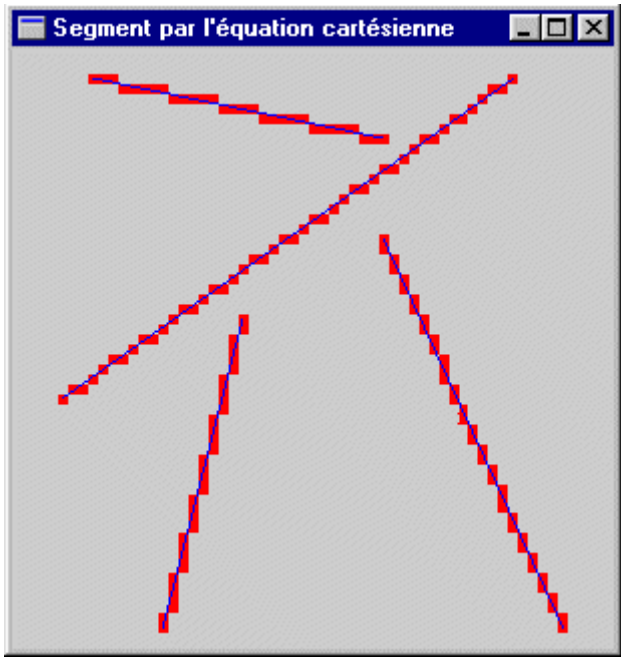
On pose comme hypothèse simplificatrice: xf > xi, yf > yi et (xf-xi) >= (yf-yi). Tous les autres cas peuvent s'y rapporter.

-> Le coefficient directeur de la droite passant par les deux sommets est positif et inférieur ou égal à 1.

On utilise l'équation cartésienne de la droite:

$$y = a\,x + b \text{ avec } a = \frac{yf - yi}{xf - xi} \text{ et } b = yi - a\,xi$$

```
void ligne(int xi,int yi,int xf,int yf) {
  int x,y ;
  double a,b ;
  a =(double) (yf-yi)/(xf-xi) ;
  b = yi - a * xi ;
  for ( x = xi ; x <= xf ; x++ ) {
    y =(int) (a * x + b) ;
    allume_pixel(x,y) ; }
}
```



Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Caractéristiques:

- Simplicité algorithmique
- Lenteur due à l'utilisation de réels, d'une division, de multiplications et de casts réel vers entier

Bresenham pour le tracé de segments

Algorithme incrémental en x et en y

On pose de nouveau les hypothèses: xf > xi, yf >= yi, xf - xi > yf - yi.

-> On allume un pixel en chaque abscisse entière x de xi à xf.

-> On utilise une variable pour stocker une estimation de la différence (positive ou négative) entre l'ordonnée entière (utilisée pour l'affichage raster) et l'ordonnée réelle calculable sur le segment continu.

```
void ligne(int xi,int yi,int xf,int yf) {
  int dx,dy,cumul,x,y ;
  x = xi ;
  y = yi ;
  dx = xf - xi ;
  dy = yf - yi ;
  allume_pixel(x,y) ;
  cumul = dx / 2 ;
  for ( x = xi+1 ; x <= xf ; x++ ) {
    cumul += dy ;
    if ( cumul >= dx ) {
      cumul -= dx ;
      y += 1 ; }
    allume_pixel(x,y) ; }
}
```

Le dernier pixel allumé a pour coordonnées (xf,yf) car dx/2+dy*dx est la valeur totale de ce qui a été accumulé dans cumul au cours de l'exécution.

-> $\frac{dx/2 + dydx}{dx}$ = dy est le nombre de fois où y a été incrémenté de 1.

-> L'ordonnée du dernier pixel allumé est yi + dy = yf.

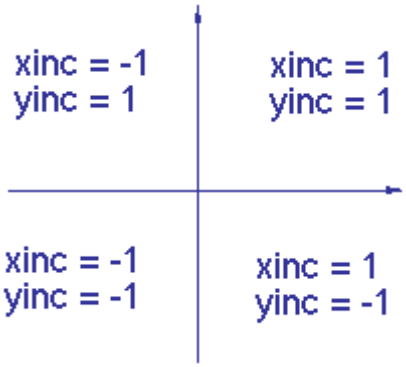
Caractéristiques

Plus grande complexité algorithmique.

Rapidité due à l'utilisation exclusive d'entiers courts (valeurs maximales de l'ordre de la résolution de l'écran -> petites valeurs) et d'opérateurs arithmétiques simples sur ces entiers (additions, soustractions, décalages binaires et comparaisons).

Algorithme adapté pour le tracé de tout segment

Utilisation de deux variables xinc et yinc pour gérer des incréments de 1 ou -1 pour x variant de xi à xf et y variant de yi à yf.



Deux parties alternatives dans l'algorithme:

- une pour incrémenter en x si segment "plutôt horizontal",
- l'autre pour l'incrémenter en y si segment "plutôt vertical",

en fonction de la pente du segment.

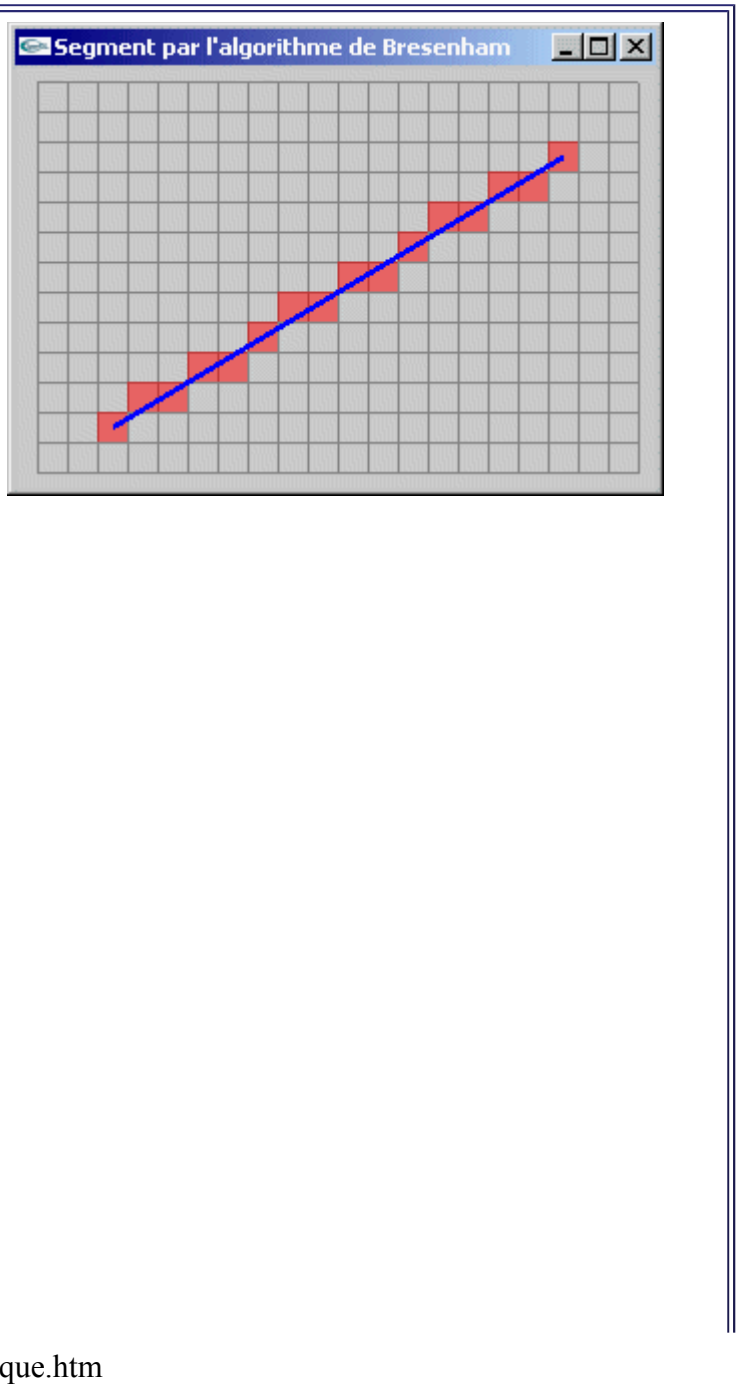
```
void ligne(int xi,int yi,int xf,int yf) {
  int dx,dy,i,xinc,yinc,cumul,x,y ;
  x = xi ;
  y = yi ;
  dx = xf - xi ;
  dy = yf - yi ;
  xinc = ( dx > 0 ) ? 1 : -1 ;
  yinc = ( dy > 0 ) ? 1 : -1 ;
  dx = abs(dx) ;
  dy = abs(dy) ;
  allume_pixel(x,y) ;
  if ( dx > dy ) {
    cumul = dx / 2 ;
    for ( i = 1 ; i <= dx ; i++ ) {
      x += xinc ;
      cumul += dy ;
      if ( cumul >= dx ) {
        cumul -= dx ;
```

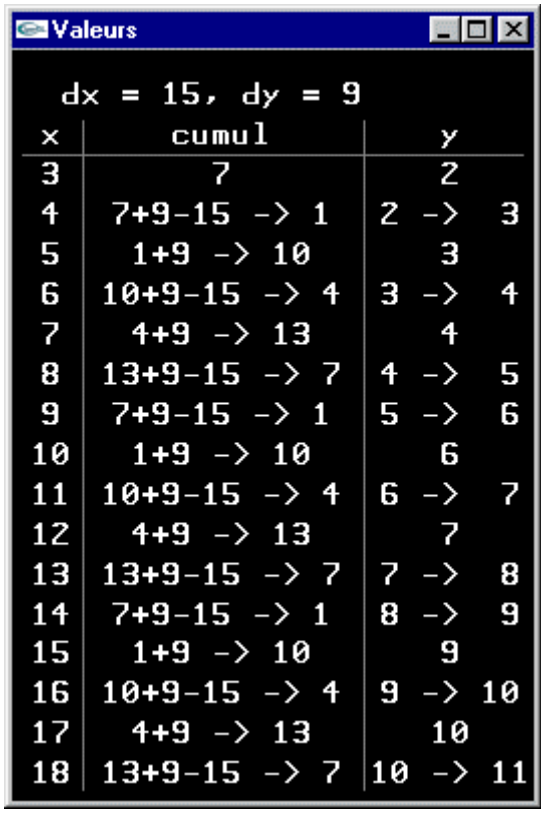
```
            y += yinc ; }
          allume_pixel(x,y) ; } }
        else {
        cumul = dy / 2 ;
        for ( i = 1 ; i <= dy ; i++ ) {
          y += yinc ;
          cumul += dx ;
          if ( cumul >= dy ) {
            cumul -= dy ;
            x += xinc ; }
          allume_pixel(x,y) ; } }
}
```

Exemple d'exécution

On souhaite tracer le segment (3,2)-(18,11).

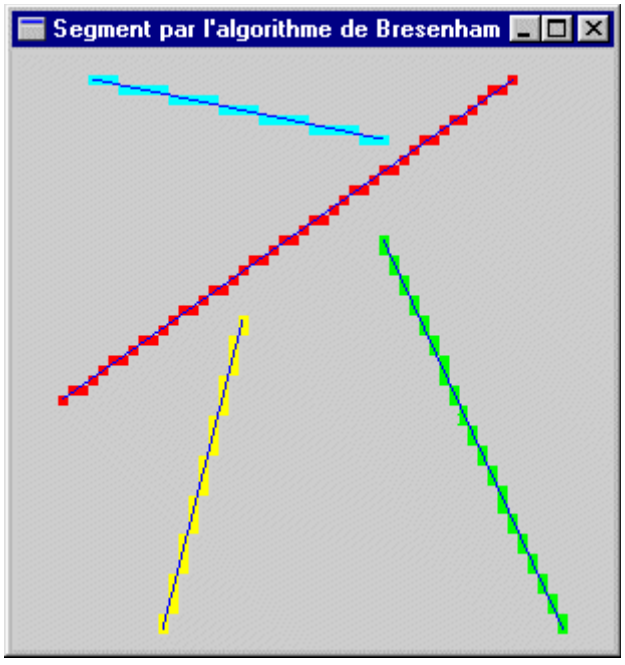
-> dx = 15, dy = 9.

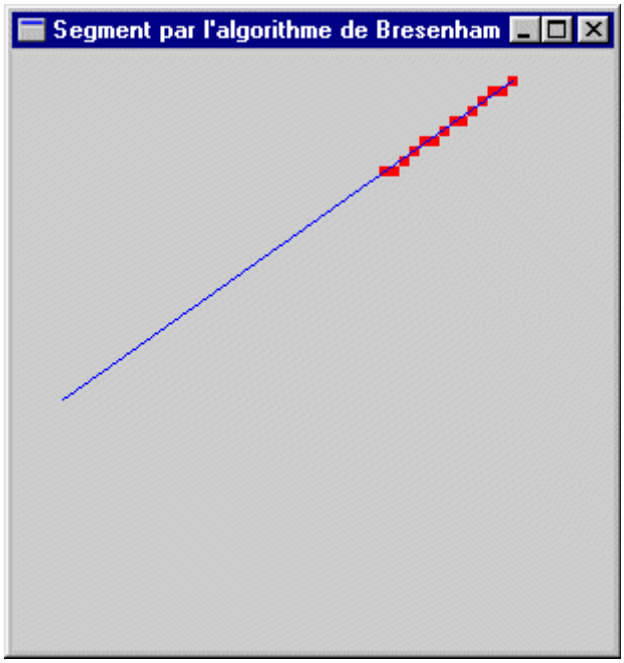


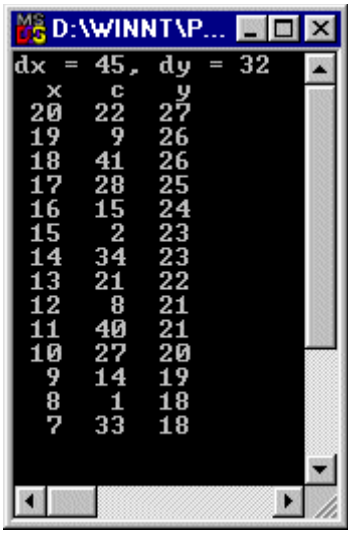


Le programme GLUt

Exécutable GLUt





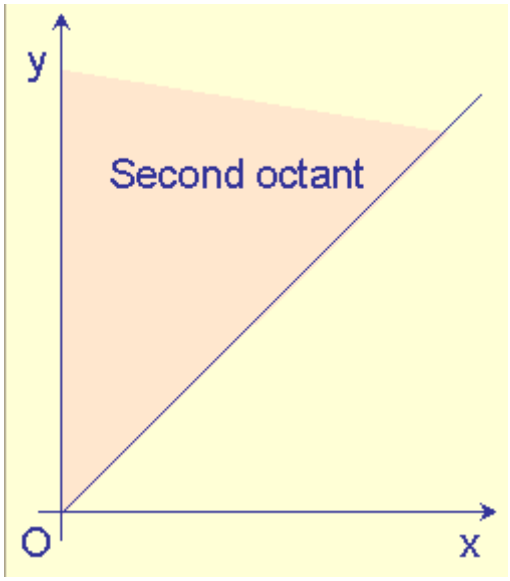


Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Bresenham pour le tracé de cercles

Principes

On désire tracer un arc de cercle de centre O et de rayon r circonscrit au deuxième octant du plan.

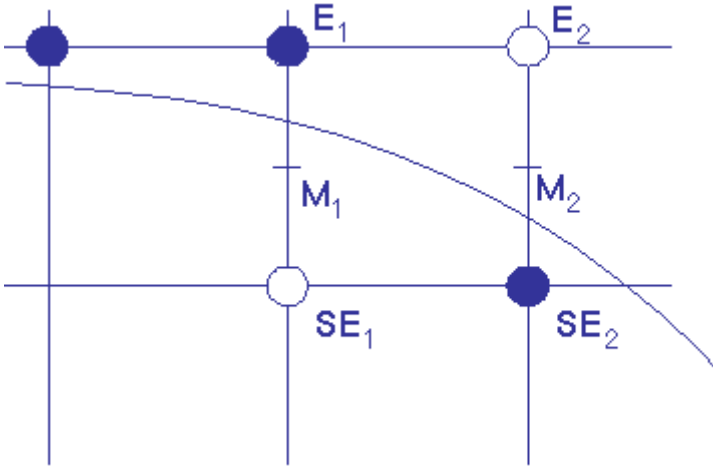


-> On trace un arc de pixels entre les points (0,r) et $(\frac{r}{\sqrt{2}},\frac{r}{\sqrt{2}})$.

Un pixel sera allumé sur chaque colonne d'abscisse comprise entre 0 et $\frac{r}{\sqrt{2}}$.

Comme dans le cadre de l'algorithme de Bresenham pour le tracé de segments, le tracé du cercle est réalisé incrémentalement:
La position d'un pixel P dépend de la position du pixel précédemment tracé. Elle est déterminée via l'évaluation de la position de P au dessus ou en dessous de l'arc réel.

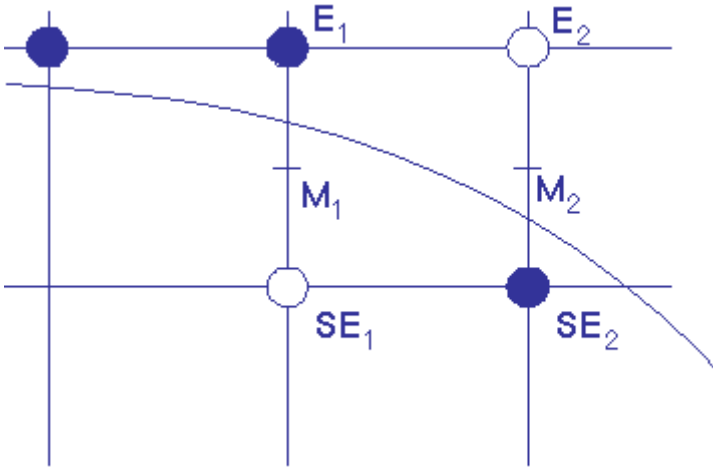
Si un pixel est allumé en position (x,y), le prochain pixel sera obligatoirement soit en position (x+1,y) soit en position (x+1,y-1) (tangente au cercle de coefficient directeur compris entre 0.0 et -1.0 sur notre portion d'arc).



L'algorithme est basé sur l'étude, pour chaque colonne de pixels, de la position (vis à vis du cercle continu: au dessus ou

en dessous) du point intermédiaire entre les deux pixels superposés candidats définis ci-dessus qui encadrent le cercle (d'où le nom alternatif d'algorithme du midpoint).

Si ce point intermédiaire est situé dans le cercle, le pixel allumé sera le pixel (x+1,y) situé en dehors du cercle (cas du pixel E_1 ci-dessous). Sinon, le pixel (x+1,y-1) situé à l'intérieur du cercle est allumé (cas du pixel E_2 ci-dessous).

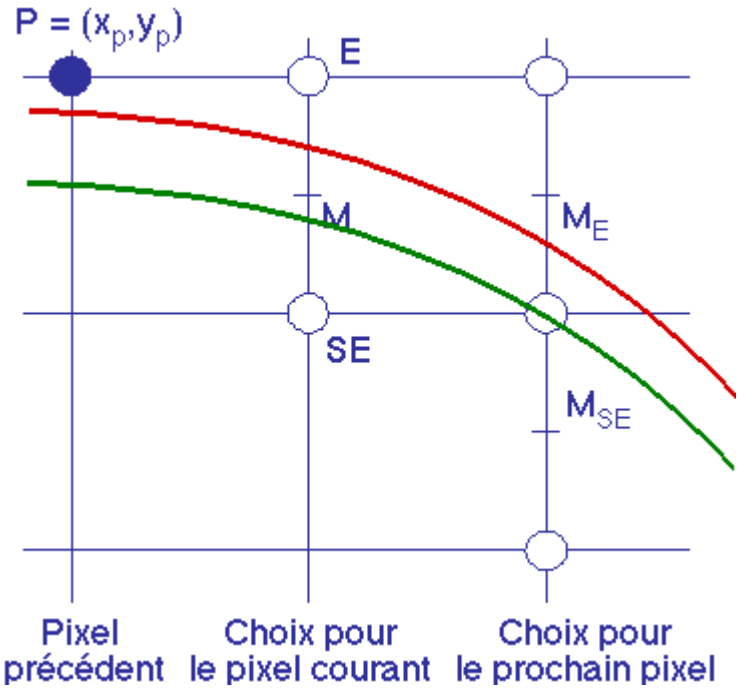


L'idée de l'algorithme est donc de définir la "position" de chaque point intermédiaire de façon incrémentale, colonne de pixels après colonne de pixels.

La position d'un point P de coordonnées (x,y) vis à vis du cercle continu est établie à partir de l'équation cartésienne f $(x,y) = x^2 + y^2 - r^2$.

-> Si $f(x,y) < 0$, P est dans le cercle, sinon il est à l'extérieur.

Première alternative : L'obtention d'un résultat négatif pour cette équation (cercle rouge ci-dessous) pour un point intermédiaire M conduit au choix du pixel extérieur E. Le prochain point intermédiaire sera situé en M_E, c'est à dire un pixel à droite de M.



Seconde alternative : Un résultat positif (cercle vert ci-dessus) conduit au choix du pixel intérieur SE, et au prochain point intermédiaire M_{SE} situé un pixel à droite et un pixel plus bas que M.

Définition récurente de l'algorithme

P = (xp,yp) est la position d'un pixel allumé.

La variable d (évaluant la position du point intermédiaire vis à vis du cercle) calculée à partir de P est:

$$d = f(xp+1,yp-\frac{1}{2}) = (xp+1)^2 + (yp-\frac{1}{2})^2 - r^2 \qquad (1)$$

(1) Si d est inférieur à 0, E est choisi. Le prochain point intermédiaire est alors M_E et la nouvelle valeur de d est calculée de la manière suivante:

$$d_{new} = f(xp+2,yp-\frac{1}{2}) = (xp+2)^2 + (yp-\frac{1}{2})^2 - r^2$$

-> $d = d_{new} = d + (2xp + 3)$ par substitution avec l'équation (1)

(2) Si d est supérieur à 0, SE est choisi. Le prochain point intermédiaire est M_{SE} et la nouvelle valeur de d est calculée selon la technique suivante:

$d_{new} = f(xp+2, yp-\frac{3}{2}) = (xp+2)^2 + (yp-\frac{3}{2})^2 - r^2$

-> $d = d_{new} = d + (2xp - 2yp + 5)$ par substitution avec l'équation (1)

-> On obtient une définition récurrente incrémentale de d n'utilisant que des entiers (coordonnées des pixels).

La première valeur de d est obtenue à partir de la position (0,r) pour un premier point intermédiaire de position placée en $(1, r-\frac{1}{2})$.

-> $d = d_{init} = 1 + r^2 - r + \frac{1}{4} - r^2 = \frac{5}{4} - r$.

-> en entier: $d = d_{init} = 1 - r$.

Sitôt que l'algorithme conduit à une valeur xp >= yp, son exécution est arrêtée car la diagonale est atteinte.

Les conditions initiales étant connues, la récurence étant définie, la condition d'arrêt étant fixée, on connaît maintenant totalement la définition récurrente de d.

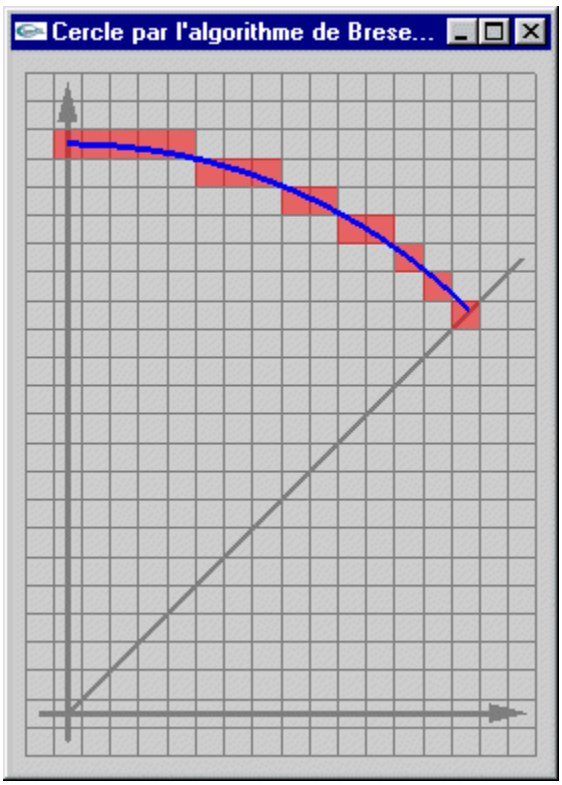
Implantation

On peut écrire une fonction de tracé d'un arc de cercle où le rayon r est un paramètre entier placé en entête.

```
void arcDeCercle(int r) {
  int x,y,d ;
  x = 0 ;
  y = r ;
  d = 1 - r ;
  allume_pixel(x,y) ;
  while ( y > x ) {
    if ( d < 0 )
      d += 2 * x + 3 ;
      else {
      d += 2 * (x - y) + 5 ;
      y-- ; }
    x++ ;
    allume_pixel(x,y) ; }
}
```

Exemple d'exécution

Arc de cercle de 20 pixels de rayon:
15 pixels tracés



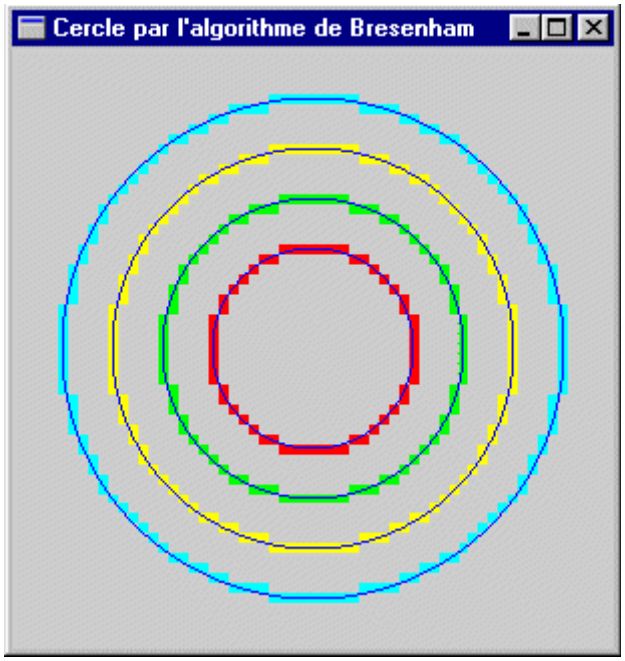
Valeurs

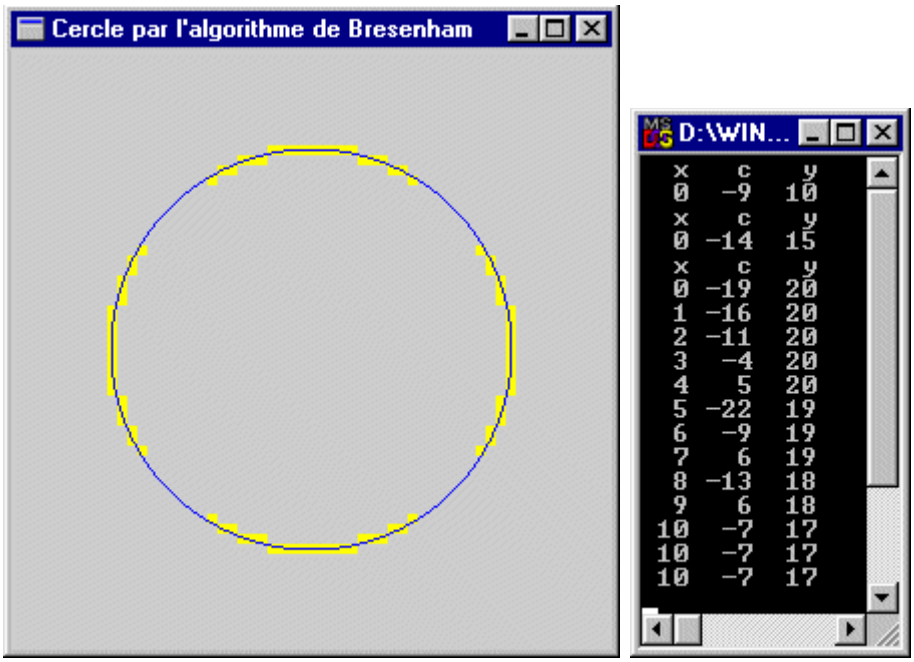
d	x	y
-19	0	20
-19+2*0+3 -> -16	1	20
-16+2*1+3 -> -11	2	20
-11+2*2+3 -> -4	3	20
-4+2*3+3 -> 5	4	20
5+2*(4-20)+5 -> -22	5	20 -> 19
-22+2*5+3 -> -9	6	19
-9+2*6+3 -> 6	7	19
6+2*(7-19)+5 -> -13	8	19 -> 18
-13+2*8+3 -> 6	9	18
6+2*(9-18)+5 -> -7	10	18 -> 17
-7+2*10+3 -> 16	11	17
16+2*(11-17)+5 -> 9	12	17 -> 16
9+2*(12-16)+5 -> 6	13	16 -> 15
6+2*(13-15)+5 -> 7	14	15 -> 14

Le programme GLUt

Exécutable GLUt

Cette fonction pourra ensuite être adaptée (au moyen de symétries) pour tracer un cercle complet.

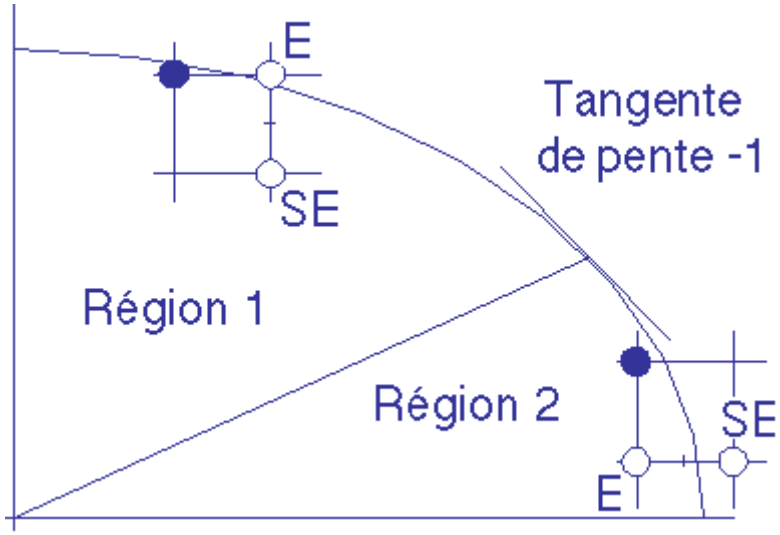




Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Le tracé d'ellipses

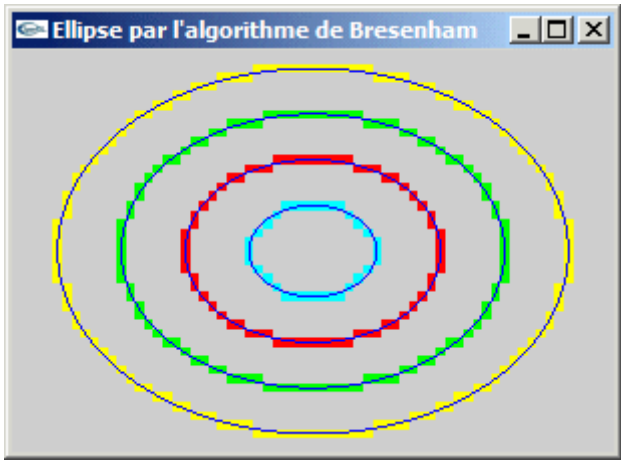
Un algorithme incrémental de tracé d'ellipses basé sur une technique du midpoint peut être défini. Le travail est plus complexe par le fait que le tracé à l'intérieur d'un quadrant n'est plus symétrique par rapport à la diagonale.

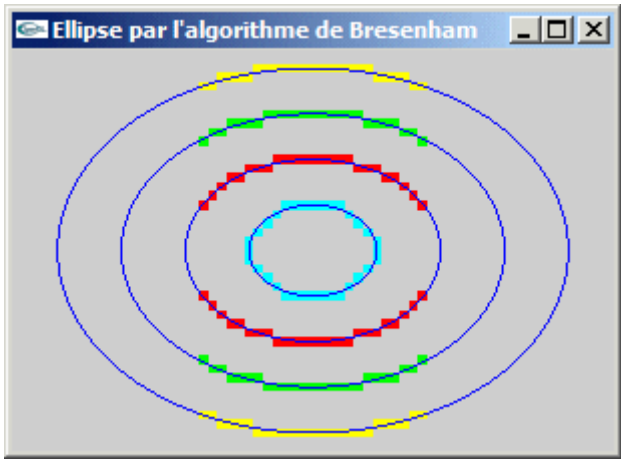


L'algorithme réalisera le tracé en deux passes consécutives pour un quadrant. Par exemple pour le quadrant 1, une passe avec un pixel par colonne pour la région 1 et une passe avec un pixel par ligne pour la région 2. La limite entre ces deux régions est le point où la pente est de -1.

```
void ellipse(long a,long b) {
  int x,y ;
  double d1,d2 ;
  x = 0 ;
  y = b ;
  d1 = b*b - a*a*b + a*a/4 ;
  allume_pixel(x,y) ;
  while ( a*a*(y-.5) > b*b*(x+1) ) {
  if ( d1 < 0 ) {
    d1 += b*b*(2*x+3) ;
    x++ ; }
    else {
    d1 += b*b*(2*x+3) + a*a*(-2*y+2) ;
    x++ ;
    y-- ; }
  allume_pixel(x,y) ; }
  d2 = b*b*(x+.5)*(x+.5) + a*a*(y-1)*(y-1) -
a*a*b*b ;
  while ( y > 0 ) {
  if ( d2 < 0 ) {
    d2 += b*b*(2*x+2) + a*a*(-2*y+3) ;
    y-- ;
    x++ ; }
    else {
    d2 += a*a*(-2*y+3) ;
    y-- ; }
  allume_pixel(x,y) ; }
}
```

Ellipse de différents grands-axes





Le programme GLUt

Exécutable GLUt

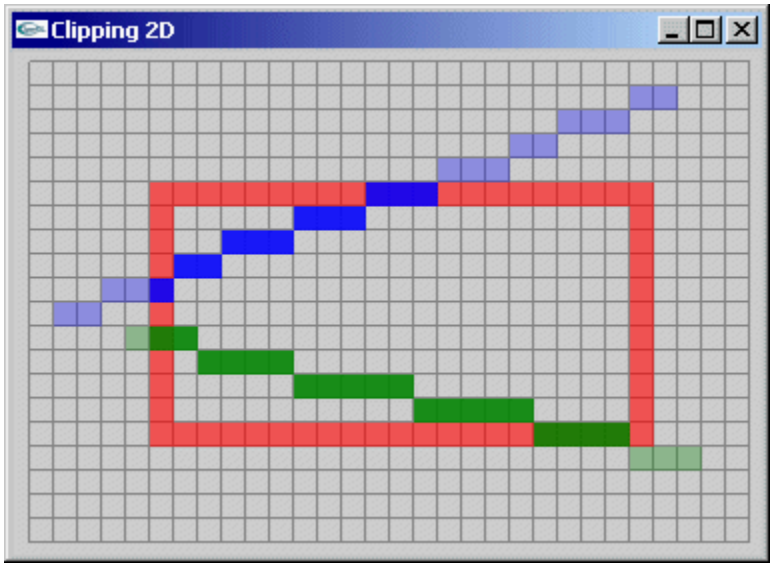
Le clipping (découpage)

Définition

Clipping écran: Traitement permettant de réduire le dessin d'un objet graphique à une région de l'écran.

Cette région est classiquement un rectangle mais peut être de toute autre forme.
Le clipping peut aussi être réalisé en 3D.

Exemple: Clipping de 2 segments à l'intérieur d'un rectangle.



Le programme GLUt

Exécutable GLUt

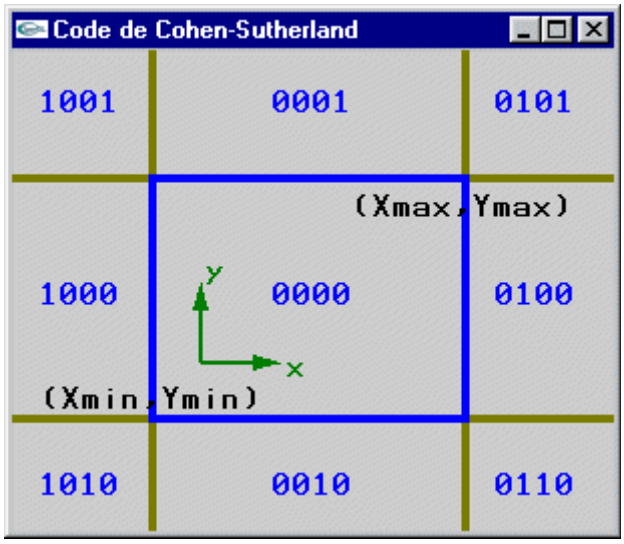
Clipping dans un rectangle à bords horizontaux et verticaux (fenêtre): Traitement de base de l'Informatique Graphique -> Nécessité d'efficacité.

Cohen-Sutherland pour le clipping d'un segment à l'intérieur d'un rectangle

Algorithme basé sur la détection d'un certain nombre de configurations de placement d'un segment relativement à un rectangle.

On associe à chacune des deux extrémités du segment un code défini sur quatre valeurs pseudo booléennes indiquant la position (x,y) du sommet par rapport aux quatre droites définissant le rectangle de clipping (xmin, xmax)-(ymin, ymax):

- Première valeur = 1 si x < xmin
- Deuxième valeur = 1 si x > xmax
- Troisième valeur = 1 si y < ymin
- Quatrième valeur = 1 si y > ymax



Le programme GLUt

Exécutable GLUt

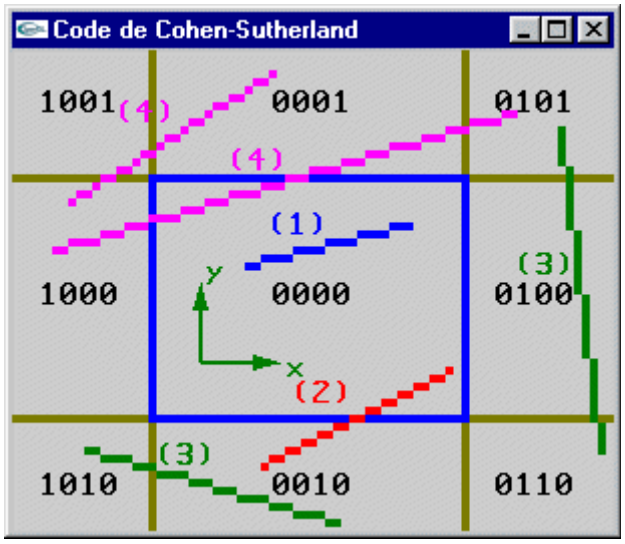
Dans trois cas on sait si un segment de coordonnées (xi,yi) -(xf,yf) passe ou ne passe pas à l'intérieur du rectangle (xmin,ymin)-(xmax,ymax):

(1) Si les codes des deux extrémités sont égaux à 0000 (cas (1) ci-desous), le segment est entièrement dans le rectangle de clipping.

(2) Si un seul des codes des deux extrémités est égal à 0000, le segment est pour partie dans le rectangle de clipping (cas (2)).

(3) Si aucun des codes des extrémités n'est égal à 0000, et que le "et logique" entre les deux codes est différent de 0000 (un 1 apparaît en position 1, 2, 3 ou 4), le segment est obligatoirement situé entièrement à l'extérieur du rectangle (cas (3)).

Dans tous les autres cas, on ne peut pas conclure avec certitude (cas (4)).



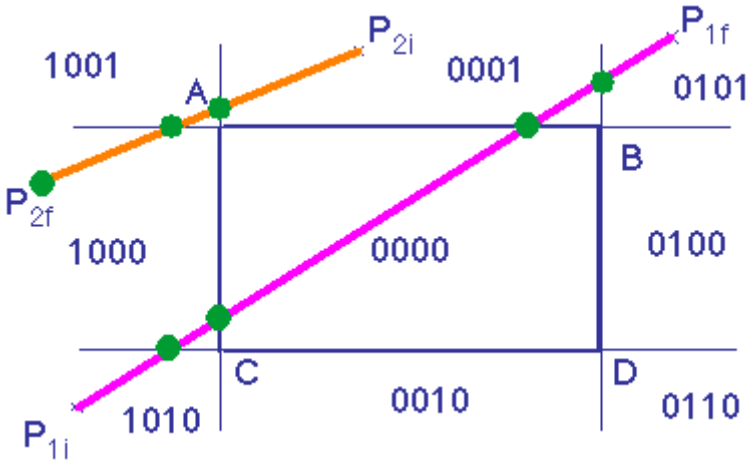
Le programme GLUt

Exécutable GLUt

Principe de l'algorithme

A partir du segment s initial, on déplace les extrémités de s vers les droites AB, BD, AC ou CD définissant le rectangle de clipping ABDC jusqu'à établir que s ainsi modifié n'appartient pas à ABDC (cas (3)) ou que s ainsi modifié appartient entièrement à ABDC (cas (1)).

Au cours de l'exécution, chaque fois qu'il est établi qu'une des valeurs du code d'une extrémité n'est pas égale à 0000, on déplace cette extrémité pour qu'elle vienne rejoindre l'une des droites AB, BD, AC ou CD. Ainsi, on en change le code. Le choix de la droite cible est réalisé en choisissant la premiére valeur du premier code d'extrémité qui n'est pas à 0.

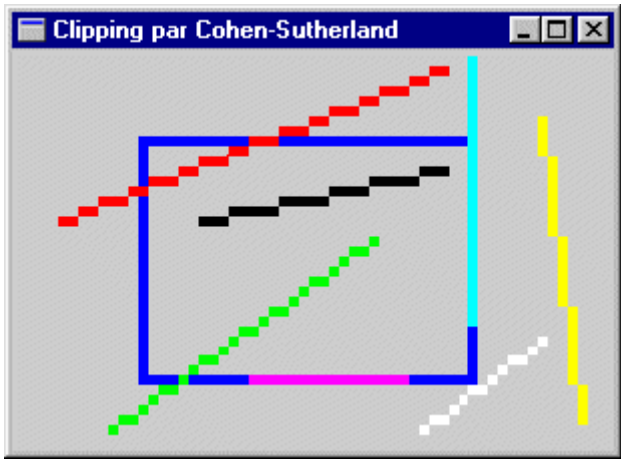


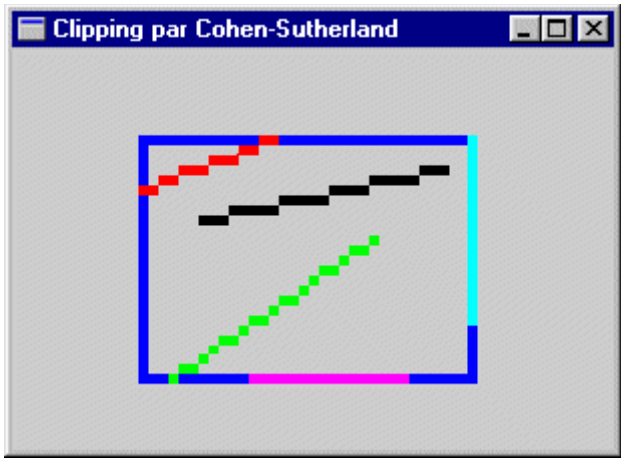
Ces modifications sont réalisées itérativement jusqu'à pouvoir conclure.

Cet algorithme de clipping peut facilement être adapté à R^3 pour le clipping de segments de droites à l'intérieur d'un parallélépipède rectangle.

Algorithme

```
procedure clip(Point2D p1,Point2D p2,fenetre f)
{
code c1,c2;
c1 = code(p1,f);
c2 = code(p2,f);
tant que (c1 ou c2 contient un 1) et
          (c1 et c2 n'ont pas de 1 commun) {
  si c1 égal 0000 {
    permuter p1 et p2;
    permuter c1 et c2; }
  si (c1 & 0001) égal 0001 {
    p1.x = abscisse du point d'ordonnee f.ymax
sur (p1,p2);
    p1.y = f.ymax; }
    sinon
    si (c1 & 0010) égal 0010 {
      p1.x = abscisse du point d'ordonnee f.ymin
sur (p1,p2);
      p1.y = f.ymin; }
      sinon
      si (c1 & 0100) égal 0100 {
        p1.y = ordonnée du point d'abscisse
f.xmax sur (p1,p2);
        p1.x = f.xmax; }
        sinon
        p1.y = ordonnée du point d'abscisse
f.xmin sur (p1,p2);
        p1.x = f.xmin; }
    c1 = code(p1,f); }
  si (c1 égal 0000) et (c2 égal 0000)
    tracer entre p1 et p2;
}
```





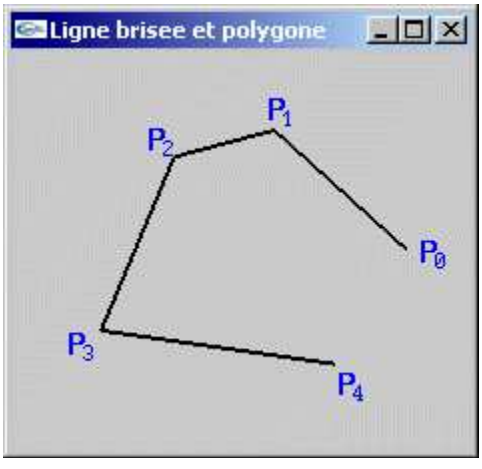
Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

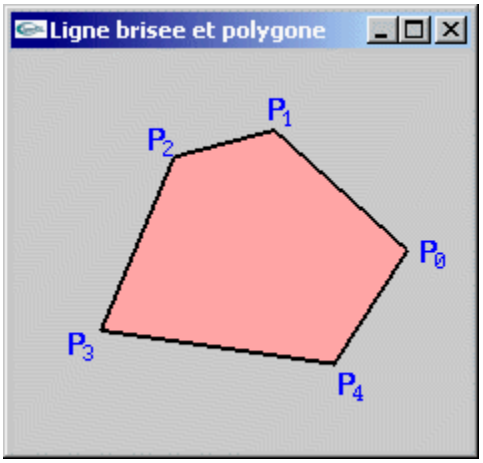
Le remplissage 2D

Définition

Ligne brisée 2D: Courbe polygonale 2D définie par une suite non circulaire de points reliés par des segments.

Polygone 2D: Surface 2D bordée par une suite circulaire de points reliés par des segments.



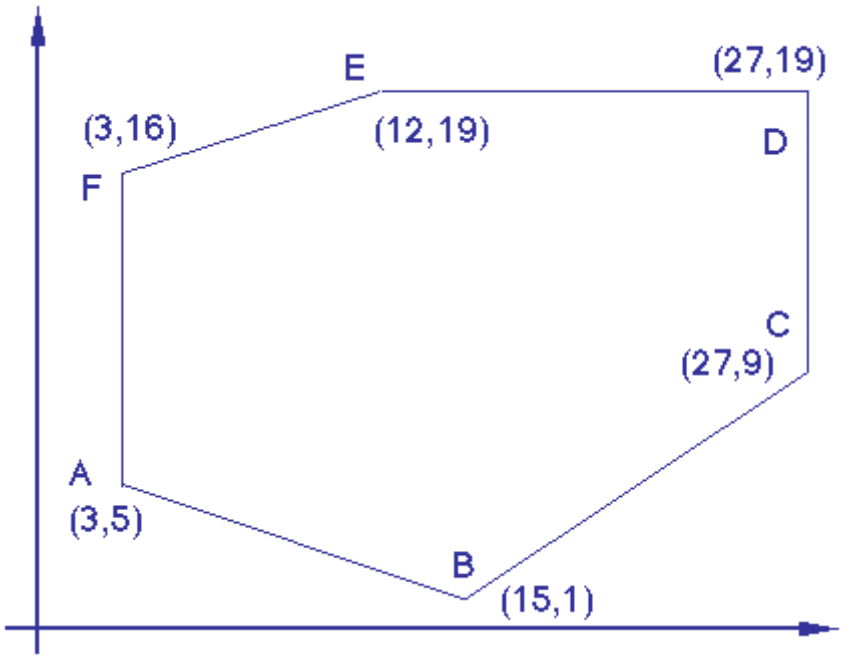


Le programme GLUt

Exécutable GLUt

Remplissage d'un polygone convexe

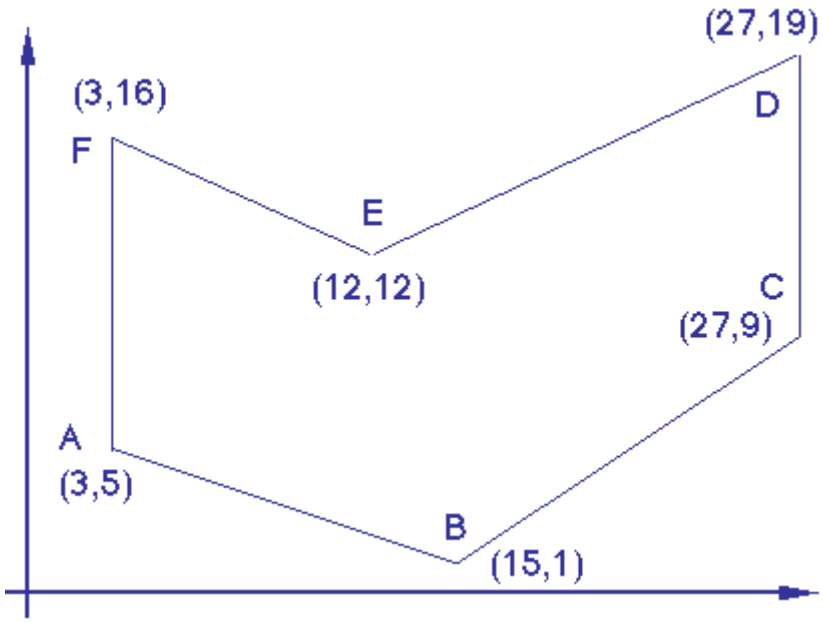
Remplissage trame après trame en commençant du sommet inférieur et en remontant les trames tout en gérant le parcours des suites de faces droite et gauche.



Suite gauche: B, A, F, E

Suite droite: B, C, D, E

Remplissage d'un polygone éventuellement concave

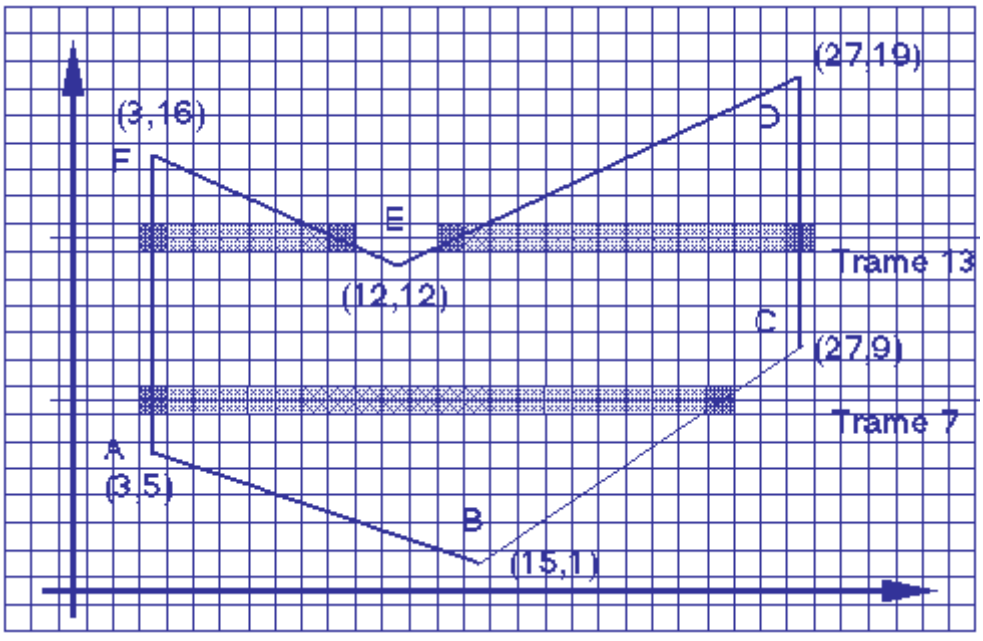


Principe

Processus de remplissage traçant les trames internes au polygone les unes à la suite des autres.

Chaque trame interceptera un nombre paire n de cotés du polygone.

Après tri en x des n intersections Pi (1 <= i <= n), on tracera des segments entre chaque couple de coordonnées (P_{2*j-1},P_{2*j}) avec 1 <= j <= $\frac{n}{2}$.



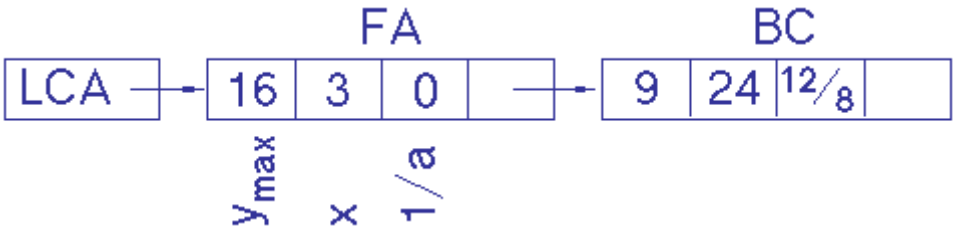
Mise en œuvre

Une liste chaînée LCA permet de gérer une suite de cotés actifs.

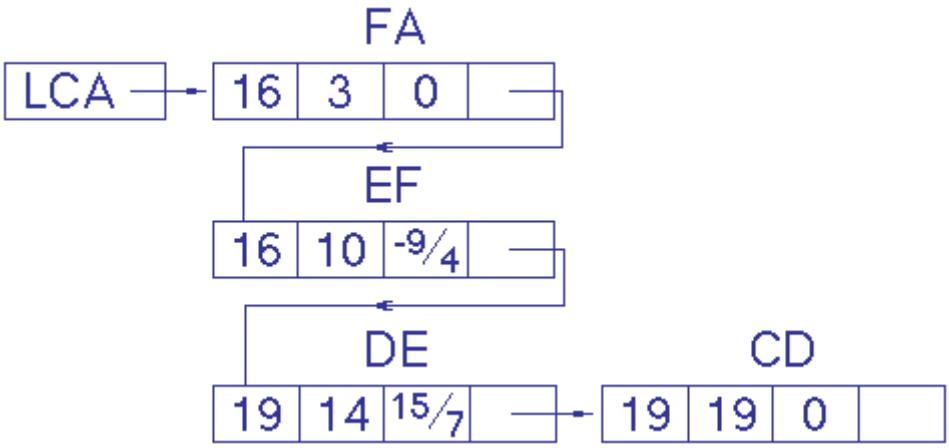
Cette liste indiquera quels sont les cotés ayant une intersection avec la trame en cours de traitement.

Elle est mise à jour à chaque avancée d'une trame au cours de l'exécution de l'algorithme.

Elle est triée en permanence par ordre croissant des abscisses x des points intersections.



LCA pour la trame 7



LCA pour la trame 13

Création d'une structure intermédiaire SI contenant les informations qui seront nécessaires à la gestion de LCA.

SI est un tableau de listes chaînées.

Pour chaque coordonnée y, on stocke l'ensemble des cotés c du polygone tels que y est l'ordonnée minimale ymin des deux extrémités de c.

(1) En parcourant SI on pourra savoir quels cotés devront entrer dans LCA pour telle ou telle ordonnée.

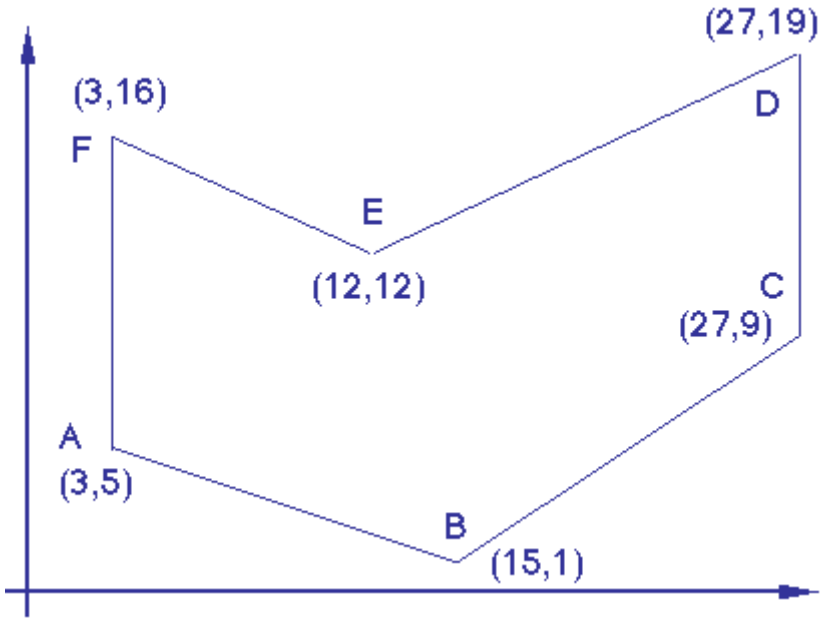
(2) Pour gérer le retrait des cotés de LCA, on mémorise l'ordonnée maximale de chaque coté.

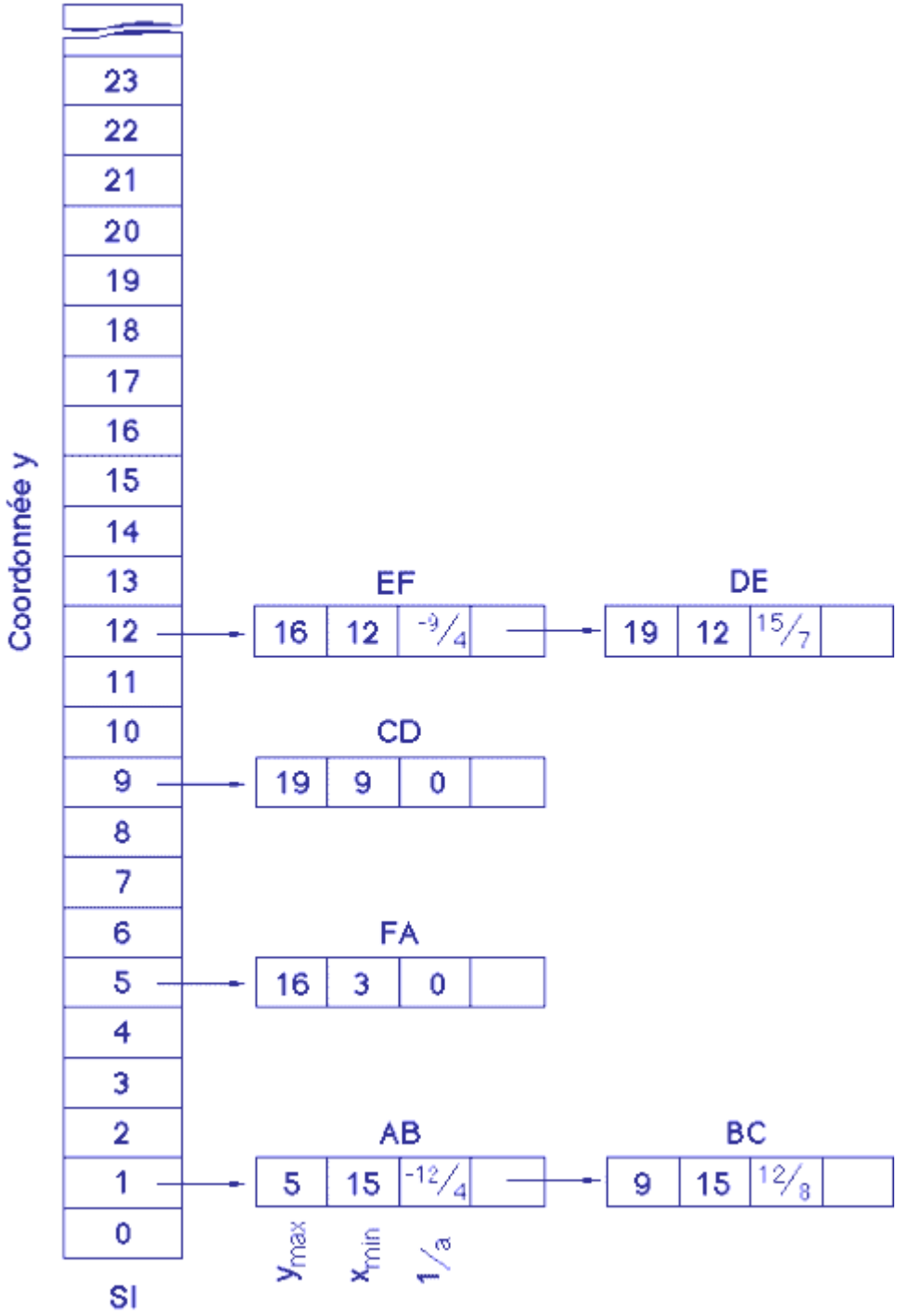
On stocke aussi pour chaque coté:

- l'inverse 1/a de son coefficient directeur (incrément en x pour un déplacement unitaire en y),

- l'abscisse xmin correspondant à ymin.

Les listes de SI sont triées par ordre croissant de xmin, puis pour deux xmin identiques, par ordre croissant de la valeur 1/a. Ce tri a pour but de faciliter le déplacement des cotés vers LCA car cet ordre de classement devra être respecté dans LCA.





Algorithme

```
Procedure remplirPolygone(Polygone p)
 début
  créer la structure SI
  initialiser la structure LCA à vide
  pour chaque trame intersectant le polygone
  début
    gérer les entrées dans LCA à partir de SI
    gérer les sorties de LCA à partir des
informations contenues dans SI
    afficher tous les morceaux de trames décrits
dans LCA
  fin
fin
```

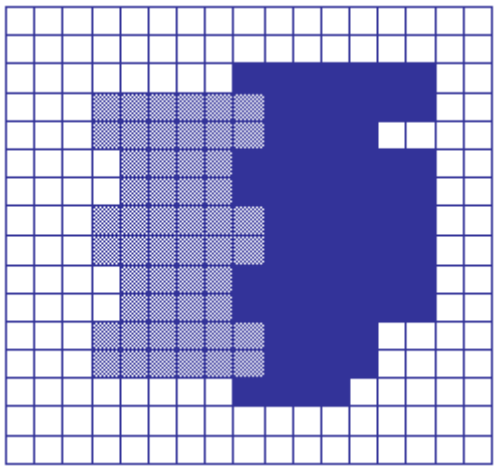
Remplissage d'une zone définie par une couleur

Définition

On entend par zone définie par une couleur:

(1) un ensemble connexe maximum de pixels de même couleur,

(2) un ensemble connexe maximum de pixels dont la couleur n'est pas une couleur définie (couleur du bord de la zone)



Cas (1): Chacune des zones grise, noire et blanche.

Cas (2): L'union des zones grise et noire vis à vis de la zone blanche.

Le remplissage est réalisé à partir d'un pixel germe.

Un premier algorithme

Principe

Au cours du remplissage, à chaque fois qu'un pixel devant être rempli est parcouru, on le remplit, puis on lance récursivement l'algorithme sur les pixels qui lui sont connexes.

Algorithme pour le remplissage d'une zone bordée

L'implantation est naturellement récursive. c est la couleur de tracé. lim est la couleur limite.

```
void remplissage(int x,int y,Couleur c,Couleur
lim) {
  Couleur cp;
```

```
cp = getCouleur(x,y) ;
if ( ( cp != lim ) && ( cp != c ) ) {
  putCouleur(x,y,c) ;
  remplissage(x,y+1,c,lim) ;
  remplissage(x,y-1,c,lim) ;
  remplissage(x+1,y,c,lim) ;
  remplissage(x-1,y,c,lim) ; }
}
```

L'utilisation de la récursivité entraîne très fréquement des problèmes de dépassement de capacité de la pile du programme si les zones remplies sont trop grandes (ce qui conduit à une profondeur de récursivité trop grande).

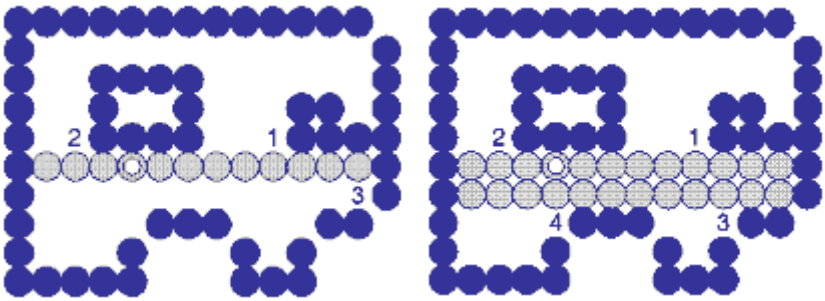
Un deuxième algorithme

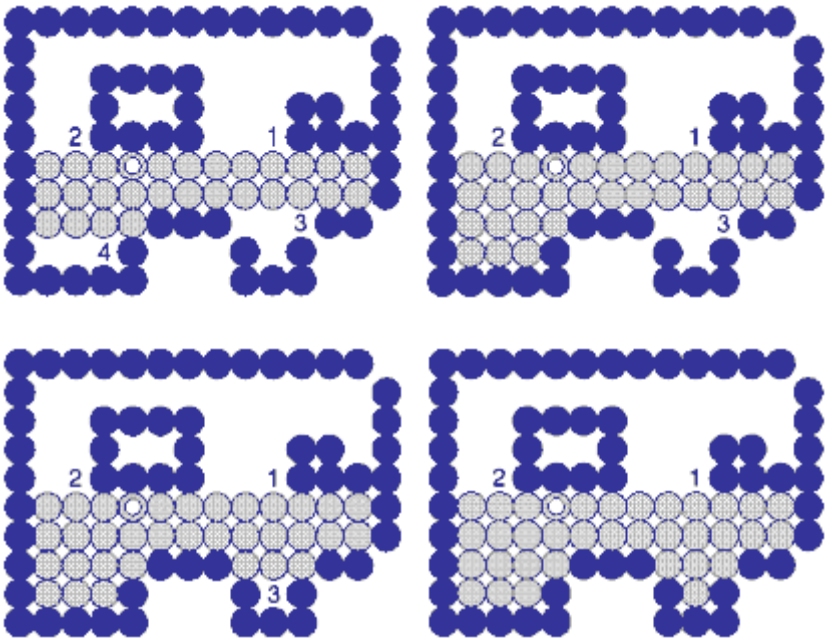
Principe

Le remplissage est effectué suite horizontale maximale de pixels après suite horizontale à partir d'un pixel germe initial.

A chaque itération:

- On remplit tous les pixels Pi jusqu'à la couleur limite à droite et à gauche du germe courant.
- On recherche parmi les pixels au dessus et en dessous des Pi ceux qui sont le plus à droite d'une suite horizontale maximale à remplir.
- Ces pixels sont empilés comme germes des itérations suivantes.





...

Algorithme pour le remplissage d'une zone bordée

L'implantation nécessite la définition d'une pile de positions de germes. L'algorithme prend la forme d'une boucle exécutée tant que la pile n'est pas vide.

```
void remplissage(int xx,int yy,int c,int lim) {
  int x,y,xi,xf ;
  p.sp = 1 ;
  p.x = calloc(1000,sizeof(int)) ;
  p.y = calloc(1000,sizeof(int)) ;
  p.x[0] = xx ;
  p.y[0] = yy ;
  setcolor(c) ;
  while ( p.sp != 0 ) {
    xi = xf = x = p.x[p.sp-1] ;
    y = p.y[p.sp-1] ;
    x++ ;
    cp = getpixel(x,y) ;
    while ( cp != lim ) {
      xf = x ;
      x++ ;
      cp = getpixel(x,y) ; }
    x = p.x[p.sp-1]-1 ;
    cp = getpixel(x,y) ;
    while ( cp != lim ) {
      xi = x ;
      x-- ;
      cp = getpixel(x,y) ; }
    line(xi,y,xf,y) ;
    p.sp-- ;
```

```
        x = xf ;
        while ( x >= xi  ) {
          cp = getpixel(x,y+1) ;
          while ( ((cp == lim) || (cp == c))
                  && (x >= xi) ){
            x-- ;
            cp = getpixel(x,y+1) ; }
          if ( (x >= xi) && (cp != lim)
               && (cp != c) ) {
            p.x[p.sp] = x ;
            p.y[p.sp] = y+1 ;
            p.sp++ ; }
          cp = getpixel(x,y+1) ;
          while ( ( cp != lim )
                  && ( x >= xi ) ) {
            x-- ;
            cp = getpixel(x,y+1) ; } }
        x = xf ;
        while ( x >= xi  ) {
          cp = getpixel(x,y-1) ;
          while ( ((cp == lim) || (cp == c))
                  && (x >= xi) ){
            x-- ;
            cp = getpixel(x,y-1) ; }
          if ( (x >= xi)
               && (cp != lim)
               && (cp != c) ) {
            p.x[p.sp] = x ;
            p.y[p.sp] = y-1 ;
            p.sp++ ; }
          cp = getpixel(x,y-1) ;
          while ( ( cp != lim )
                  && ( x >= xi ) ) {
            x-- ;
            cp = getpixel(x,y-1) ; } } }
      free(p.x) ;
      free(p.y) ;
    }
```