Inf 431 – Cours 16

Algorithmes géométriques

jeanjacqueslevy.net

secrétariat de l'enseignement:
Catherine Bensoussan
cb@lix.polytechnique.fr
Aile 00, LIX,
01 69 33 34 67

www.enseignement.polytechnique.fr/informatique/IF

Plan

1. Graphique *bitmap*
2. Enveloppe convexe
3. Recherche de points dans des intervalles
4. Intersection de segments orthogonaux
5. Intersection de segments

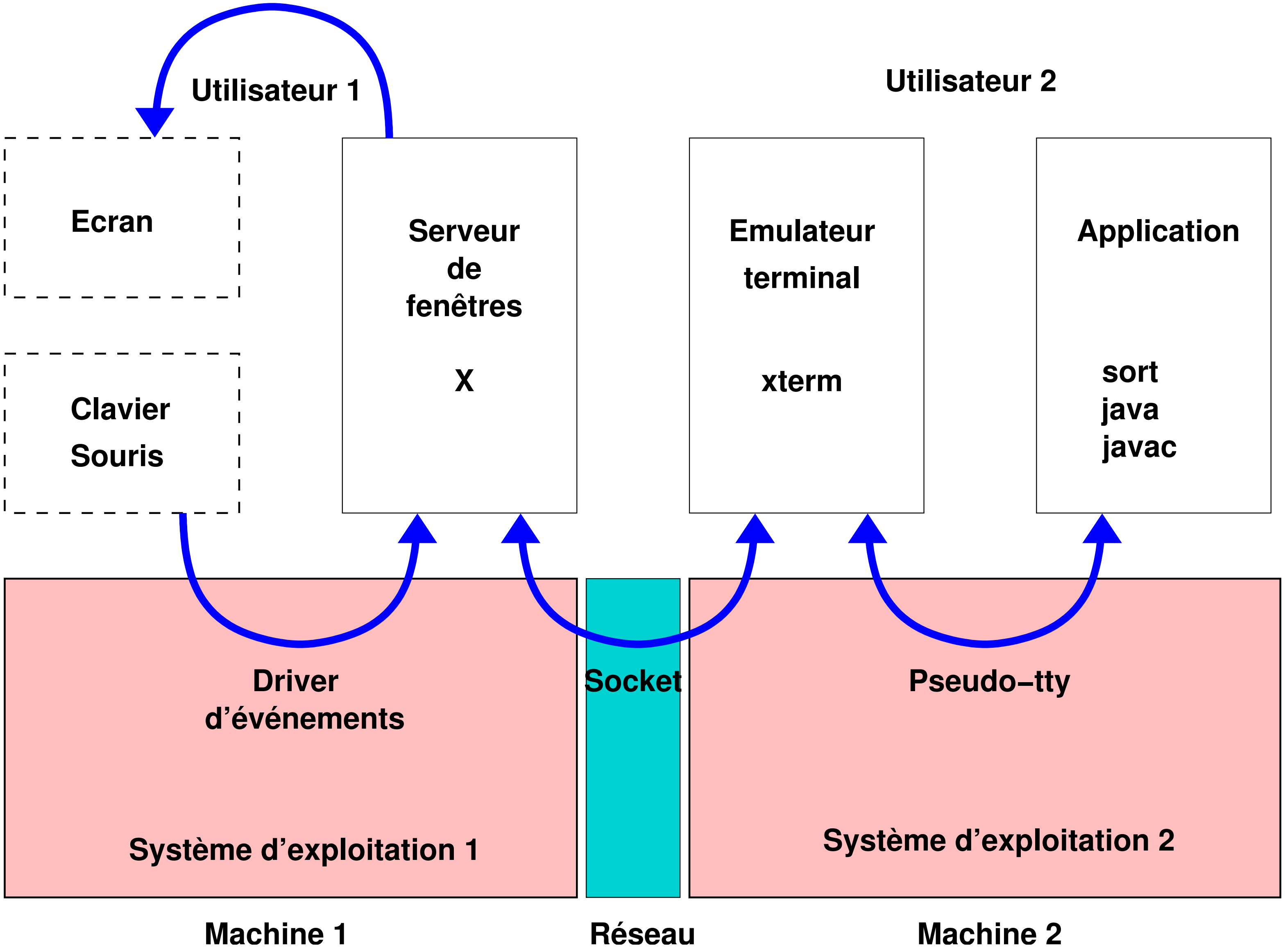
Bibliographie

J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hugues, *Computer Graphics, Principle and Practice*, Addison Wesley, 1990.

D.E. Knuth, *The Metafont book*, Addison Wesley, 1986.

D.E. Knuth, *Metafont : The Program*, Addison Wesley, 1986.

Architecture de X-window

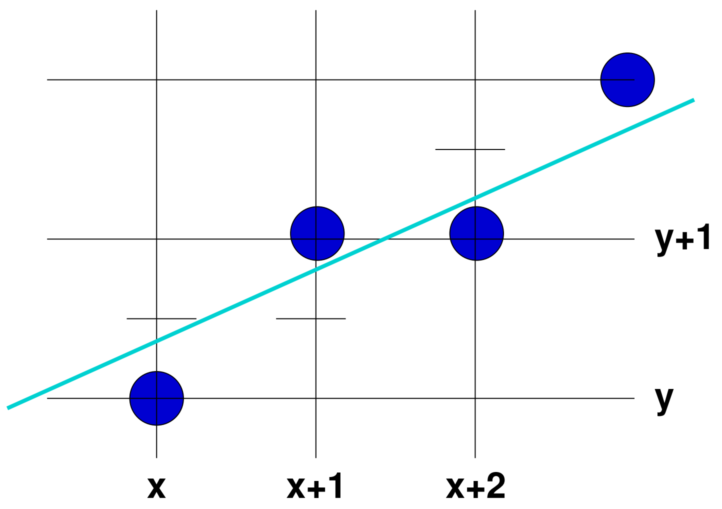


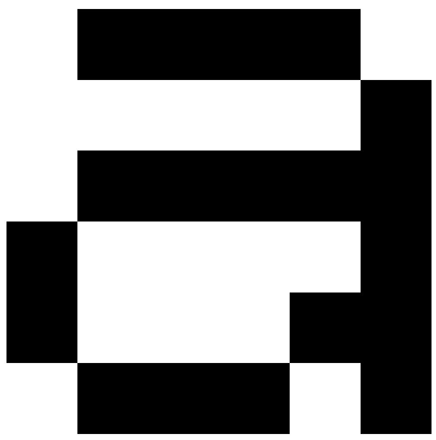
cf. Cours systèmes d'exploitation et réseaux en majeure 2

Graphique *bitmap*

- Ecran = matrice de *pixels* $(1280 \times 854 \times 32)$.
- Ecran = zone mémoire (*mémoire vidéo*), directement accessible par le processeur, et/ou par son co-processeur graphique.
- Tous les tracés sont digitalisés.

Par exemple pour un vecteur ou le dessin de la lettre "a"





Tracé de vecteur (1/4)

- But : tracer le vecteur $\overrightarrow{P_0 P_1}$ entre les points P_0 et P_1 de coordonnées (x_0, y_0) et (x_1, y_1) ($x_0 \in \mathbf{N}$, $y_0 \in \mathbf{N}$, $x_1 \in \mathbf{N}$, $y_1 \in \mathbf{N}$)
- La méthode la plus simple consiste à calculer la pente $m = dy/dx$ où $dy = y_1 - y_0$ et $dx = x_1 - x_0$, et en supposant $0 \leq m \leq 1$ et $0 < dx$.

```
static void vecteur (int x0, int y0, int x1, int y1) {
  int dx = x1 - x0, dy = y1 - y0;
  float m = ((float)dy) / dx;
  for (int x = x0, float y = y0; x <= x1; ++x) {
    setPixel(x, Math.round(y));
    y = y + m;
  }
}
```

- Opérations flottantes. Calcul de l'arrondi. C'est un peu long.
- Exécution

Tracé de vecteur (2/4)

- L'équation de la droite passant par (x_0, y_0) et (x_1, y_1) est

$$-xdy + ydx + x_0dy - y_0dx = 0$$

  où $dy = y_1 - y_0$ et $dx = x_1 - x_0$.

- On maintient une erreur e entre le point (x, y) et la droite.

$$e = -xdy + ydx + x_0dy - y_0dx$$

- On suppose $0 \leq dy/dx \leq 1$. Pour savoir si le pixel suivant sur la droite $y = x + 1$ est à l'est ou au nord-est du point (x, y), on regarde le signe de l'erreur pour le point $(x + 1, y + 0.5)$

$$e_m = -(x + 1)dy + (y + 0.5)dx + x_0dy - y_0dx$$

  Soit

$$e_m = e - dy + dx/2$$

  En multipliant par 2, on obtient

$$2e_m = 2e - 2dy + dx$$

Tracé de vecteur (3/4)

Si $2e_m \leq 0$, on positionne le pixel nord-est et $2e \leftarrow 2e_m - dx$.
Sinon on positionne le pixel est ; l'erreur devient $2e \leftarrow 2e_m + dx$.

D'où le programme quand la pente est positive et inférieure à 1.

```
static void vecteur (int x0, int y0, int x1, int y1) {
  int dx = x1 - x0, dy = y1 - y0;
  int e = 0;
  for (int x = x0, y = y0; x <= x1; ++x) {
    setPixel(x,y);
    int em = e - 2*dy + dx;
    if (em <= 0) {
      ++y;
      e = em + dx;
    } else
      e = em - dx;
  }
}
```

Si x_0, y_0, x_1, y_1 sont entiers, toutes les opérations sont des additions entières de constantes. [Bresenham]

Tracé de vecteur (4/4)

Exercice 1 Compléter le programme pour qu'il trace un vecteur de pente arbitraire.

Exercice 2 Trouver un algorithme genre Bresenham pour les tracés de cercles, ou d'ellipses.

Si on dessine le vecteur dans une fenêtre, on peut avoir à l'intersecter avec un rectangle (*clipping*). Par exemple avec le bord gauche $x = x_{min}$ d'un rectangle. On calcule

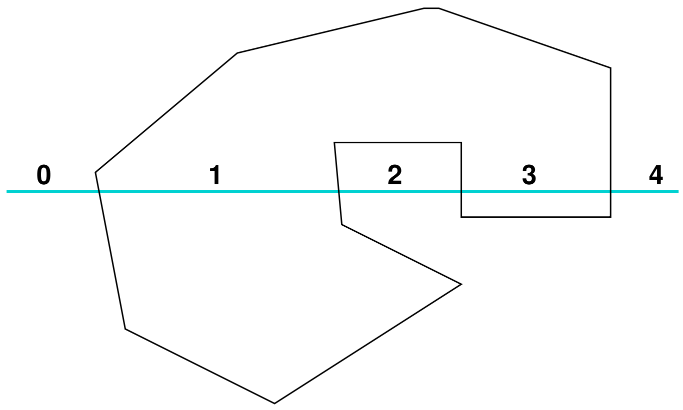
$$y = y_0 + \lfloor (x_{min} - x_0)\frac{dy}{dx} + 0.5 \rfloor$$

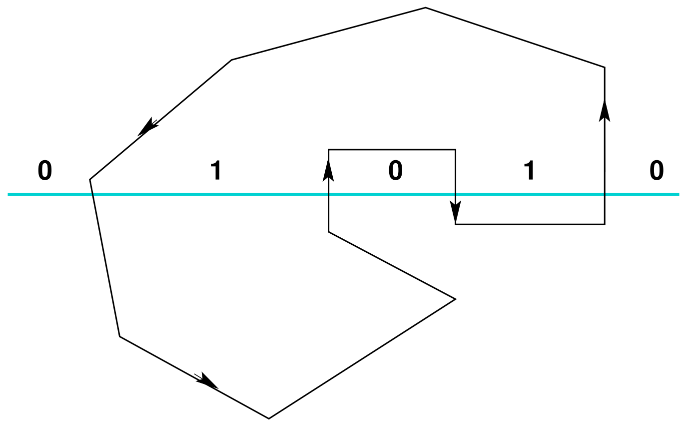
et on démarre le tracé avec une erreur non nulle

$$2e = -2x_{min}dy + 2ydx + 2x_0dy - 2y_0dx$$

Remplissage de polygones

- Deux manières pour définir l'intérieur d'un polygone.
  - pair-impair : on compte la parité des intersections d'une droite intersectant le polygone.
  - la règle de l'enroulement : les bords sont des vecteurs orientés. L'intérieur est toujours à gauche du vecteur bord.





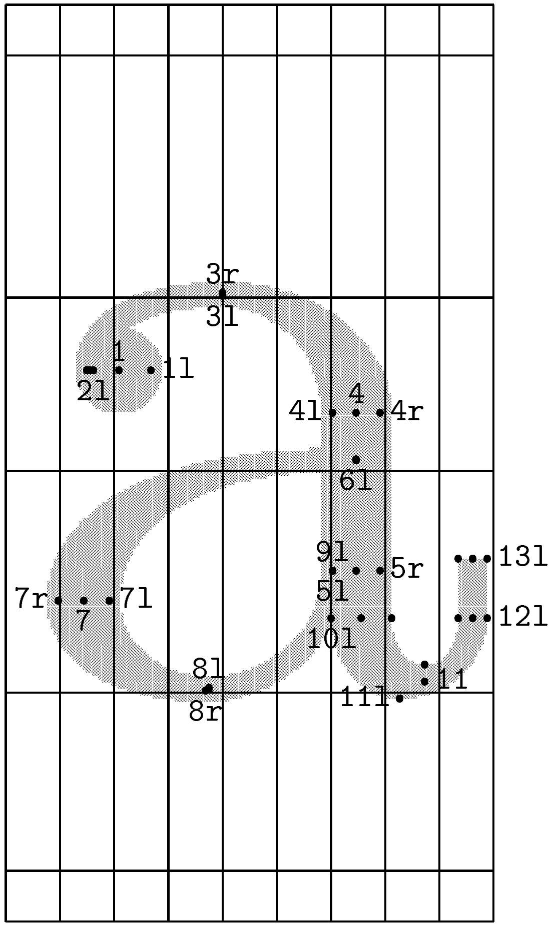
- on balaie par une ligne horizontale, et on adapte le Bresenham de vecteurs en gérant une file de priorité pour l'arrivée de nouveaux segments.

Bitblt

- Pour afficher du texte, chaque lettre est un petit rectangle de pixels qu'on recopie à l'endroit voulu sur l'écran.
- Pour le défilement du texte dans une fenêtre, on recopie un rectangle de quelques lignes vers le haut (*scrolling*).
- $\Rightarrow$ opérations rapides pour recopier des rectangles de pixels. *Bit Block Transfer* (*bitblt*) ou encore paquetage *Raster-op*, réalisées par des processeurs vidéo spécialisés (avec beaucoup de mémoire pour stocker les polices de caractères).
- Autrefois, ces opérations étaient réalisées par des processeurs normaux, avec plein d'optimisations [Pike, Locanthi, Reiser, 84].
- Les opérations *biblt* viennent du premier écran bitmap : l'Alto de Xerox PARC, [Lampson, McCreight, Thacker, 74]
- pixels pour chaque lettre, on part d'une description de chaque caractère dans une police par des cubiques. Polices de *Postscript/PDF* [Warnock] ou de *Metafont* [Knuth].

Dessin des lettres

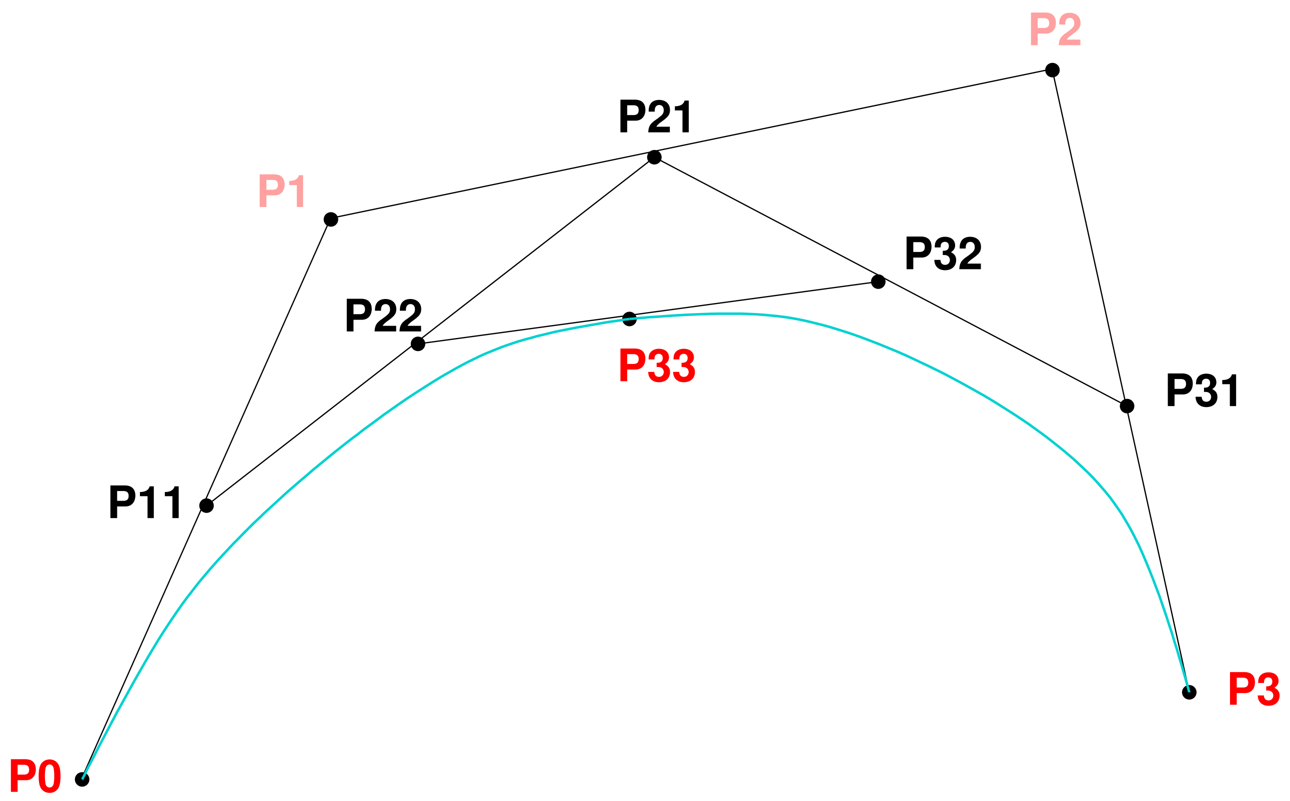
METAFONT output 2002.06.11:1450 Page 27 Character 97 “The letter a”




```
2 = 2l + (-1,0)
3 = 3r + (0,-0.5)
5 = 5r + (-7.5,0)
6 = 6l + (0,0.2)
8 = 8l + (0,-0.5)
9 = 9l + (0,0)
10 = 10l + (9.5,0)
12 = 12l + (-4.5,0)
13 = 13l + (-4.5,0)
1r = 2l + (-2,0)
2r = 2l + (-2,0)
6r = 6l + (0,0.3)
9r = 9l + (0,0)
10r = 5r + (3.5,-15.1)
11r = 11 + (0,5.4)
12r = 12l + (-9,0)
13r = 13l + (-9,0)
```

Tracé d'une cubique de Bézier

Par Bresenham ou par dichotomie :



Ordre trigonométrique

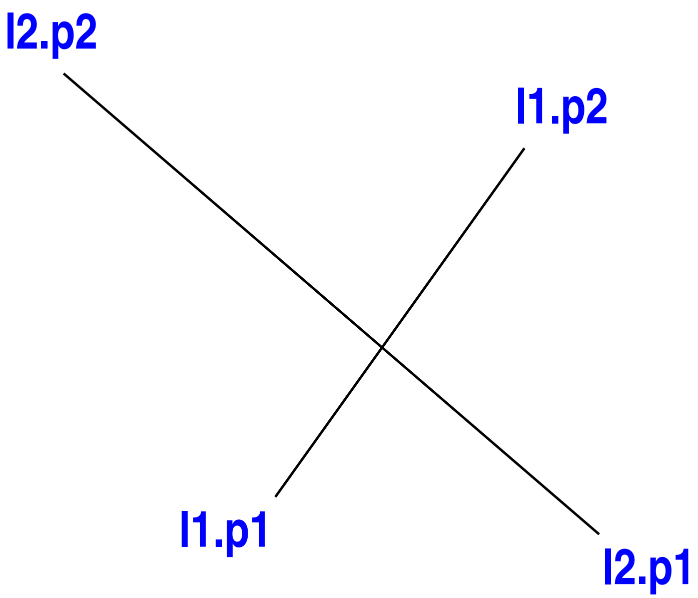
On cherche à savoir si l'angle $0 \leq \widehat{P_1 P_0 P_2} < \pi$?
Dans le cas où $\widehat{P_1 P_0 P_2} = 0$, on exige alors $P_0 P_1 < P_0 P_2$.

En calculant le produit vectoriel $\overrightarrow{P_0 P_1} \wedge \overrightarrow{P_0 P_2}$. Si l'angle est nul, par convention on compare les normes.

```
static int ordreTrigo (Point p0, Point p1, Point p2) {
  int dx1 = p1.x - p0.x; int dy1 = p1.y - p0.y;
  int dx2 = p2.x - p0.x; int dy2 = p2.y - p0.y;
  if (dx1 * dy2 > dy1 * dx2) return 1;
  else if (dx1 * dy2 < dy1 * dx2) return -1;
  else {
    if (dx1 * dx2 < 0 || dy1 * dy2 < 0) return -1;
    else if (dx1*dx1 + dy1*dy1 < dx2*dx2 + dy2*dy2) return 1;
    else if (dx1*dx1 + dy1*dy1 == dx2*dx2 + dy2*dy2) return 0;
    else return -1;
  }
}
```

Intersection de segments

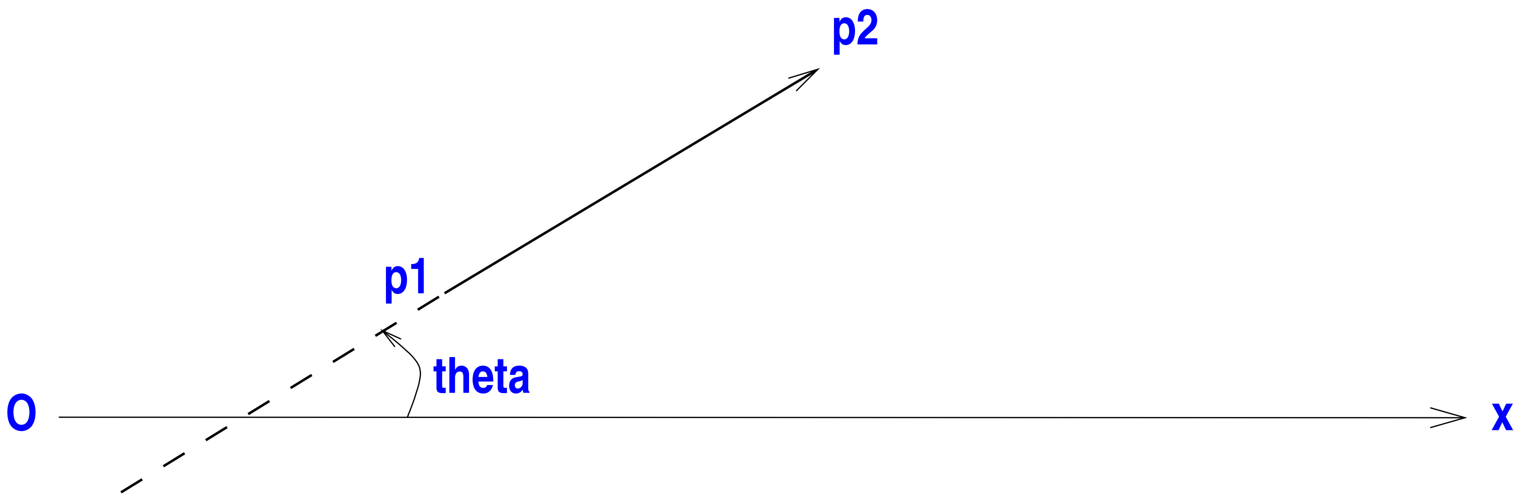
```
static boolean intersection (Line l1, Line l2) {
  return ordreTrigo (l1.p1, l1.p2, l2.p1)
        * ordreTrigo (l1.p1, l1.p2, l2.p2) <= 0
       && ordreTrigo (l2.p1, l2.p2, l1.p1)
        * ordreTrigo (l2.p1, l2.p2, l1.p2) <= 0;
}
```



Pente d'un vecteur

```
static float theta (Point p1, Point p2) {
  float t; int dx = p2.x - p1.x; int dy = p2.y - p1.y;
  if (dx == 0 && dy == 0) t = 0;
  else t = (float) dy / (Math.abs(dx) + Math.abs(dy));
  if (dx < 0) t = 2 - t;
  else if (dy < 0) t = 4 + t;
  return t * 90.f;
}
```

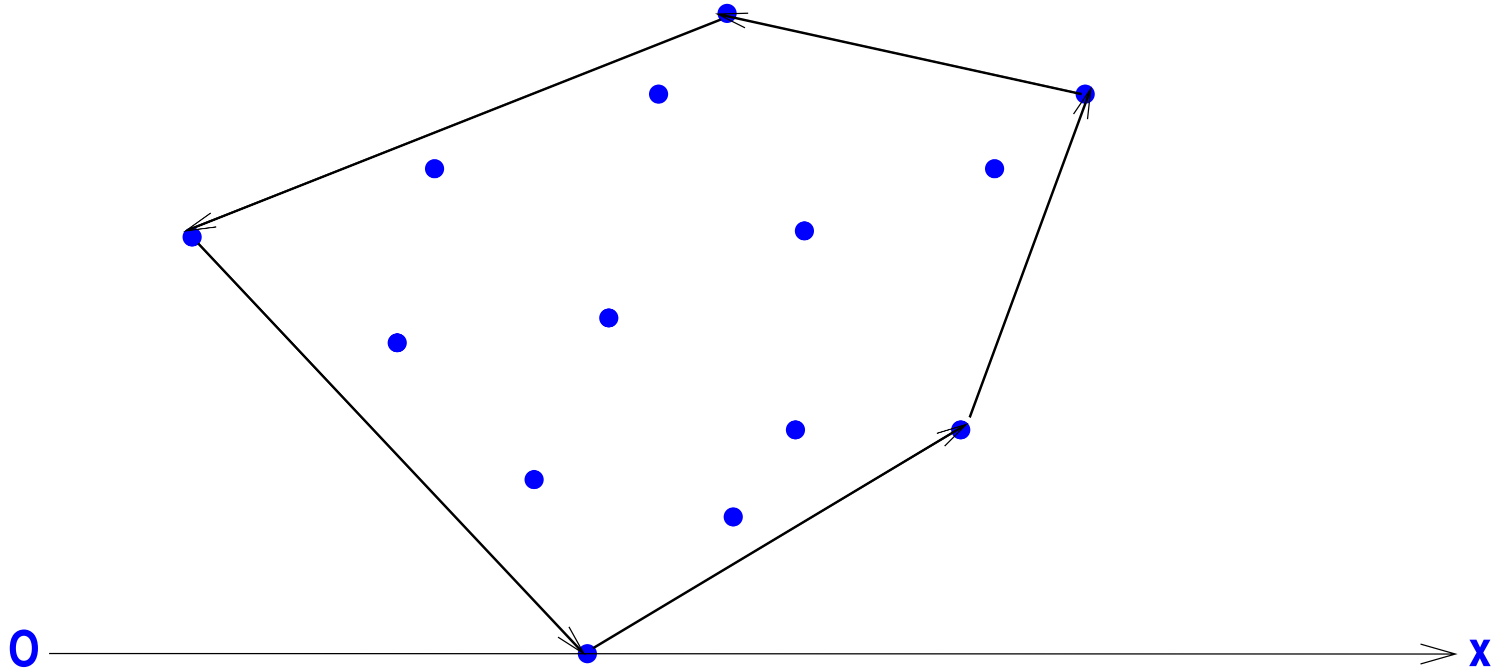
Cette fonction évite le long calcul de Math.atan(dy/dx).



Enveloppe convexe (1/4)

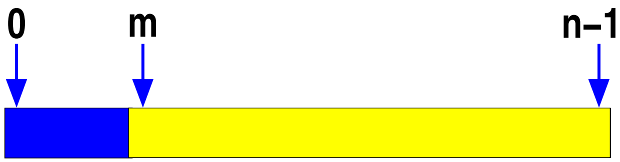
[Jarvis]

- Chercher un point avec y minimum.
- Chercher les points successifs de l'enveloppe dans l'ordre trigonométrique.
- Un point P_{m+1} sur l'enveloppe est le P_i tel que $\overrightarrow{P_m P_i}$ fait un angle θ minimal et positif avec l'axe $\overrightarrow{0x}$.



Enveloppe convexe (2/4)

```
static int enveloppe (Point[ ] p) {
  int m = 0, n = p.length;
  if (n > 0) {
    int iMin = 0;
    for (int i = 1; i < n; ++i) if (p[i].y < p[iMin].y) iMin = i;
    float angleMin = 400;
    do {
      Point t = p[m]; p[m] = p[iMin]; p[iMin] = t;
      ++m; iMin = 0;
      for (int i = m; i < n; ++i) {
        float alpha = theta(p[m-1], p[i]);
        if (alpha < angleMin) { iMin = i; angleMin = alpha; }
      }
      angleMin = theta(p[iMin], p[0]);
    } while (iMin != 0);
  }
  return m;
}
```

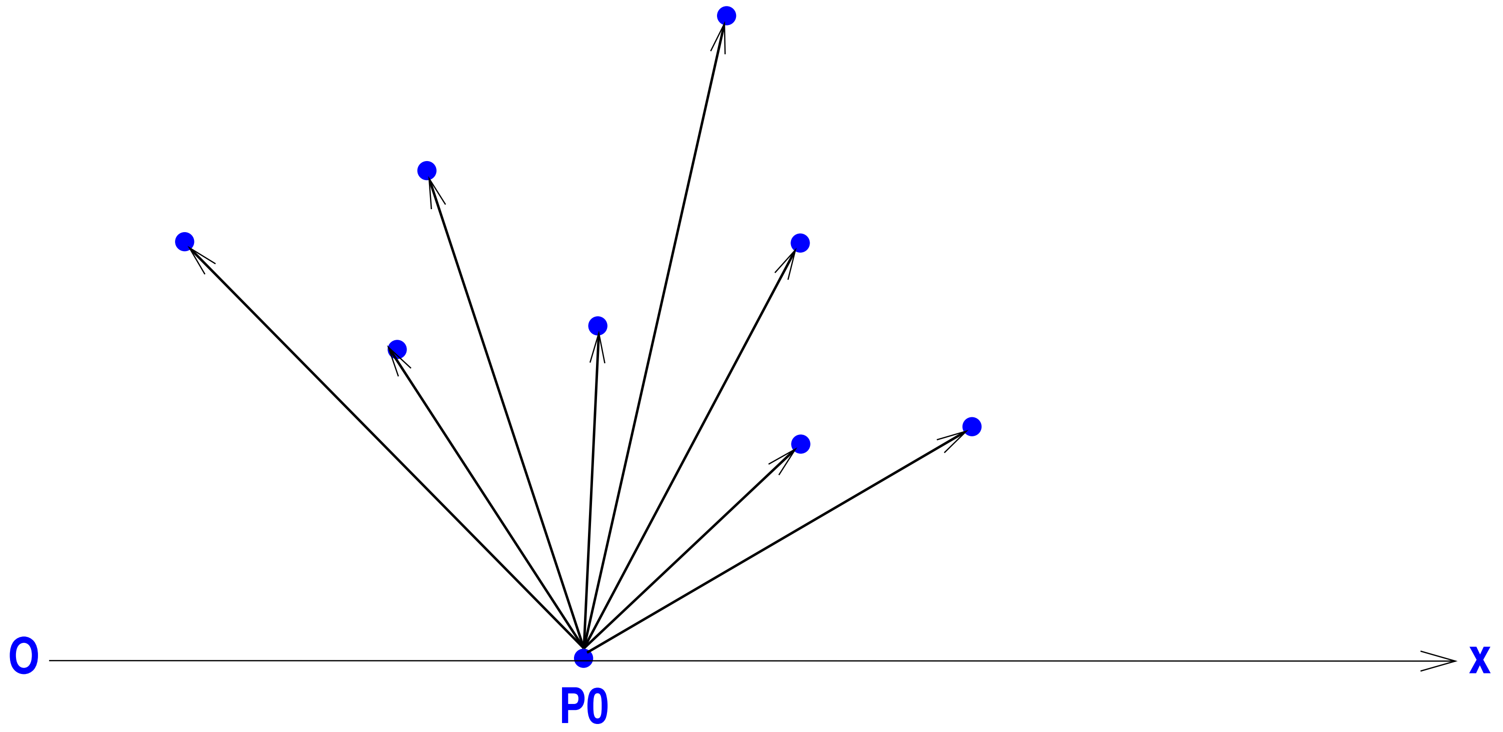


Complexité $= O(n^2)$

Enveloppe convexe (3/4)

[Graham]

- Chercher un point P_0 avec y minimum.
- Trier les points P_i sur l'angle formé avec P_0.
- Partir de ce point en tournant toujours à gauche.



Enveloppe convexe (3/4)

```
static int enveloppe (Point[ ] p) {
  int n = p.length; if (n <= 2) return n;
  else {
    int iMin = 0;
    for (int i = 1; i < n; ++i)
      if (p[i].y < p[iMin].y || p[i].y == p[iMin].y && p[i].x > p[iMin].x)
        iMin = i;
    Point t = p[0]; p[0] = p[iMin]; p[iMin] = t;
    trier(p); int m = 2;
    for (int i = 3; i < n; ++i) {
      while (ordreTrigo(p[m], p[m-1], p[i]) >= 0)
        --m;
      ++m;
      t = p[m]; p[m] = p[i]; p[i] = t;
    }
    return m+1;
  }
}
```

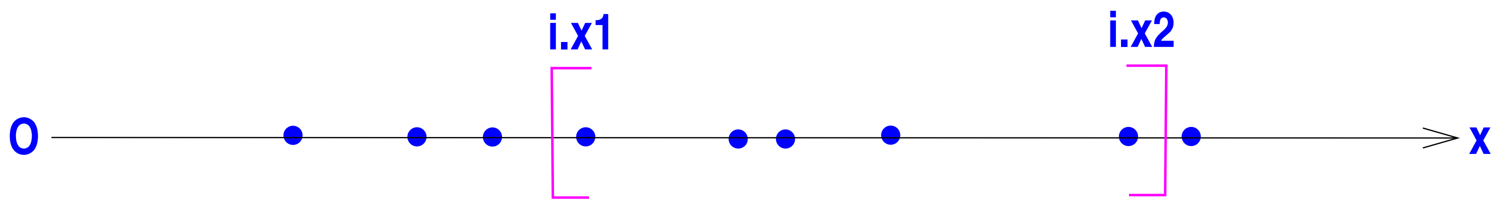
Exercice 3 Complexité $= O(n \log n)$

Recherche dans des intervalles (1/3)

En dimension 1, représenter les points avec un arbre de recherche.

```
static void rechercher (Arbre a, intervalle i) {
  if (a != null) {
    boolean bg = i.x1 <= a.x, bd = a.x <= i.x2;
    if (bg) rechercher (a.gauche, i);
    if (bg && bd) System.out.print (a.x + " ");
    if (bd) rechercher (a.droite, i);
  }
}
```

Exercice 4 Complexité ?



Recherche dans des intervalles (2/3)

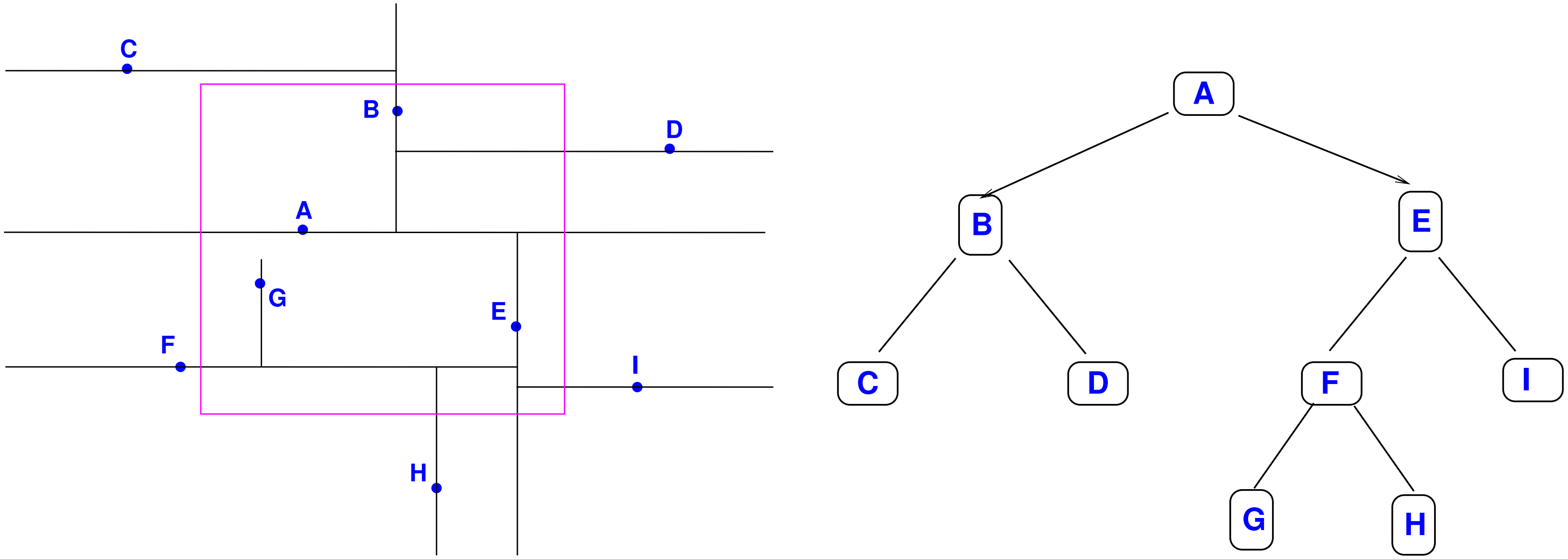
En dimension 2, arbre de recherche en alternant le rangement sur x et y.

```
static void rechercher (Arbre a, Rect r, boolean d) {
  boolean b1, b2;
  if (a != null) {
    boolean bx1 = r.x1 <= a.p.x, bx2 = a.p.x <= r.x2;
    boolean by1 = r.y1 <= a.p.y, by2 = a.p.y <= r.y2;
    if (d) { b1 = bx1; b2 = bx2; }
    else { b1 = by1; b2 = by2; }
    if (b1)
       rechercher (a.gauche, r, !d);
    if (dansRect (a.p, r))
       System.out.print (a.p + " ");
    if (b2)
       rechercher (a.droite, r, !d);
   }
}
```

Exercice 5 Complexité ?

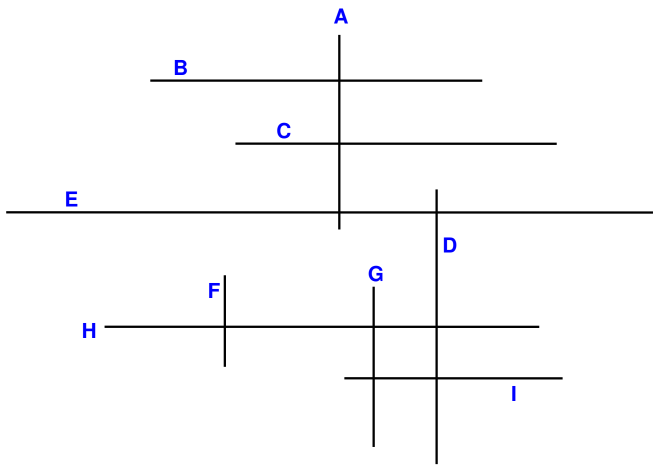
Recherche dans des intervalles (3/3)

Graphiquement



Intersection de segments orthogonaux

- On balaie le plan avec une ligne horizontale de bas en haut (*scanline*). Il faut trier les extrémités des segments sur y.
- A chaque point début de segment vertical, on rajoute dans l'arbre de recherche sa coordonnée x.
- A chaque segment horizontal, on fait une recherche des points dans l'intervalle correspondant au segment.
- A chaque fin de segment vertical, on retire de l'arbre de recherche la coordonnée x.
- $O(k + n \log n)$.



Intersection de segments (1/3)

[Shamos-Hoey]

Recherche d'une intersection :

- On balaie le plan avec une ligne horizontale de bas en haut (*scanline*). Il faut donc trier les extrémités des segments. Soit Q cet ensemble. Au début, $R = \emptyset$.
- A chaque point p dans Q dans l'ordre des y croissants,

  1. si p est le début du segment s. On insère s dans R. On teste si s intersecte le segment de gauche ou de droite dans R, et on retourne cette intersection.
  2. si p est la fin du segment s. On teste si les segments à gauche de s et à droite de s dans R s'intersectent, et on retourne cette intersection. On enlève s de R.

- $O(n \log n)$.

Intersection de segments (2/3)

[Bentley-Ottmann, 79]

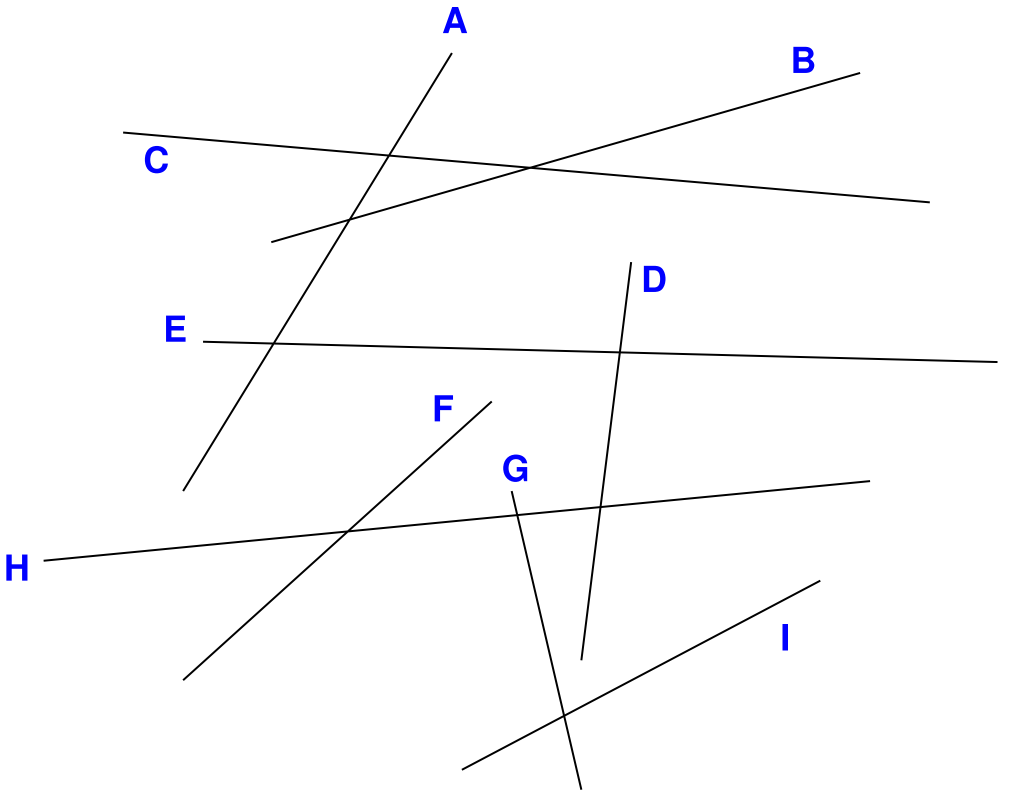
- On balaie le plan avec une ligne horizontale de bas en haut (*scanline*). Il faut donc trier les extrémités des segments. Soit Q cet ensemble. Au début, $R = \emptyset$.
- A chaque point p dans Q dans l'ordre des y croissants,

  1. si p est le début du segment s. On insère s dans R. Si s intersecte le segment de gauche ou de droite t dans R, on rajoute le point d'intersection de s et t dans Q (en respectant l'ordre des y croissants).
  2. si p est la fin du segment s. Si l'intersection des segments à gauche de s et à droite de s dans R n'est pas dans Q, on teste l'intersection et on l'ajoute à Q. On enlève s de R.
  3. si p est l'intersection de s et t, on écrit p et on échange s et t dans R. (Remarque: ils sont alors adjacents). On teste si le segment de gauche s s'intersecte avec le segment à gauche de lui dans R, et le segment à droite de t, et on rajoute cette intersection à Q.

Intersection de segments (3/3)

[Bentley-Ottmann, 79] (cf. l'appliquette)

- $O(n \log n + k \log n)$, en utilisant des arbres équilibrés pour R et Q comme une file de priorité.



La vision et la synthèse d'images sont enseignées en Majeure 1 et 2.