
Informatique Graphique

Sylvain Contassot-Vivier

Phuc Ngo

Christian Minich

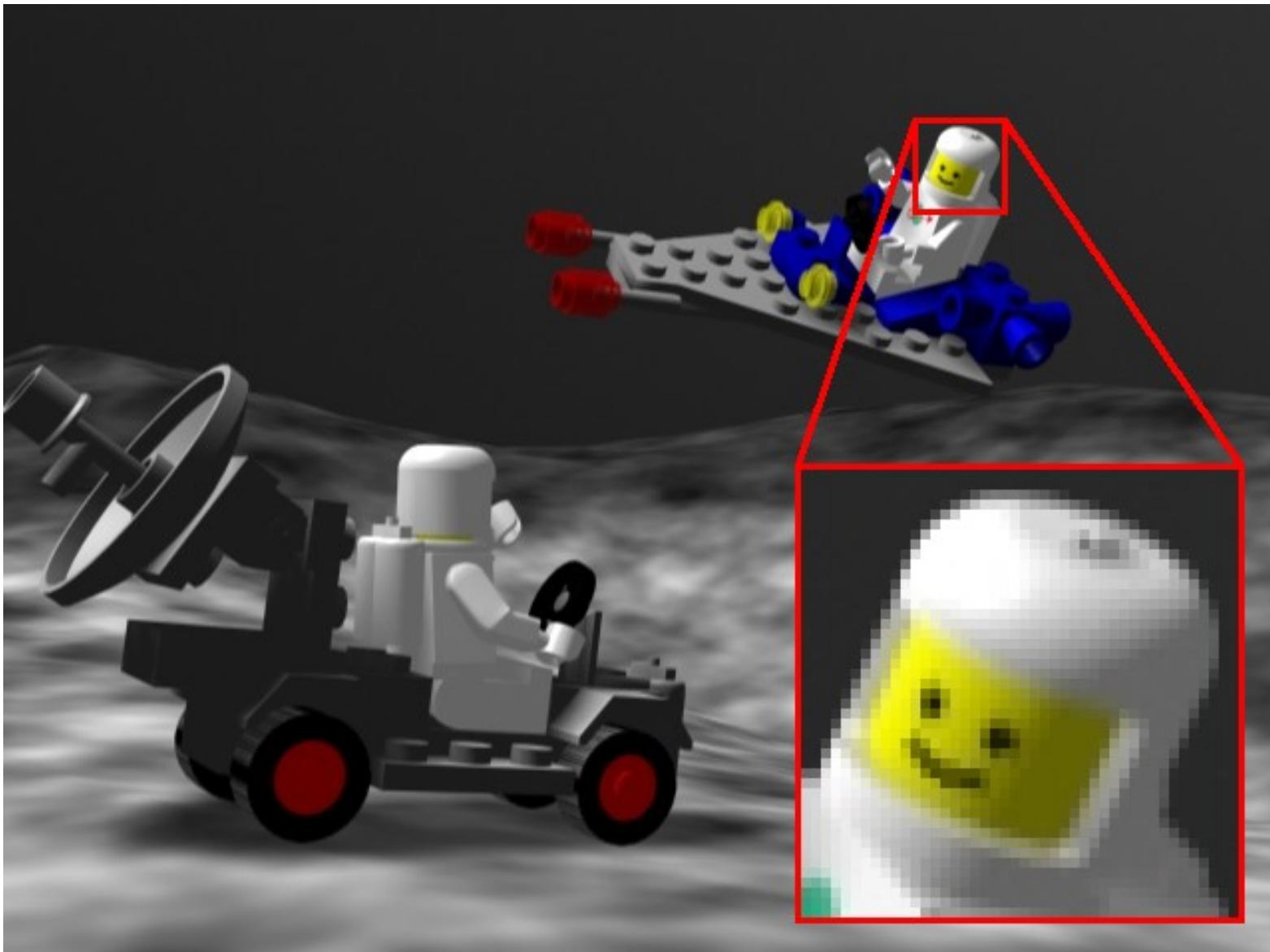
Image numérique

- Définition :
 - Contexte d'images numériques 2D
 - Écran subdivisé en grille régulière de *pixels*
- Types d'images numériques :
 - Deux grands types :
 - Images vectorielles
 - Images matricielles (bitmap)

Images matricielles

- Grille régulière de pixels :
 - Couleur varie d'un pixel à l'autre
 - Directement affichable sur les écrans
 - Encombrement indépendant du nombre d'objets dans l'image (mais de la taille de la grille)
 - Limitation du zoom
 - Pas de séparation sémantique des objets
 - Plutôt adaptée aux images naturelles :
 - Photos, dessins,...

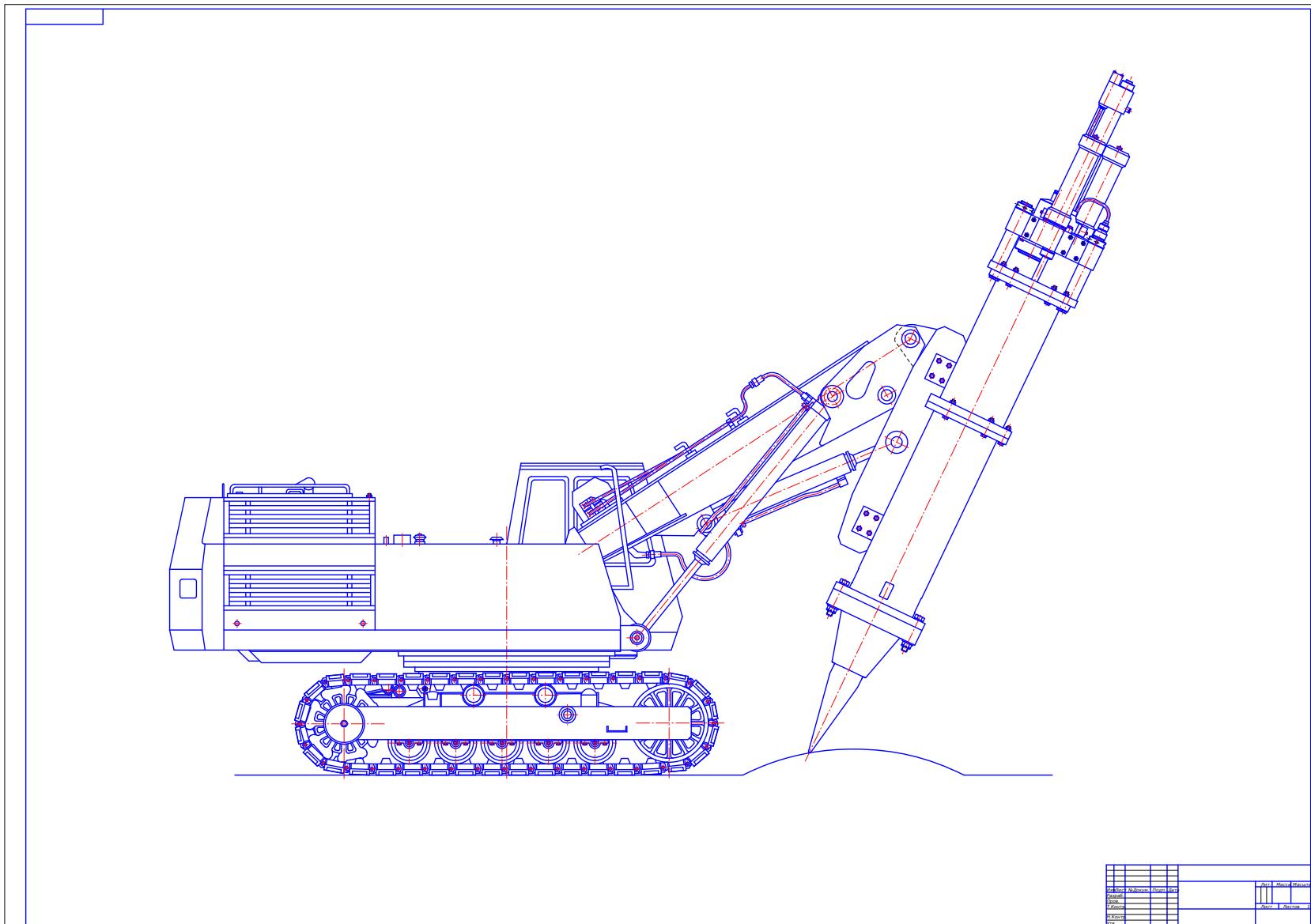
Image numérique 2D matricielle



Images vectorielles

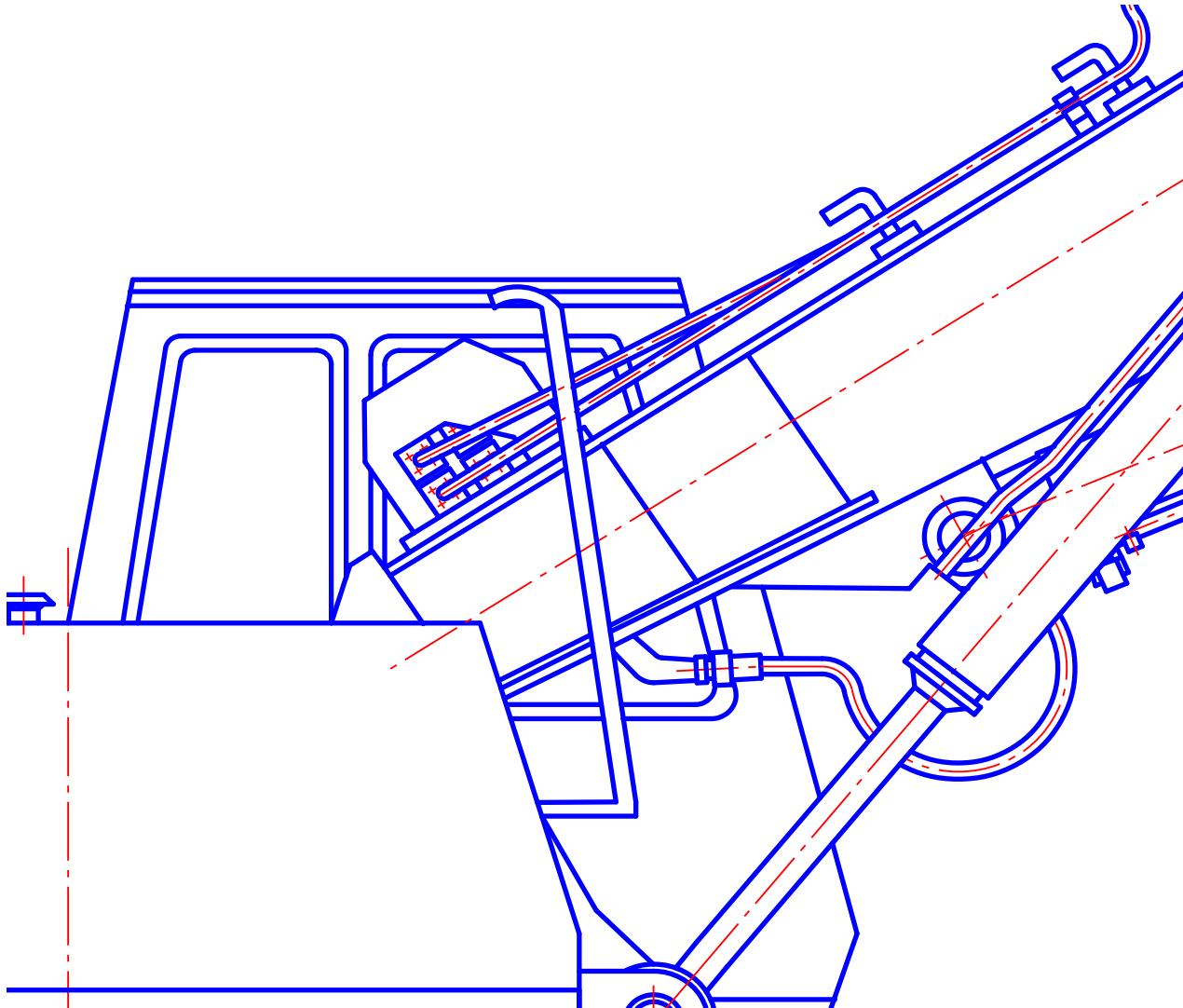
- Image composée d'une liste d'objets définis mathématiquement :
 - Lignes, rectangles, cercles,...
 - Peu encombrant si peu d'objets dans l'image mais peut devenir très encombrant si beaucoup d'objets
 - Précision illimitée (zoom arbitraire)
 - Difficulté à représenter des objets avec textures
 - Possibilité de modifier les objets indépendamment les uns des autres
 - Plutôt adaptée au dessin technique :
 - Plans, schémas,...

Image numérique 2D vectorielle



Source : Wikipedia → https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

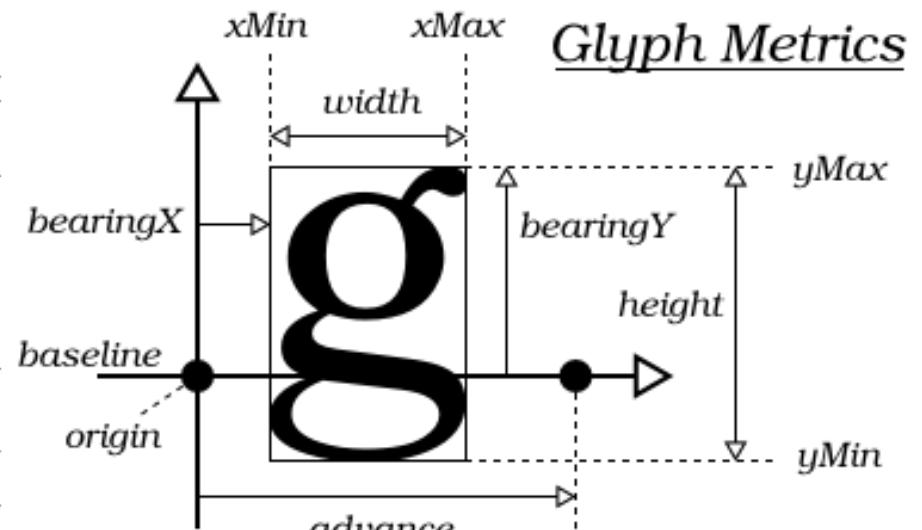
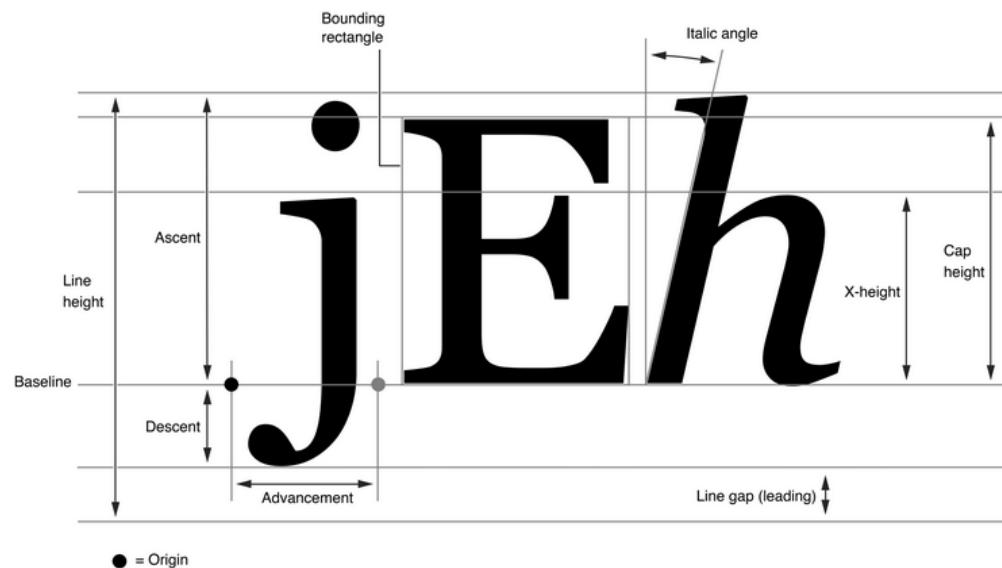
Agrandissement vectoriel



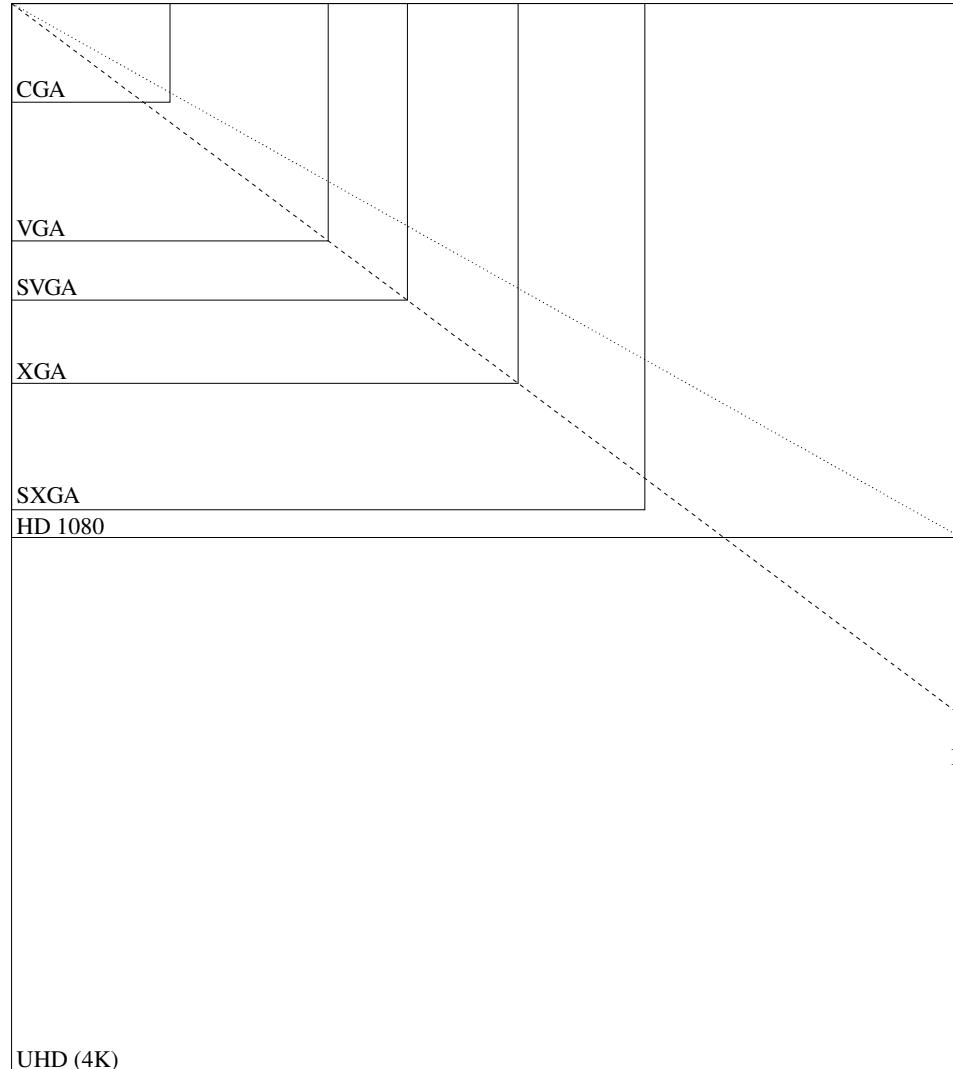
Source : Wikipedia → https://en.wikipedia.org/wiki/Scalable_Vector_Graphics

Police de caractères

- Police TrueType et FreeType



Principales résolutions standards

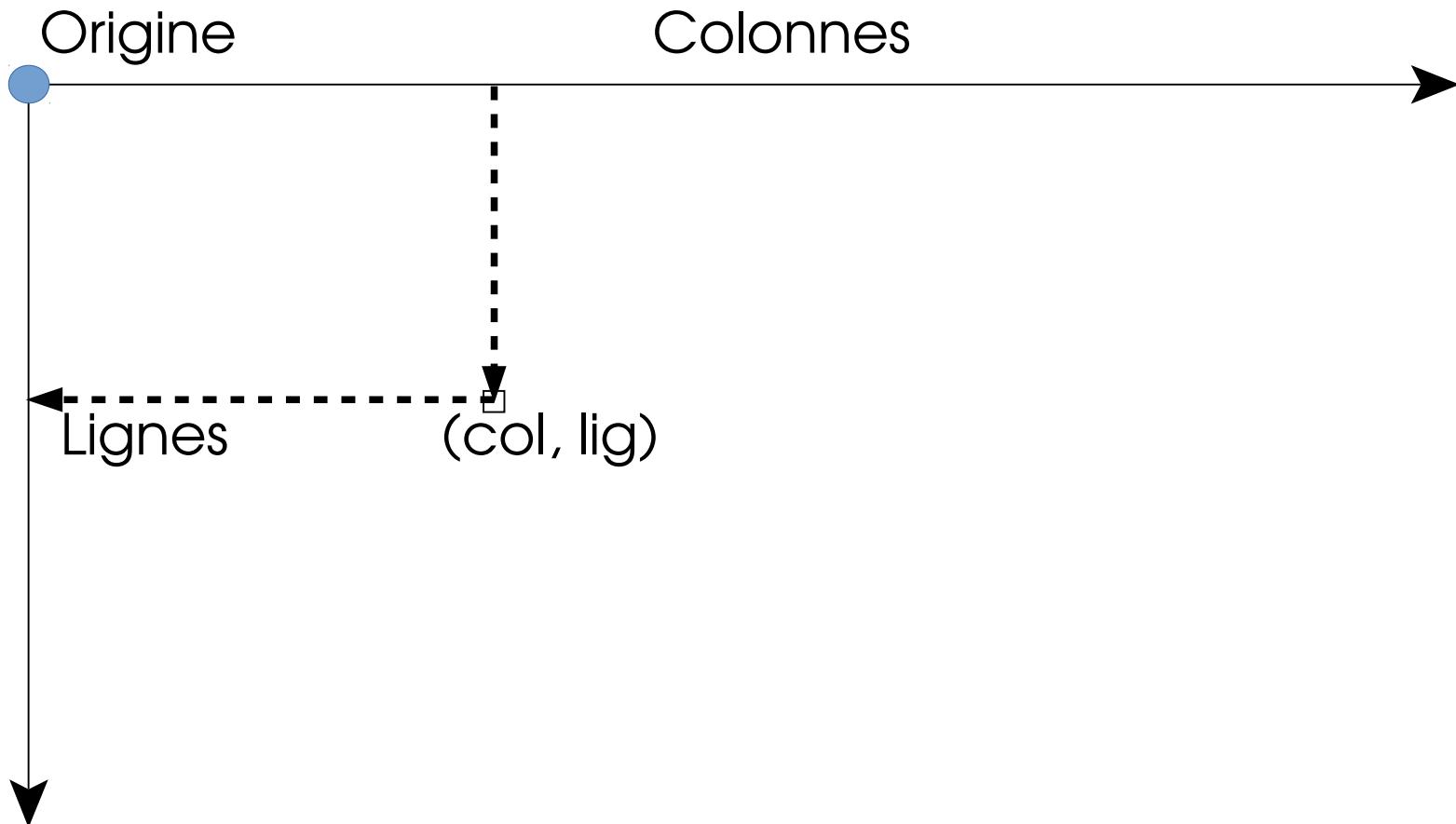


Nom	résolution	format (proportions)
CGA	320x200	(16 :10)
VGA	640x480	(4 :3)
PAL	768x576	(4 :3)
SVGA	800x600	(4 :3)
XGA	1024x768	(4 :3)
SXGA	1280x1024	(5 :4)
HD720	1280x720	(16 :9)
HD1080	1920x1080	(16 :9)
UHD (4K)	3840x2160	(16 :9)
UHD (8K)	7680x4320	(16 :9)

Format 4:3

Format 16:9

Repère de l'image



Quantification et codages binaires



1 bit par pixel



2 bits par pixel



4 bits par pixel



8 bits par pixel

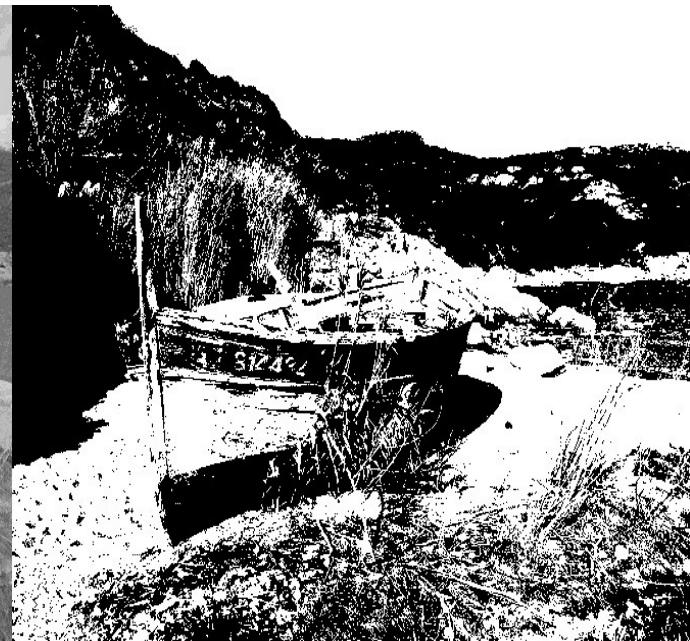
Exemple visuel en niveaux de gris



8 bits / pixel



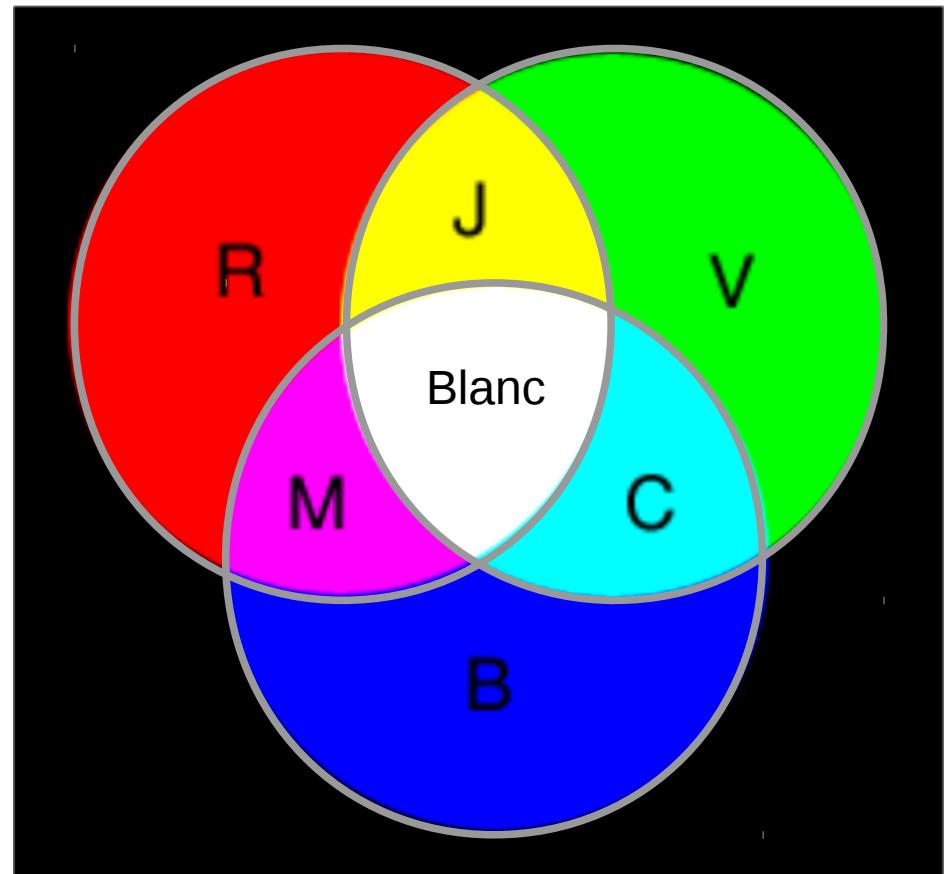
4 bits / pixel



1 bit / pixel

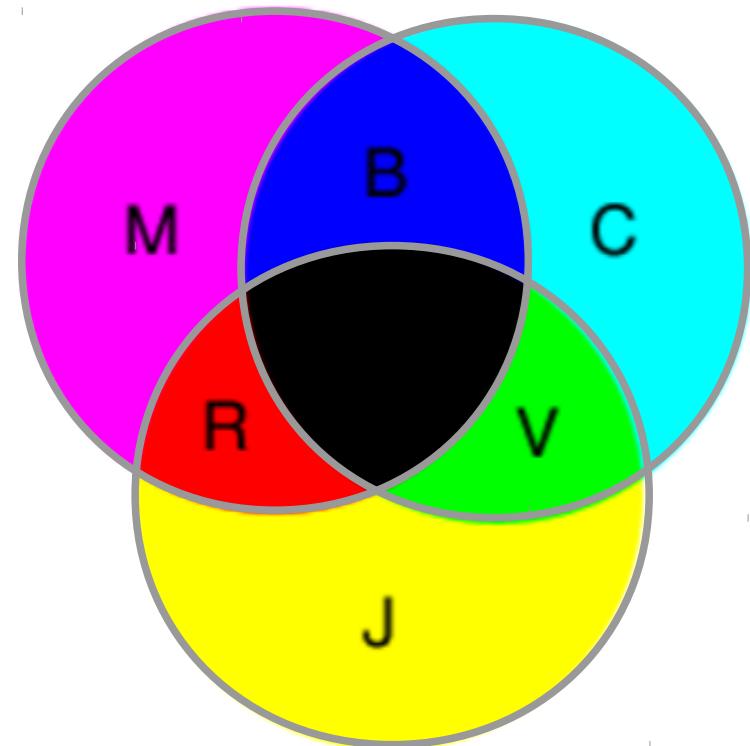
Modèle additif

- Couleurs primaires :
 - Rouge
 - Vert
 - Bleu
- Couleurs secondaires :
 - Jaune
 - Cyan
 - Magenta
- Relations :
 - $R+V = J$
 - $R+B = M$
 - $V+B = C$
 - $R+V+B = Blanc$

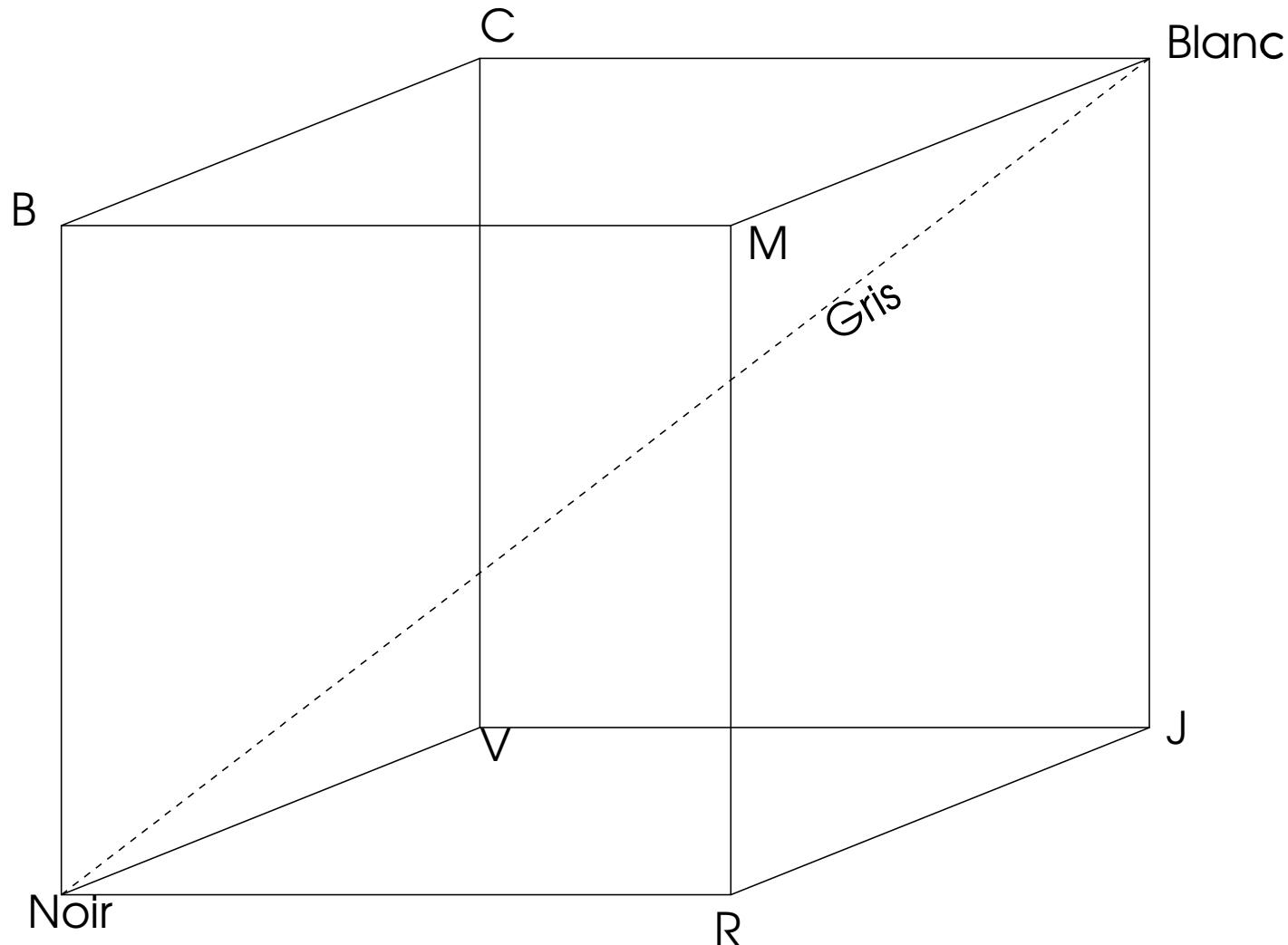


Modèle soustractif

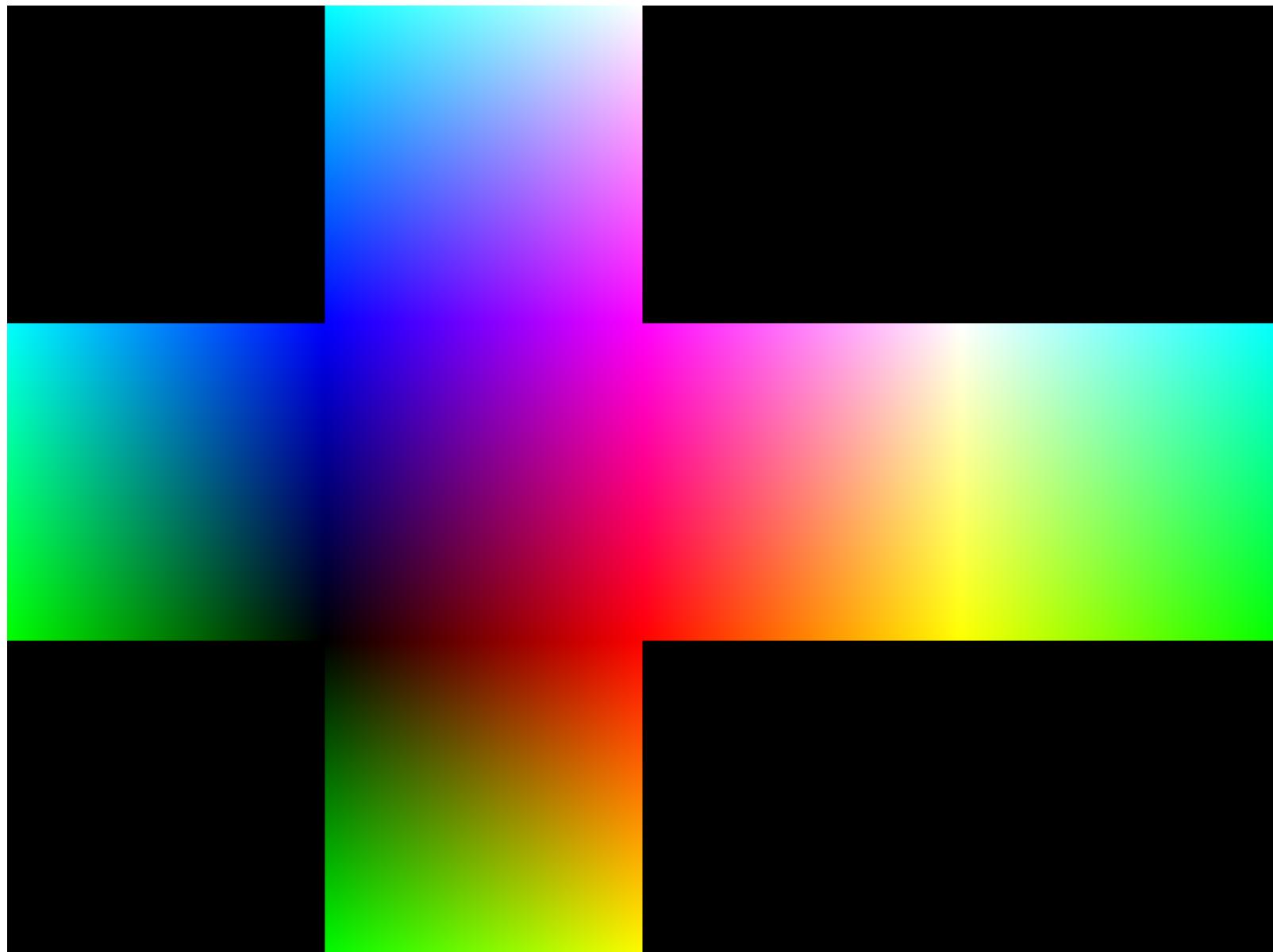
- Couleurs primaires :
 - Jaune
 - Cyan
 - Magenta
- Couleurs secondaires :
 - Rouge
 - Vert
 - Bleu
- Relations :
 - Blanc - (J+C) = V
 - Blanc - (J+M) = R
 - Blanc – (C+M) = B
 - Blanc – (J+C+M) = Noir



Cube des couleurs RVB



Cube déplié



Structure de données

- Définition du type Couleur :

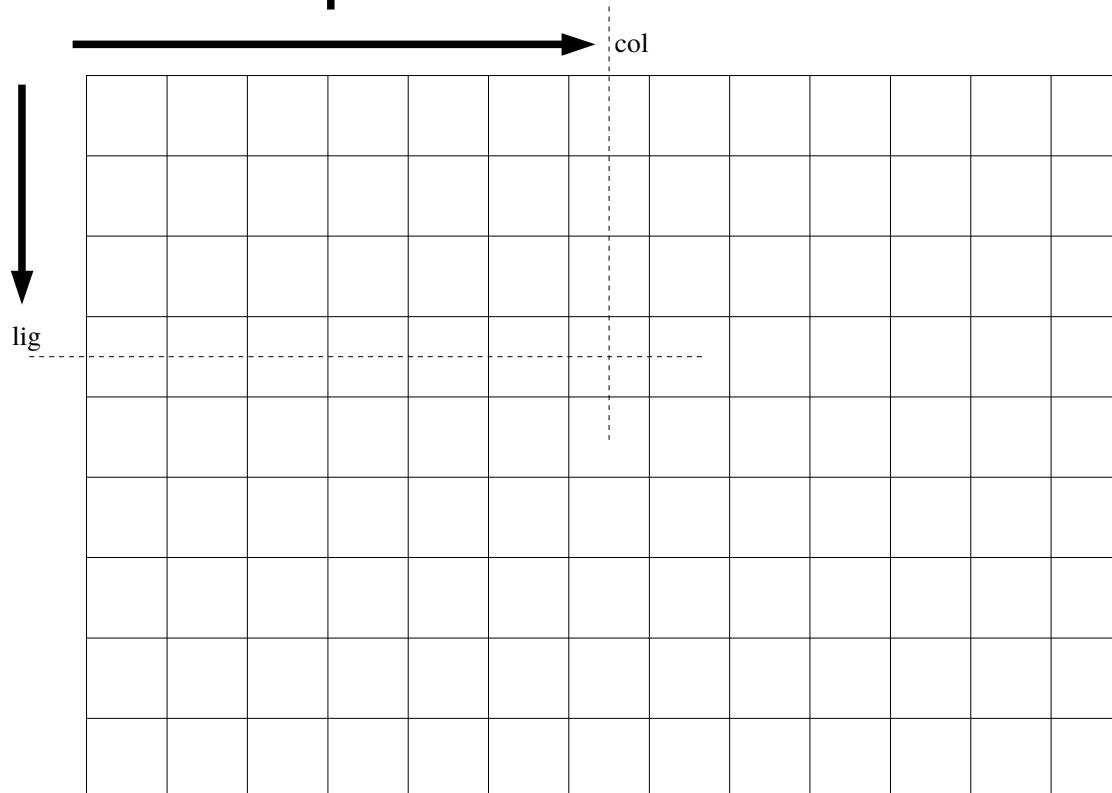
```
Couleur = structure
    R : entier // Composante rouge
    V : entier // Composante verte
    B : entier // Composante bleue
```

- Fonction de coloriage d'un pixel :

```
fonction ColoriePixel(col : entier, lig : entier, coul : Couleur) : Vide
// Colorie le pixel à la position (col,lig) dans l'image avec la couleur coul
```

Position d'un pixel

- La position d'un pixel est définie en son centre

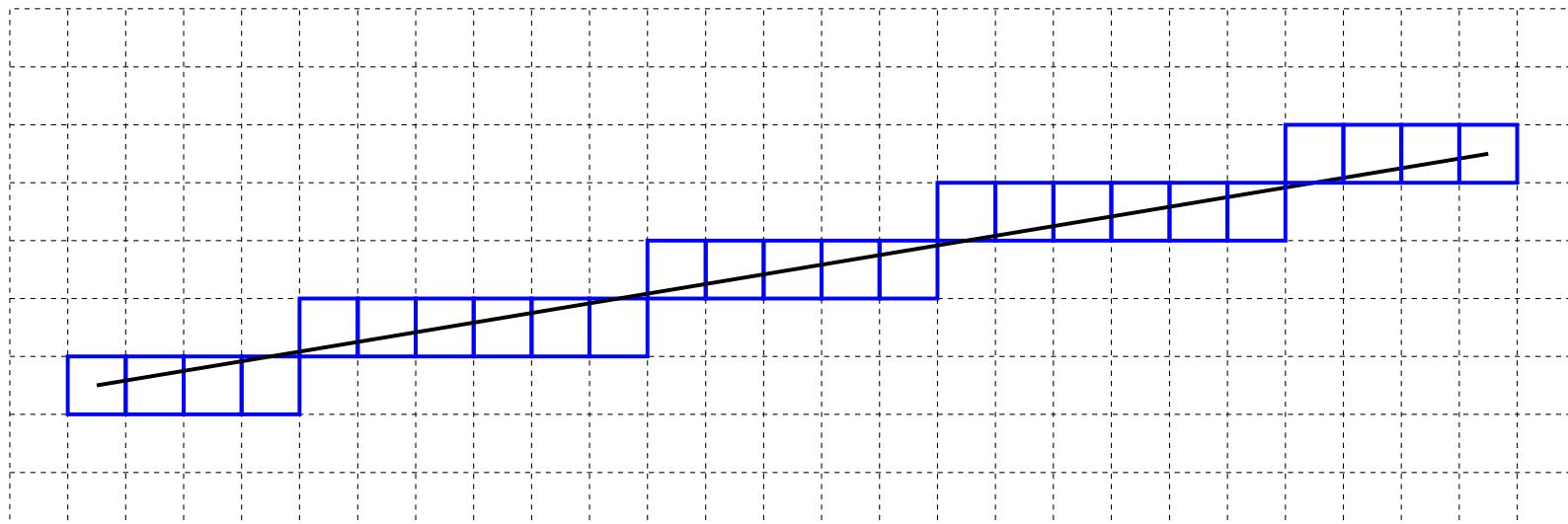


- Structure d'un point de l'image :

```
PointImage = structure
    col : entier // Colonne du point dans l'image
    lig : entier // Ligne du point dans l'image
```

Tracé de segment dans l'image

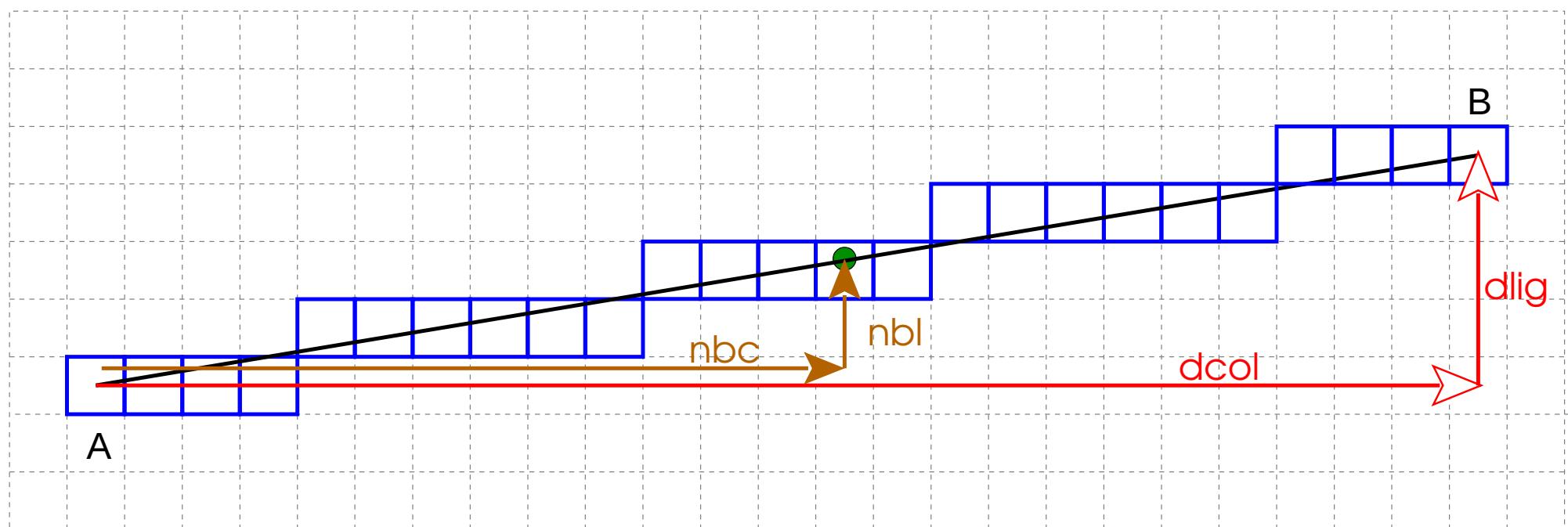
- On donne deux points A et B dans l'image :
 - $A = (A_c, A_l)$ et $B = (B_c, B_l)$
- On souhaite tracer le segment entre ces deux points dans l'image en respectant les règles suivantes :
 - Colorier uniquement des pixels traversés par le segment
 - Pas de trou dans le segment (assurer la continuité)
 - Pas plus de 2 voisins par pixel du segment



Pixels à colorier

Première version (analytique)

- On utilise l'équation de la droite :
 - On calcule la pente : $dlig / dcol$
 - À partir de l'extrémité gauche, on parcourt les colonnes et on calcule les lignes
 - On prend comme référence le point de départ



Algorithme analytique simple

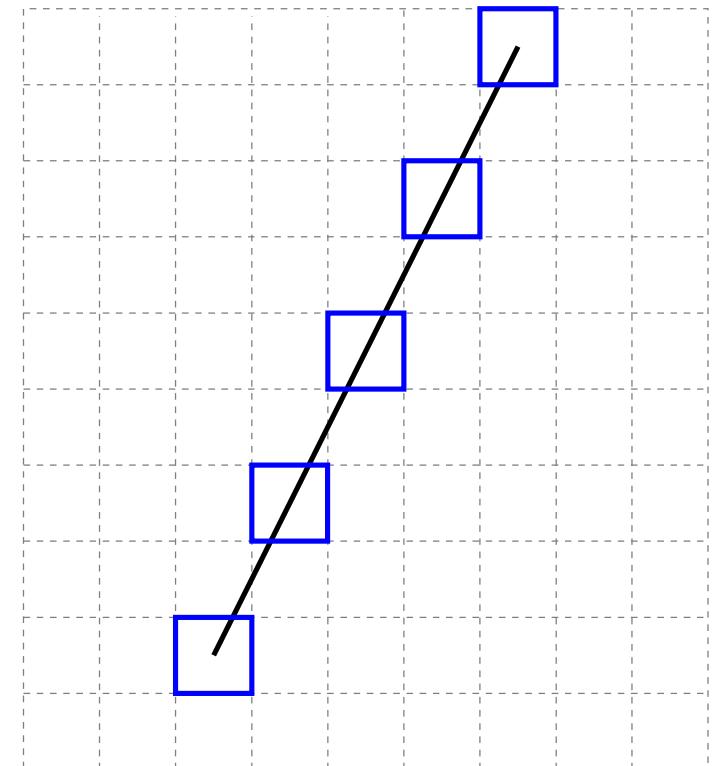
- Version considérant que le point A est l'extrémité gauche du segment

```
fonction DessineSegmentImage(A : PointImage, B : PointImage, coul : Couleur) : Vide
// Trace le segment entre les pixels A et B avec la couleur coul

dlig ← B.lig - A.lig // Variation en lignes
dcol ← B.col - A.col // Variation en colonnes
pente ← dlig / dcol // Pente de la droite AB
// Parcours des colonnes et calcul des lignes
pour col de A.col à B.col faire
    lig ← arrondi((col - A.col) * pente) + A.lig
    ColoriePixel(col, lig, coul)
fpour
```

Problèmes de la version analytique

- Modèle utilisé non général :
 - Segment vertical → pente infinie
- Nécessaire de trouver le point à gauche
 - Déterminer l'ordre de A et B
- Calculs inutiles si segment horizontal
 - Mauvaise performance
- Mais surtout, il peut y avoir des trous !!
 - Lorsque la pente $> 45^\circ$
 - Il faudrait colorier plusieurs pixels pour chaque colonne
 - Alors que l'on n'en colorie qu'un seul

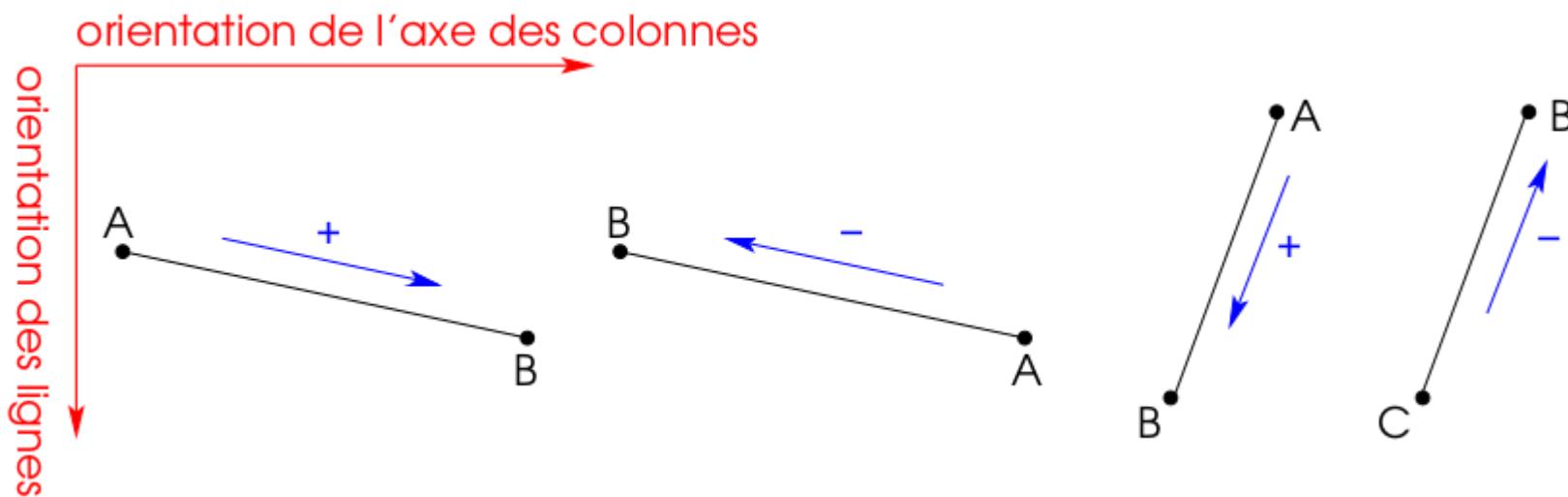


Solution générale

- On va séparer le problème en différents cas :
 - Segment horizontal :
 - Parcours des colonnes sur une même ligne
 - Segment vertical :
 - Parcours des lignes sur une même colonne
 - Segment de pente $\leq 45^\circ$:
 - Algo précédent
 - Parcours des colonnes et calcul des lignes
 - Segment de pente $> 45^\circ$:
 - Symétrie de l'algo précédent par échange des lignes et colonnes
 - Parcours des lignes et calcul des colonnes
 - Ordre des extrémités : segment [AB]
 - Parcours de A à B quelles que soient leurs positions respectives

Sens de parcours

- Dépend de la position des deux extrémités



Algorithme analytique général

```

fonction DessineSegmentImage(A : PointImage, B : PointImage, coul : Couleur) : Vide
// Trace le segment entre les pixels A et B avec la couleur coul

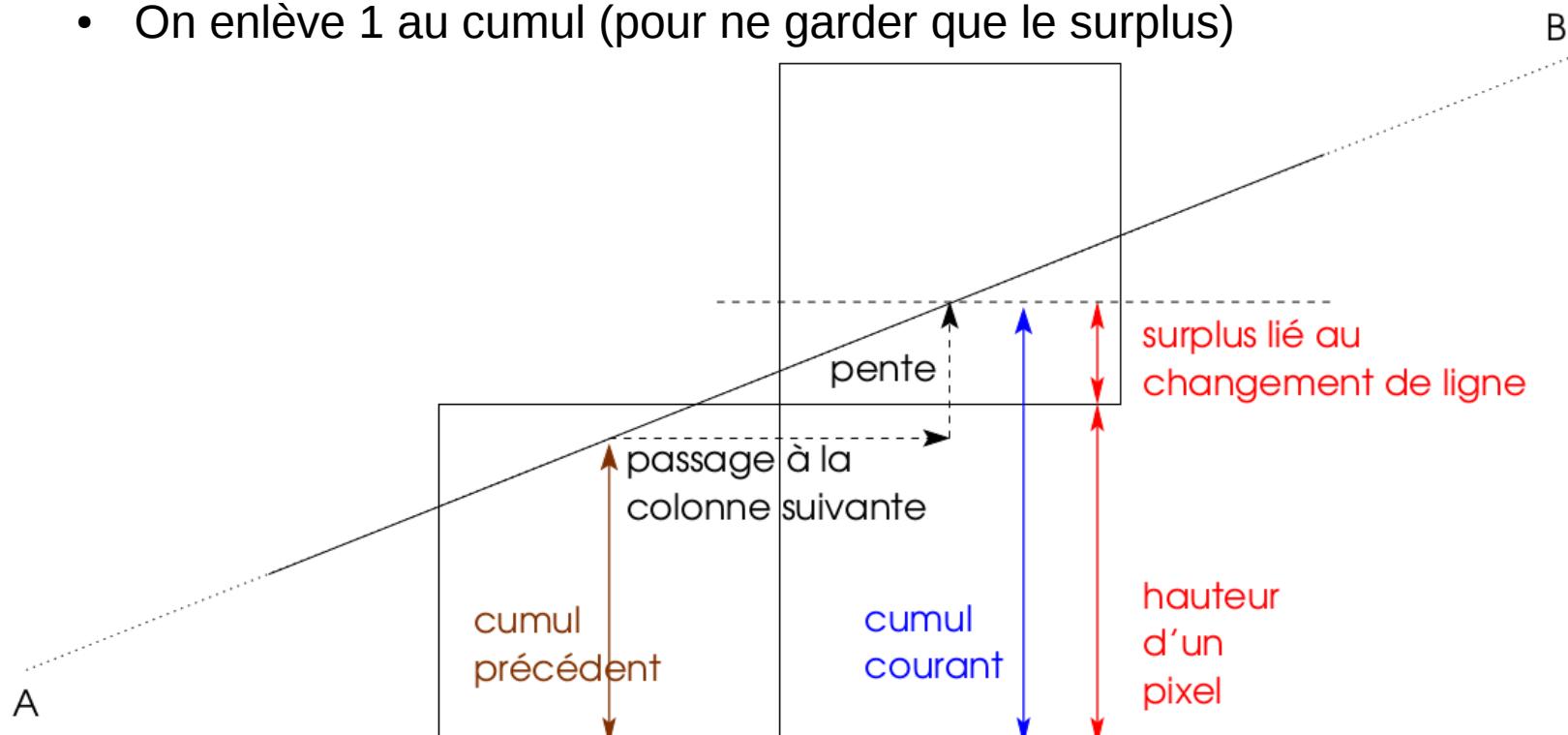
    dlig ← B.lig - A.lig // Variation en lignes
    dcol ← B.col - A.col // Variation en colonnes
    col ← A.col           // On commence le parcours du segment au point A
    lig ← A.lig
    sens ← 1              // Sens initial de parcours des lignes ou colonnes

    si dlig = 0 alors      // Segment horizontal
        si dcol < 0 alors    // Mise à jour du sens de parcours
            sens ← -1
        fsi
        tant que col ≠ B.col + sens faire
            ColoriePixel(col, lig, coul)
            col ← col + sens
        ftant
    sinon
        si dcol = 0 alors    // Segment vertical
            ... // Symétrique au segment horizontal en permutant lignes et colonnes
        sinon                  // Cas généraux
            si abs(dcol) ≥ abs(dlig) alors // Parcours des colonnes
                si dcol < 0 alors // Mise à jour du sens de parcours
                    sens ← -1
                fsi
                pente ← dlig / dcol // ATTENTION : la pente doit être calculée en réel
                tant que col ≠ B.col + sens faire
                    ColoriePixel(col, lig, coul)
                    col ← col + sens
                    lig ← arrondi((col - A.col) * pente) + A.lig // Calcul réel arrondi ↴
                                    ↴ au plus près
                ftant
            sinon // Parcours des lignes
                ... // Symétrique au parcours des colonnes
            fsi
        fsi
    fsi

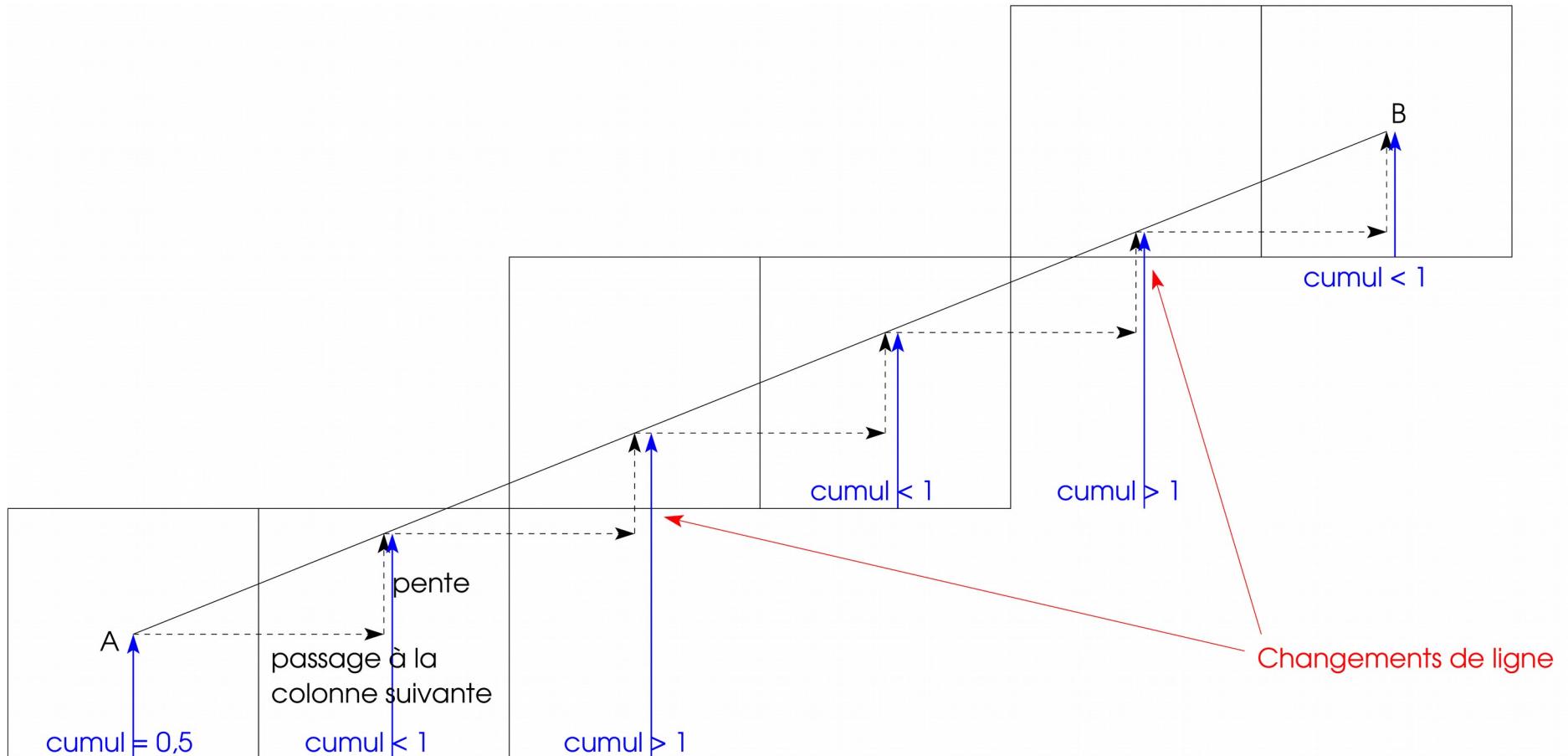
```

Version incrémentale

- **Principe :** déduire le y courant du y du pixel précédent
- Lors des déplacements unitaires selon l'axe de parcours :
 - On se déplace de la pente (<1) selon l'autre axe
 - Lorsque le cumul des déplacements dépasse 1 :
 - On change de position courante (± 1) sur le second axe
 - On enlève 1 au cumul (pour ne garder que le surplus)



Exemple de parcours



Algorithme incrémental

```

fonction DessineSegmentImage(A : PointImage, B : PointImage, coul : Couleur) : Vide
// Trace le segment entre les pixels A et B avec la couleur coul

    dlig ← B.lig - A.lig // Variation en lignes
    dcol ← B.col - A.col // Variation en colonnes
    col ← A.col           // On commence toujours au point A
    lig ← A.lig
    sensLig ← 1            // Sens initial de parcours des lignes
    sensCol ← 1            // Sens initial de parcours des colonnes
    cumul ← 0,5             // Le cumul commence au centre du pixel

    si dcol < 0 alors      // Mise à jour du sens de parcours des colonnes
        sensCol ← -1
    fsi
    si dlig < 0 alors      // Mise à jour du sens de parcours des lignes
        sensLig ← -1
    fsi
    si dlig = 0 alors      // Segment horizontal
        ... // Similaire à la version précédente
    sinon
        si dcol = 0 alors    // Segment vertical
            ... // Symétrique au segment horizontal en permutant lignes et colonnes
        sinon                  // Cas généraux
            si abs(dcol) ≥ abs(dlig) alors // Parcours des colonnes
                pente ← abs(dlig / dcol) // ATTENTION : la pente doit être un réel
                tant que col ≠ B.col + sens faire
                    ColoriePixel(col, lig, coul)
                    col ← col + sensCol
                    cumul ← cumul + pente // Ajout de la pente au cumul
                    si cumul > 1 alors // Test de changement de ligne
                        lig ← lig + sensLig // Changement de ligne dans le sens qui convient
                        cumul ← cumul - 1,0 // Réduction pour ne garder que le surplus
                    fsi
                    ftant
                sinon // Parcours des lignes
                    ... // Symétrique au parcours des colonnes
                fsi
            fsi
    fsi

```

Version entière

- La version précédente supprime les \times MAIS
 - Elle utilise des calculs réels coûteux, que l'on peut éliminer
- Les seuls variables réelles sont le cumul et la pente
- La pente est réelle via la division par *dcol*
 - ⇒ Il suffit de multiplier l'échelle d'un pixel par *dcol* !
- L'initialisation du cumul à *dlig/2* pose aussi problème
 - ⇒ Il suffit de multiplier l'échelle d'un pixel par 2 !
- Au final, on multiplie l'échelle d'un pixel par $2 * \text{dcol}$
- On obtient un gain de performance non négligeable :
 - 20 % sur version C, 260 % sur version python

Version entière (Bresenham)

```

fonction DessineSegmentImage(A : PointImage, B : PointImage, coul : Couleur) : Vide
// Trace le segment entre les pixels A et B avec la couleur coul

    dlig ← B.lig - A.lig // Variation en lignes
    dcol ← B.col - A.col // Variation en colonnes
    absdcol ← abs(dcol) // Valeur absolue de dcol
    absdlig ← abs(dlig) // Valeur absolue de dlig
    col ← A.col          // On commence toujours au point A
    lig ← A.lig
    sensLig ← 1           // Sens initial de parcours des lignes
    sensCol ← 1           // Sens initial de parcours des colonnes

    si dcol < 0 alors      // Mise à jour du sens de parcours des colonnes
        sensCol ← -1
    fsi
    si dlig < 0 alors      // Mise à jour du sens de parcours des lignes
        sensLig ← -1
    fsi

    si absdcol ≥ absdlig alors          // Parcours des colonnes
        cumul ← absdcol                  // Départ au centre du pixel
        tant que col ≠ B.col + sensCol faire
            ColoriePixel(col, lig, coul)
            cumul ← cumul + 2 * absdlig   // Ajout du déplacement vertical équivalent à un déplacement horizontal d'un pixel
            si cumul ≥ 2 * absdcol alors // Test de changement de ligne
                lig ← lig + sensLig       // Changement de ligne
                cumul ← cumul - 2 * absdcol // Réduction pour ne garder que le surplus
            fsi
            col ← col + sensCol
        ftant
    sinon // Parcours des lignes
        ... // Symétrique au parcours des colonnes
    fsi

```

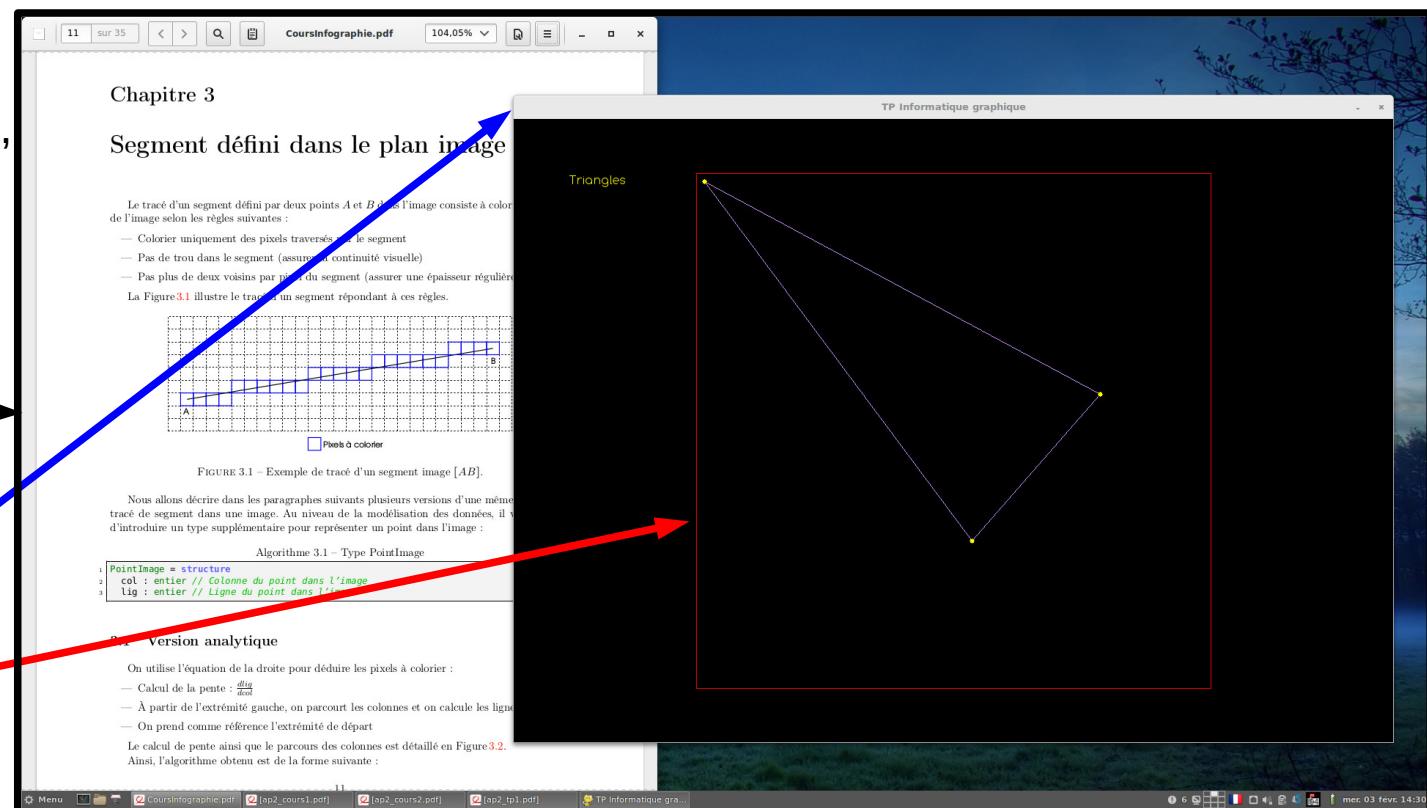
Hiérarchie de fenêtres

- Les écrans numériques ont une résolution fixe (nb de pixels), qui définit implicitement une zone rectangulaire sur l'espace N^2
- Dans les OS avec interface graphique, les applications affichent leurs informations dans des fenêtres indépendantes (images générées par l'application et affichées sur tout ou partie de l'écran)
- Enfin, dans une même fenêtre d'application, on peut avoir besoin de créer plusieurs zones de dessin (sortes de sous-fenêtres)
- On voit donc qu'il y a une hiérarchie de fenêtres (écran, fenêtre application, fenêtre dessin,...), dans laquelle chaque élément est défini dans le repère de l'élément supérieur

Copie d'un écran →

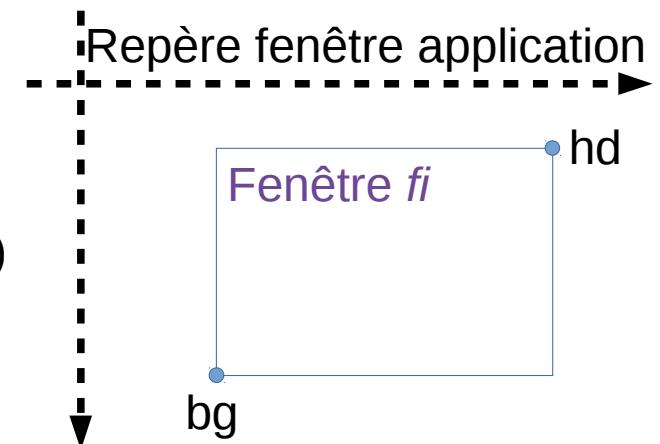
Fenêtre d'application
(notre repère global)

Fenêtre de dessin
dans l'application
(notre *fenêtre image*)



Fenêtre dans le plan image

- On s'intéresse ici à la définition d'une fenêtre de dessin dans le repère de la fenêtre d'application
 - Le niveau supérieur (avec l'écran) est géré par l'OS
 - La fenêtre de dessin (fenêtre *fi*) est définie :
 - dans le repère de la fenêtre application (notre repère de N^2)
(origine en haut à gauche, colonnes vers la droite et lignes vers le bas)
 - par deux points diagonalement opposés :
 - Choix : bas-gauche (*bg*) et haut-droite (*hd*)
- On crée pour cela une structure de fenêtre dans le plan image :



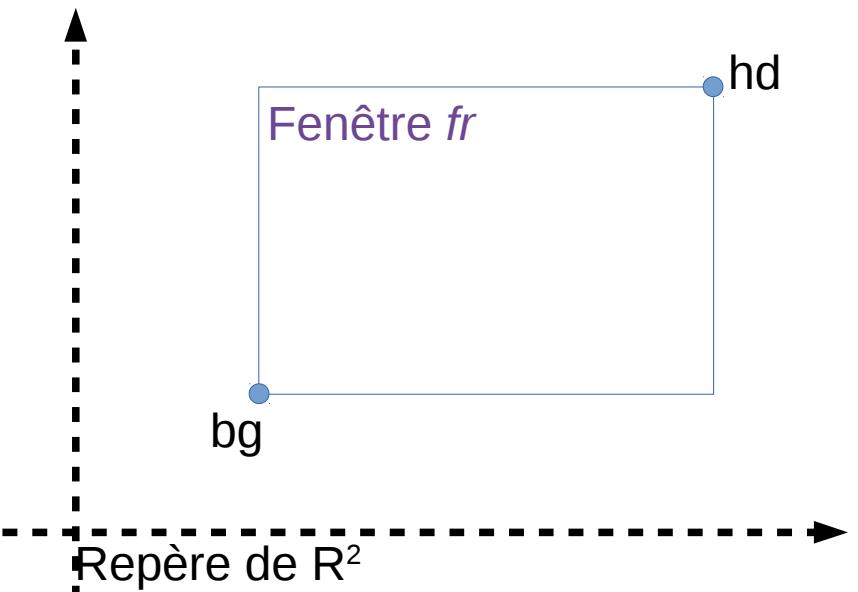
```

PointImage = structure
    col : entier // Colonne du point dans l'image
    lig : entier // Ligne du point dans l'image

FenetreImage = structure
    bg : PointImage // Sommet en bas à gauche de la fenêtre du plan image
    hd : PointImage // Sommet en haut à droite de la fenêtre du plan image
  
```

Fenêtre dans le plan réel

- De même que pour la fenêtre image, on peut définir une fenêtre dans le plan réel R^2 pour manipuler des objets continus
 - Cela définit la zone du plan (fenêtre *fr*) visualisée dans la fenêtre image *fi*
- On crée pour cela une structure de fenêtre dans le plan réel :



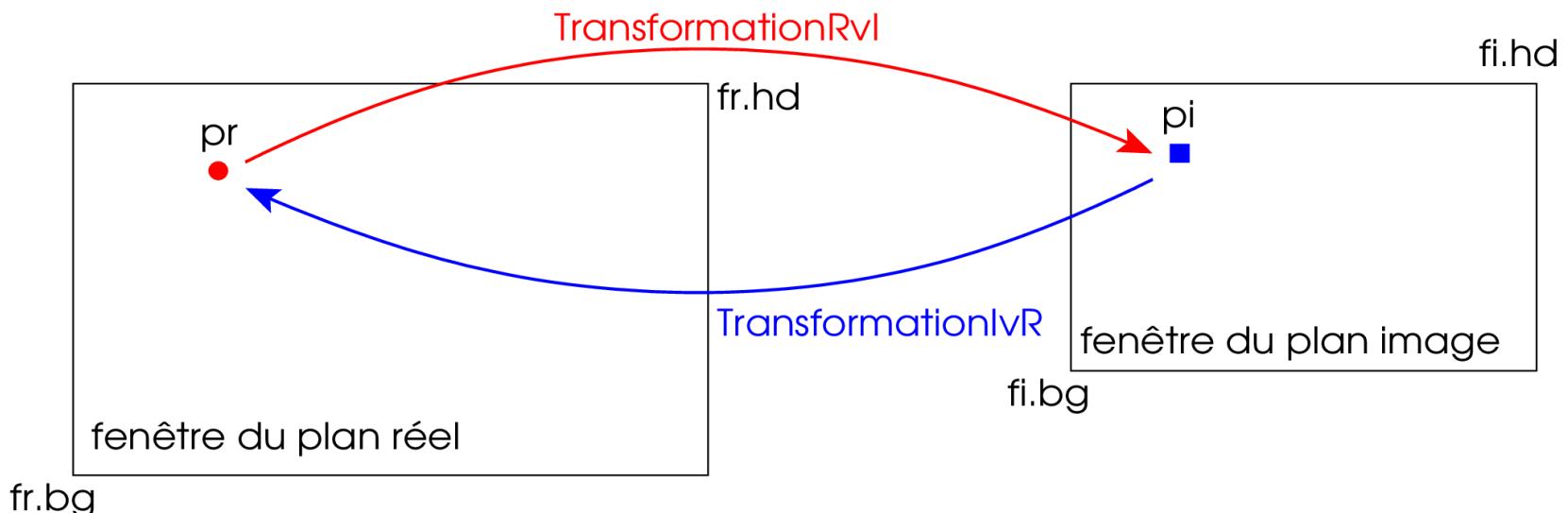
```

PointReel = structure
  x : réel // Abscisse du point
  y : réel // Ordonnée du point

FenetreReel = structure
  bg : PointReel    // Sommet en bas à gauche de la fenêtre du plan réel
  hd : PointReel    // Sommet en haut à droite de la fenêtre du plan réel
  
```

Transformations entre fenêtres

- On dispose donc de deux fenêtres définies dans deux espaces (et repères) différents
- On doit pouvoir passer de l'une à l'autre :
 - On calcule la relation de superposition des deux fenêtres :
 - un point dans une fenêtre a son correspondant dans l'autre fenêtre



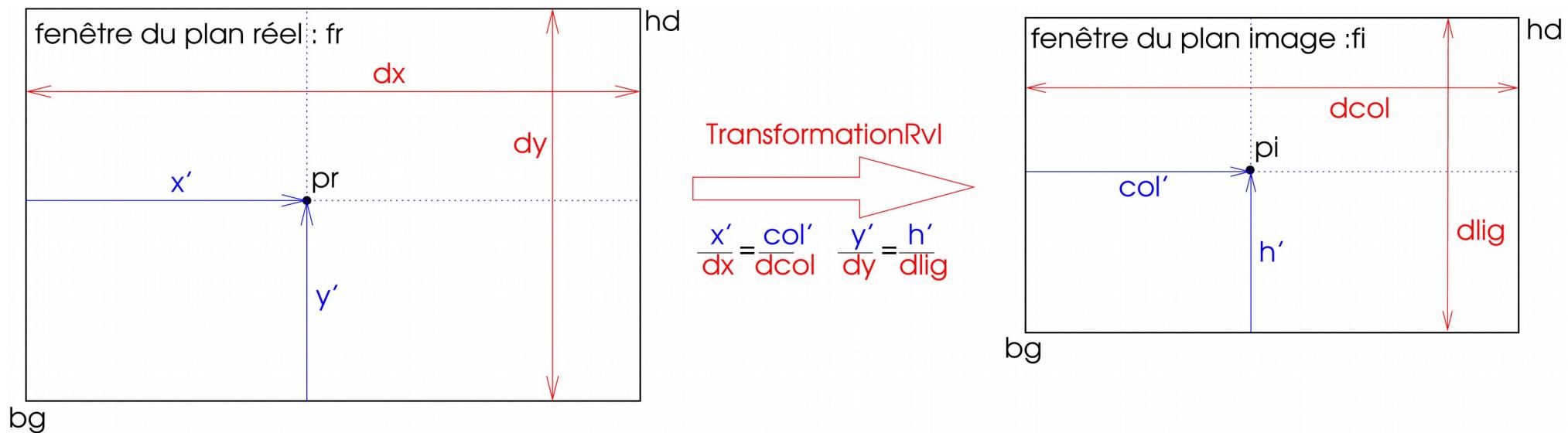
Transformations entre fenêtres

- Pour respecter la position d'un même point dans les deux fenêtres, il faut exprimer sa position relative à l'intérieur de chaque fenêtre :
 - Pourcentages de la largeur et de la hauteur
- On a donc besoin des dimensions (largeur, hauteur) des deux fenêtres :
 - Dimensions de fr :
 - $dx = fr.hd.x - fr.bg.x$
 - $dy = fr.hd.y - fr.bg.y$
 - Dimensions de fi :
 - $dcol = fi.hd.col - fi.bg.col$
 - $dlig = fi.bg.lig - fi.hd.lig$

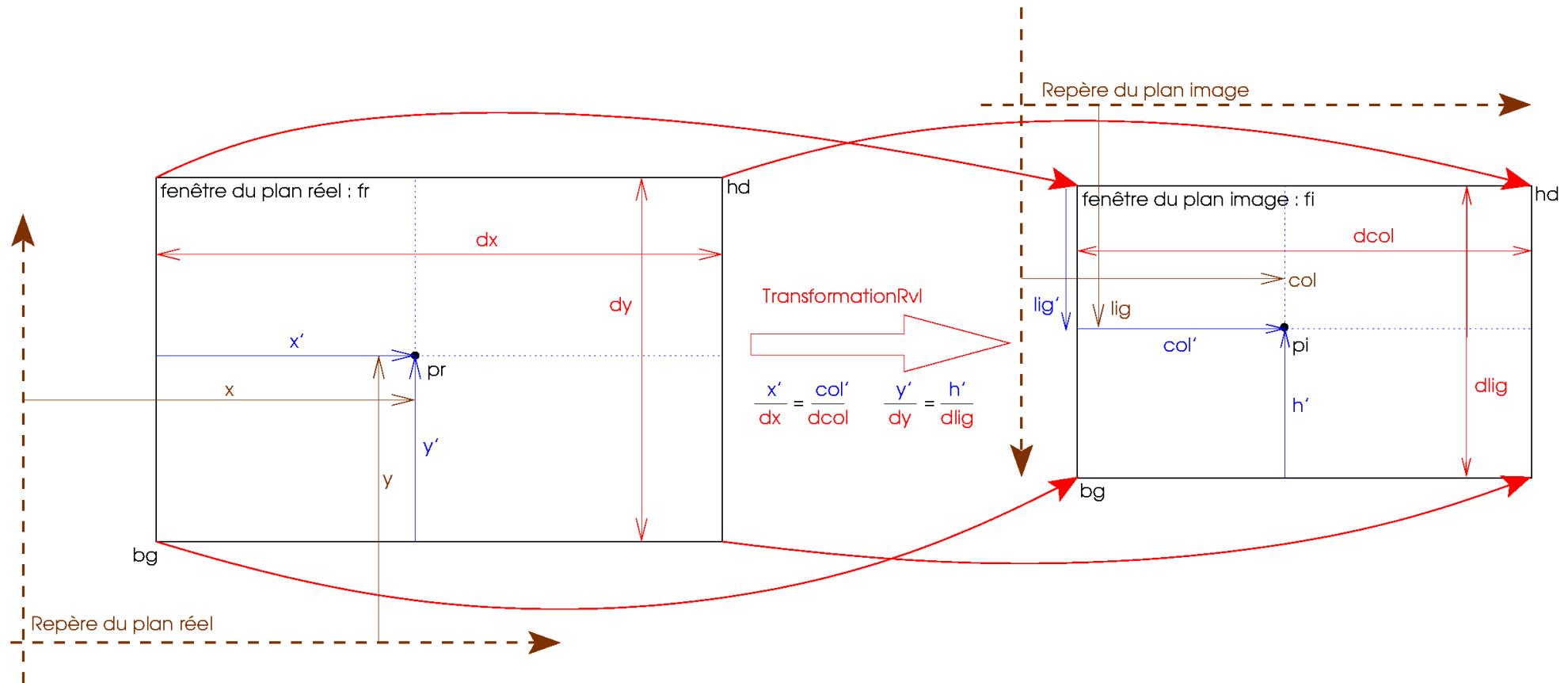
// car axe vertical de haut vers bas

Transformations entre fenêtres

- Positions relatives identiques entre les deux fenêtres



Positions dans les repères globaux



$$x = bg.x + x'$$

$$y = bg.y + y'$$

$$col = bg.col + col'$$

$$lig = bg.lig - h'$$

$$= hd.lig + dlig - h'$$

$$= hd.lig + lig'$$

Axe vertical inversé !!

Transformations entre fenêtres

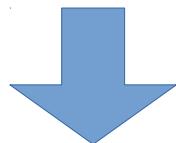
- Pour différencier les repères utilisés, on va noter :
 - pr : le point du plan réel dans le repère R^2
 - pr' : le point du plan réel dans le repère de la fenêtre fr
 - pi : le point du plan image dans le repère N^2
 - pi' : le point du plan image dans le repère de la fenêtre fi
- La transformation $pr \rightarrow pi$ se fait donc en trois étapes :
 - 1) Transfo de pr en pr' : passage du repère de R^2 à celui de fr
 - 2) Transfo de pr' en pi' : passage de fr à fi
 - 3) Transfo de pi' en pi : passage du repère de fi à celui de N^2

$$\begin{array}{ccc} 1 & 2 & 3 \\ pr \rightarrow pr' \rightarrow pi' \rightarrow pi \end{array}$$

Transformations entre fenêtres

- Étapes 1 et 3 et enchaînement des étapes :

- Point pr dans \mathbb{R}^2

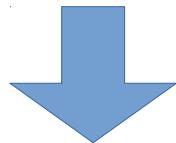


Étape 1

$$pr'.x = pr.x - fr.bg.x$$

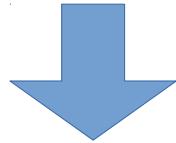
$$pr'.y = pr.y - fr.bg.y$$

- Point pr' dans fr



Étape 2

- Point pi' dans fi



Étape 3

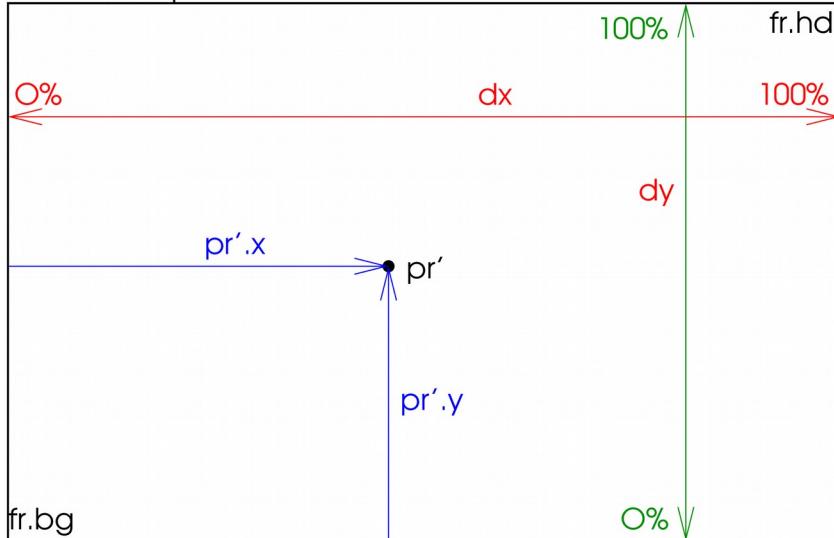
- Point pi dans N^2

$$pi.col = pi'.col + fi.bg.col$$

$$pi.lig = pi'.lig + fi.hd.lig$$

Étape 2 de la transformation

fenêtre du plan réel : fr

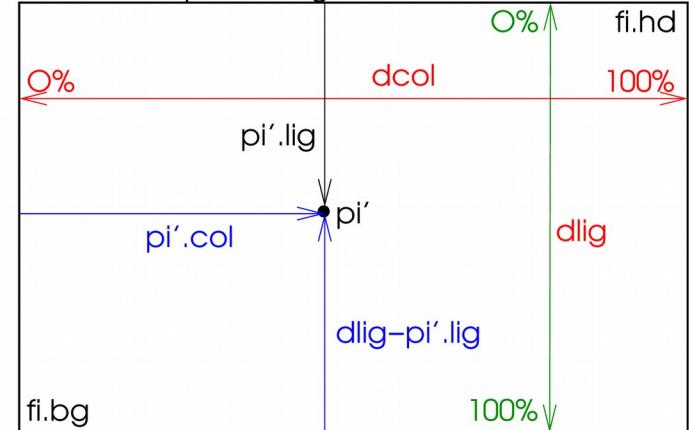


TransformationRvl

$$\frac{pr'.x}{dx} = \frac{pi'.col}{dcol}$$

$$\frac{pr'.y}{dy} = \frac{dlig - pi'.lig}{dlig}$$

fenêtre du plan image : fi



- On déduit les positions relatives (pourcentages) à partir du point bas-gauche (origine de *fr*) :

- De *pr'* dans *fr* :

Horizontale : $pr'.x / dx$ // 0 (=0%) ~ 1 (100%)

Verticale : $pr'.y / dy$ // idem

- De *pi'* dans *fi* :

Horizontale : $pi'.col / dcol$

Verticale : $(dlig - pi'.lig) / dlig$

// !! division réelle nécessaire !!

// soustraction car axe vertical inversé !

Étape 2 et fusion des étapes

- L'identité des positions relatives de pr' et pi' implique :

$$pr'.x / dx = pi'.col / dcol$$

$$pr'.y / dy = (dlig - pi'.lig) / dlig$$
- Et l'on en déduit les coordonnées de pi' :

$$pi'.col = pr'.x * dcol / dx$$

$$pi'.lig = dlig - pr'.y * dlig / dy$$
- Fusion des 3 étapes : pi en fonction de pr

$$pi.col = \text{arrondi}((\text{pr.x} - \text{fr.bg.x}) * \boxed{dcol / dx} + \boxed{\text{fi.bg.col}})$$

1) 2) 3)

$$pi.lig = \text{arrondi}(\boxed{dlig - (\text{pr.y} - \text{fr.bg.y}) * dlig / dy} + \boxed{\text{fi.hd.lig}})$$

2) 1) 3)

Formulation linéaire de la transformation

- On développe les expressions précédentes et on obtient une transformation de la forme :

$$\text{pi.col} = \text{arrondi}(A * \text{pr.x} + B)$$

$$\text{pi.lig} = \text{arrondi}(C * \text{pr.y} + D)$$

$$\text{avec : } A = d\text{col} / dx \quad B = fi.\text{bg.col} - A * fr.\text{bg.x}$$

$$C = -d\text{lig} / dy \quad D = d\text{lig} - C * fr.\text{bg.y} + fi.\text{hd.lig}$$

- Les coefficients A et C représentent respectivement les rapports d'échelles entre :
 - Les axes horizontaux des fenêtres *fr* et *fi*
 - Les axes verticaux des fenêtres *fr* et *fi*
- Les coefficients B et D correspondent à la conjonction des translations des deux fenêtres dans leurs plans respectifs

Structure et fonction de transformation

- Contient les deux fenêtres et les 4 coefficient A, B, C et D
- Nommage : préfixe ri = réel → image

```
TransfosFenetres = structure
    fr : FenetreReel // Fenêtre du plan réel liée aux transformations
    fi : FenetreImage // Fenêtre du plan image liée aux transformations
    riA : réel         // Rapport d'échelle horizontal de fr vers fi
    riB : réel         // Décalage horizontal de fr vers fi
    riC : réel         // Rapport d'échelle vertical de fr vers fi
    riD : réel         // Décalage vertical de fr vers fi
```

- Fonction de transformation :

```
fonction TransformationRvI(pr : PointReel, transfo : TransfosFenetres) : PointImage
// Transforme le point pr en un point image

// Point image résultant de la transformation
pi : PointImage

// Calcul du point pi
pi.col ← arrondi(transfo.riA * pr.x + transfo.riB)
pi.lig ← arrondi(transfo.riC * pr.y + transfo.riD)

// Renvoie le point image obtenu par la transformation du point réel
retourne pi
```

Version matricielle

- Étant donné que les transformations entre les fenêtres sont linéaires, on peut les exprimer sous la forme d'une matrice T_{ri} :

$$pi = \text{arrondi}(T_{ri} * pr) = \begin{pmatrix} pi.col \\ pi.lig \end{pmatrix} = \left[\begin{pmatrix} A & 0 & B \\ 0 & C & D \end{pmatrix} \times \begin{pmatrix} pr.x \\ pr.y \\ 1 \end{pmatrix} \right]$$

- Coordonnées homogènes :
 - Ajout à pr d'une 3^{ème} composante (=1) pour exprimer les translations sous forme matricielle
 - On peut aussi ajouter une 3^{ème} ligne à T_{ri} et à pi pour exprimer des transformations plus complexes

Transformation de *pi* en *pr*

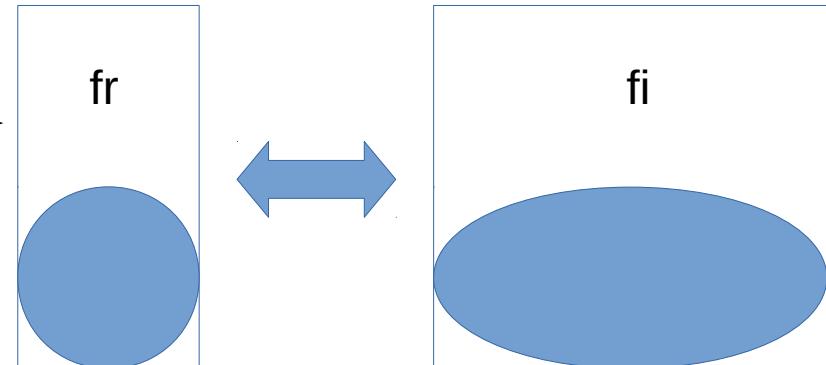
- Une fois que l'on a déterminé la transformation dans un sens, on peut facilement déduire sa réciproque, dans l'autre sens
- Pour déduire la transformation de *pi* vers *pr*, il suffit d'inverser les formules de *pr* vers *pi* en isolant les variables *x* et *y* :
$$riA * x + riB = col \Rightarrow x = (col - riB) / riA$$
$$riC * x + riD = lig \Rightarrow y = (lig - riD) / riC$$
- Ce qui permet de déduire les coefficients *irA*, *irB*, *irC* et *irD* de la nouvelle transformation (*ir* = image → réel, *ri* = réel → image) :
$$x = irA * col + irB \Rightarrow irA = 1 / riA \text{ et } irB = -riB / riA$$
$$y = irC * lig + irD \Rightarrow irC = 1 / riC \text{ et } irD = -riD / riC$$
- Enfin, *x* et *y* étant réels, l'arrondi à la fin du calcul n'est pas pertinent dans ce sens là

Proportions des fenêtres

- Lorsque les proportions sont différentes :

$$|dy/dx| \neq |dlig / dcol|$$

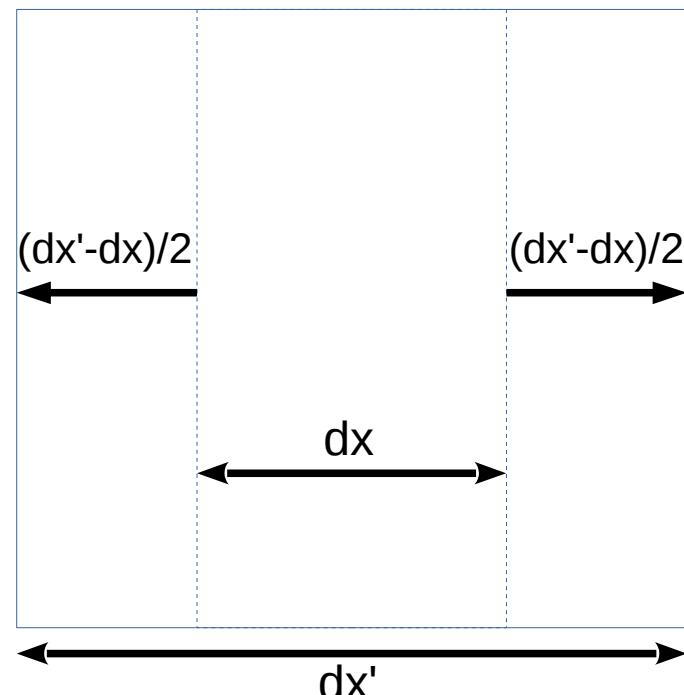
les objets sont déformés...



- Il faut alors modifier la taille de l'une des fenêtres pour retrouver les bonnes proportions
- 4 possibilités de changement, selon que l'on modifie dx , dy , $dcol$ ou $dlig$ afin de rétablir l'égalité des proportions
- Selon la modification effectuée, on obtient :
 - Un agrandissement de la largeur (ou hauteur) de l'une des fenêtres :
 - Toujours possible pour fr MAIS pas pour fi (surface d'affichage peut être limitée)
 - Une réduction de la largeur (ou hauteur) de l'une des fenêtres :
 - Toujours possible pour chaque fenêtre MAIS pas toujours souhaitable (moins d'informations affichées car zone visualisée plus petite)

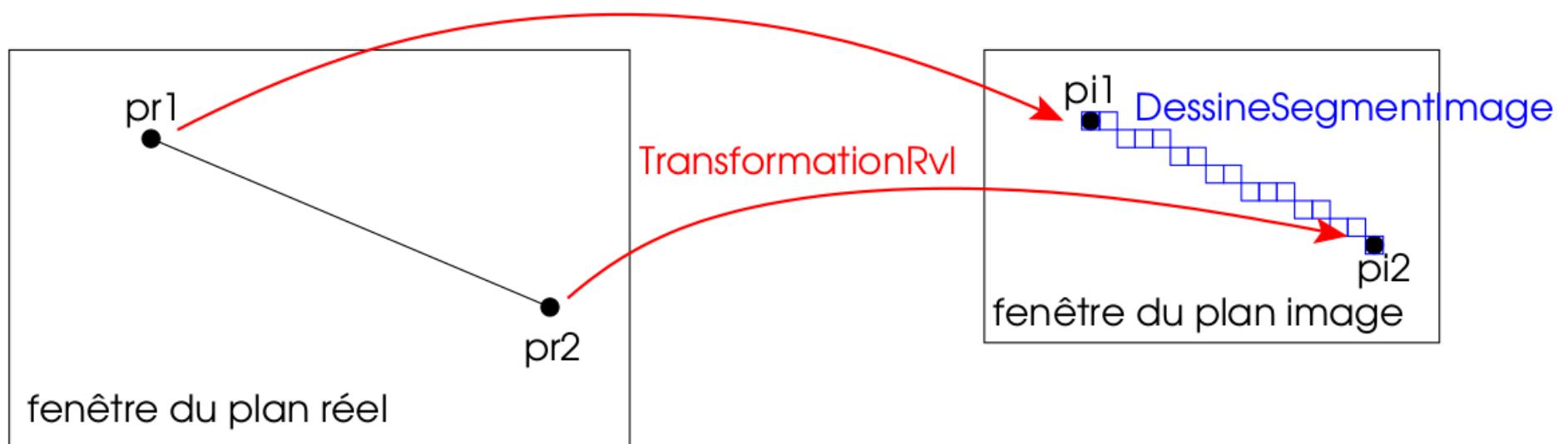
Proportions des fenêtres

- Il est alors préférable de toujours privilégier l'agrandissement de *fr* :
 - $dy/dx > dlig/dcol \rightarrow$ augmenter la largeur de *fr*
 - dx remplacé par $dx' = dy * dcol / dlig$
 - $dy/dx < dlig/dcol \rightarrow$ augmenter la hauteur de *fr*
 - dy remplacé par $dy' = dx * dlig / dcol$
- En général, on positionne la nouvelle fenêtre de sorte que l'ancienne soit au centre de la nouvelle
- Agrandissement horizontale :
 - $fr.bg.x' = fr.bg.x - (dx'-dx)/2$
 - $fr.hd.x' = fr.hd.x + (dx'-dx)/2$
- Symétrie en *y* pour l'agrandissement vertical



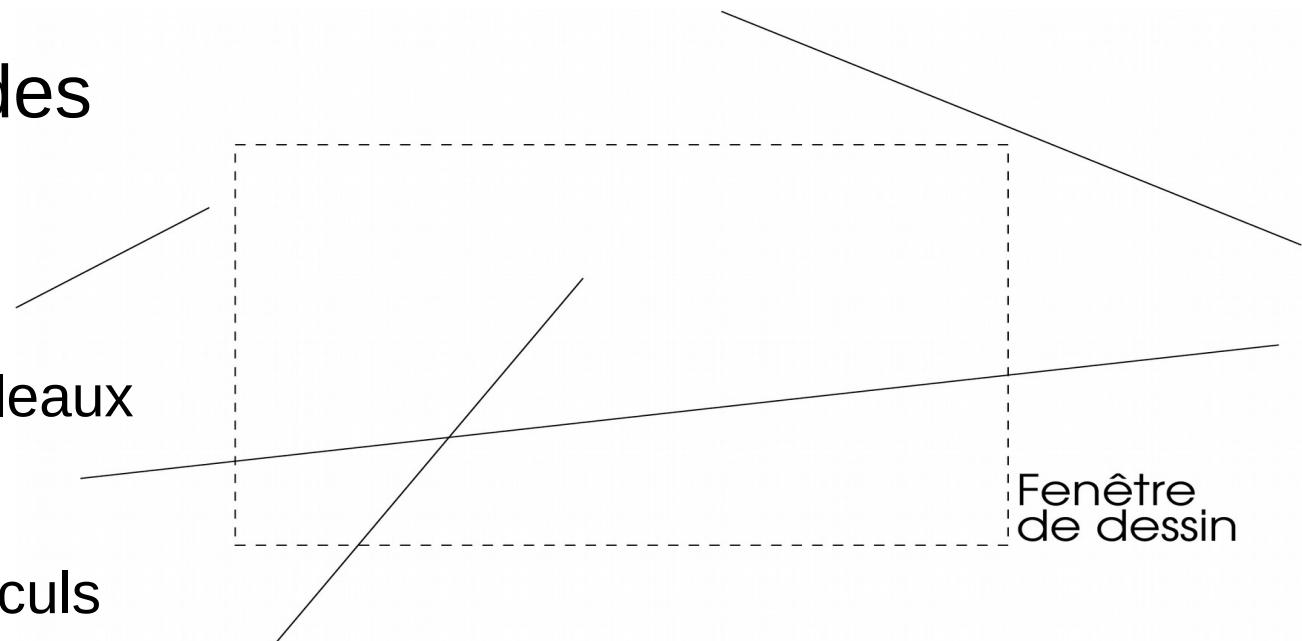
Tracé d'un segment du plan réel

- Transformation des deux extrémités
- Tracé du segment image obtenu



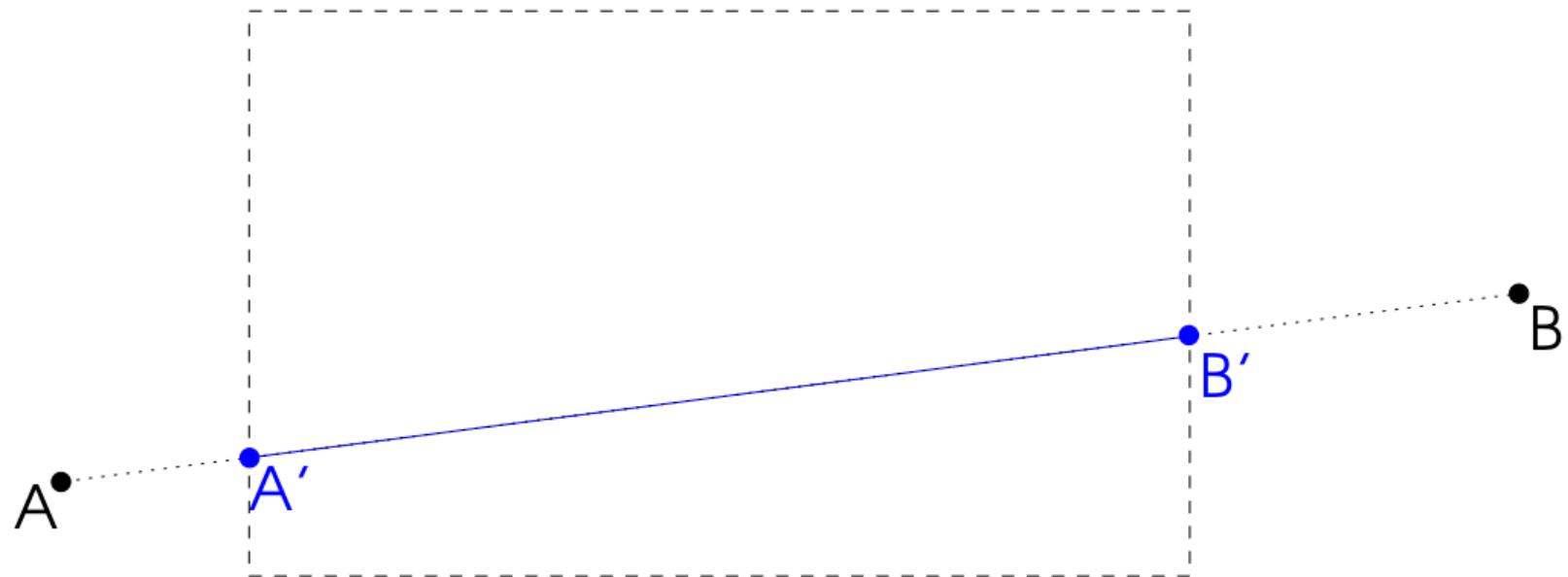
Découpage de segment

- Le problème lors du dessin d'un segment réel est qu'il peut ne pas être inclus totalement dans la fenêtre de dessin
- Cela peut poser des problèmes de :
 - Validité
 - Accès hors tableaux
 - Performance
 - Parcours et calculs inutiles
- Nécessité de découper les segments !



Principe du découpage

- Transformer le segment initial en un autre segment dont les extrémités sont dans la fenêtre
 - Si le segment est hors fenêtre, les extrémités doivent l'être aussi pour détecter qu'il ne doit pas être affiché



- On passe de $[AB]$ à $[A'B']$

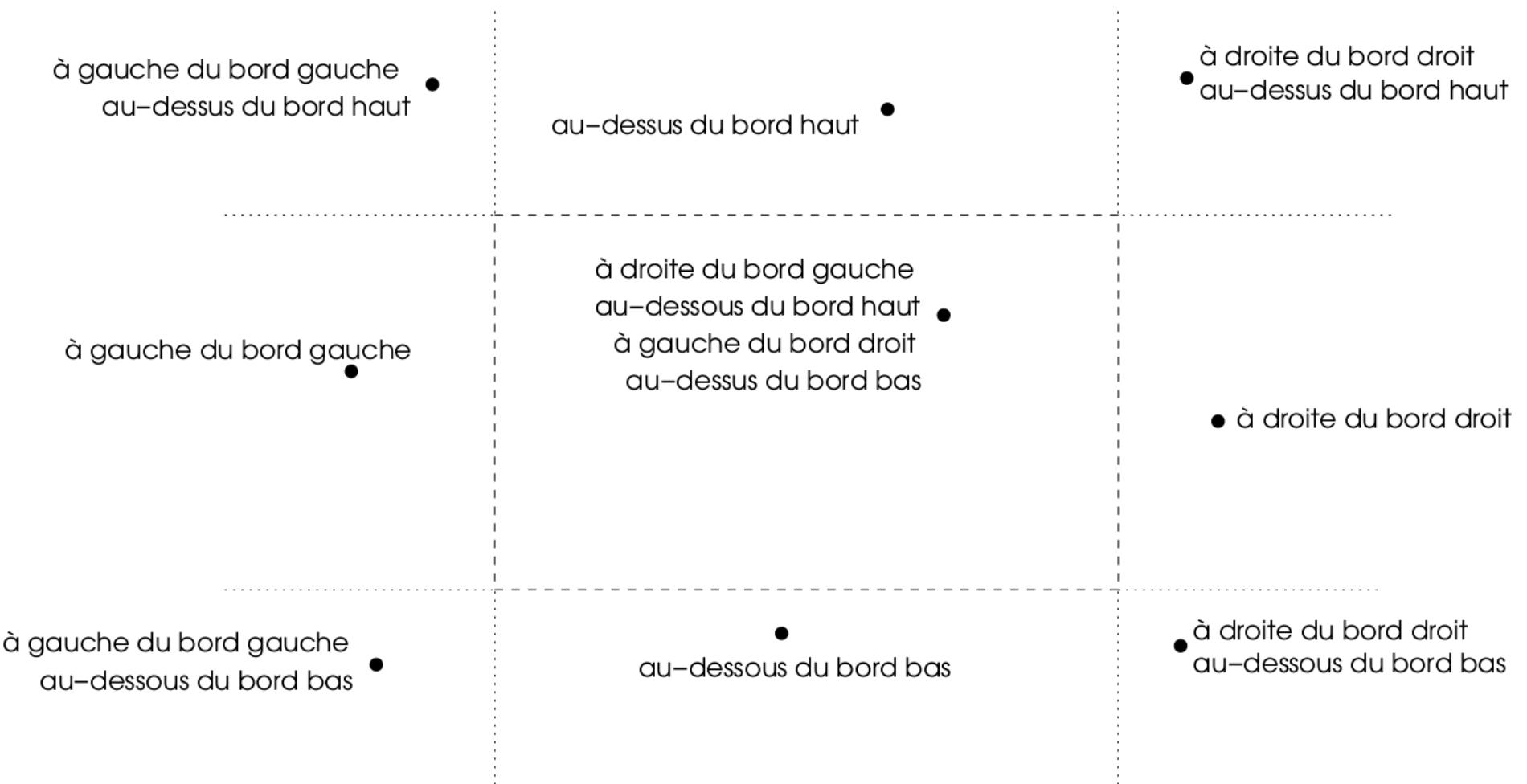
Algorithme général

- Le schéma général de l'algorithme est le suivant :

Pour chaque extrémité du segment :

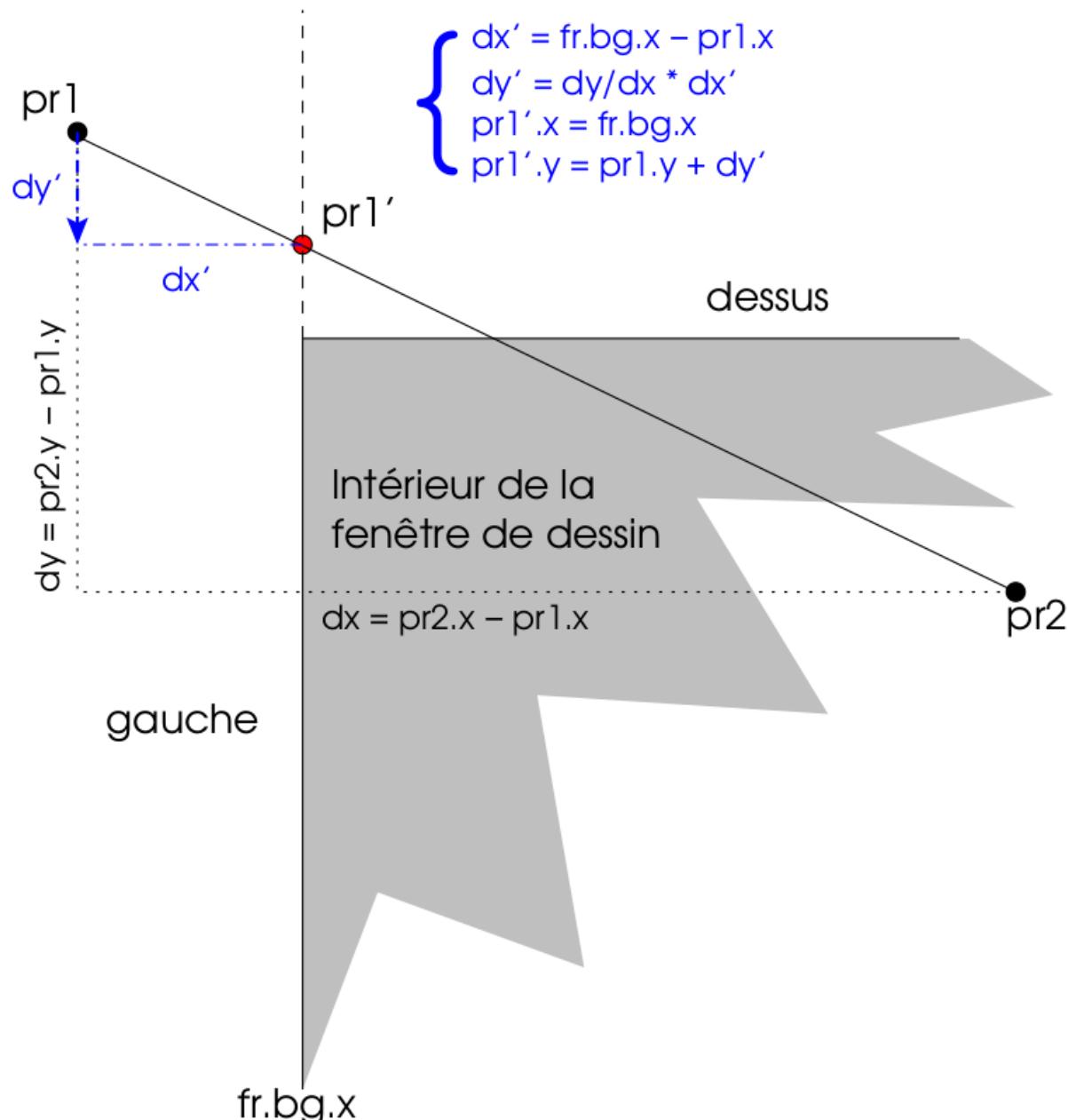
- Pour chaque bord de la fenêtre :
 - Si l'extrémité est à l'extérieur par rapport à ce bord alors :
 - remplacer l'extrémité par l'intersection du segment avec ce bord de la fenêtre

Test de la position d'un point



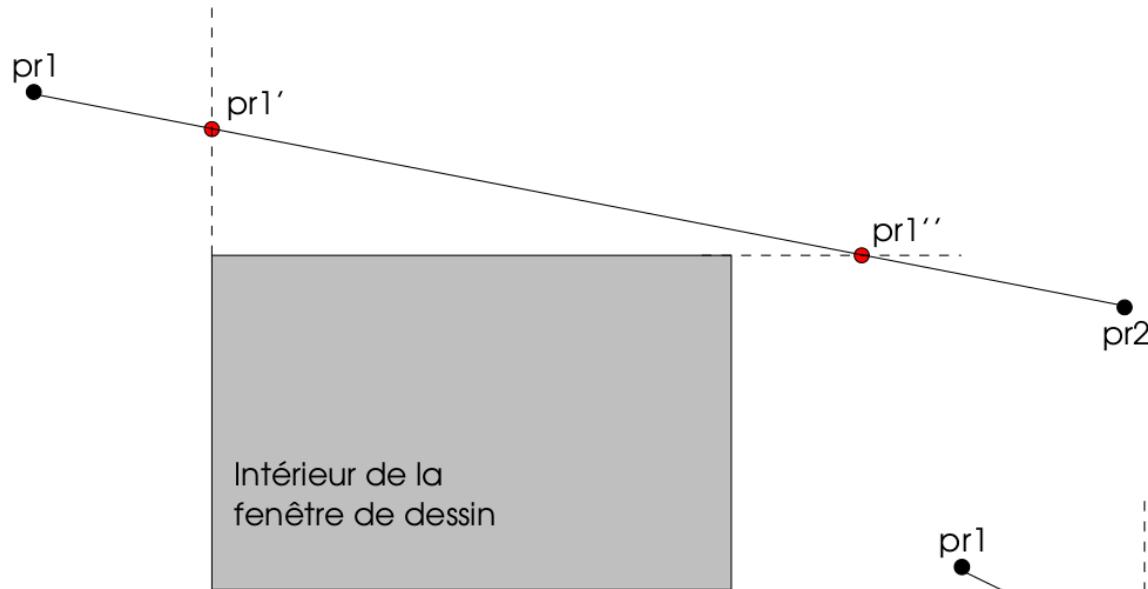
Projection d'un point sur un bord

- $pr1 \rightarrow pr1'$

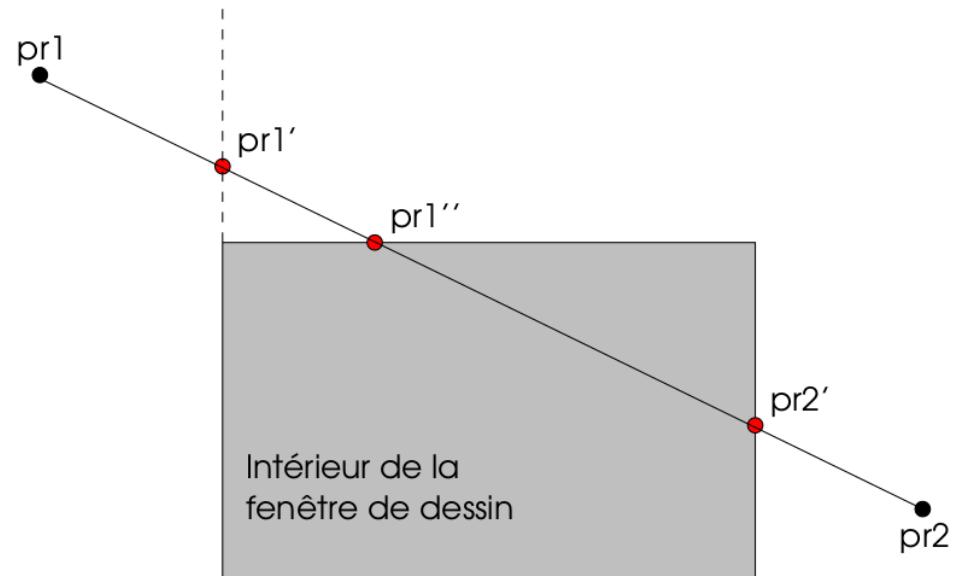


Processus de projection

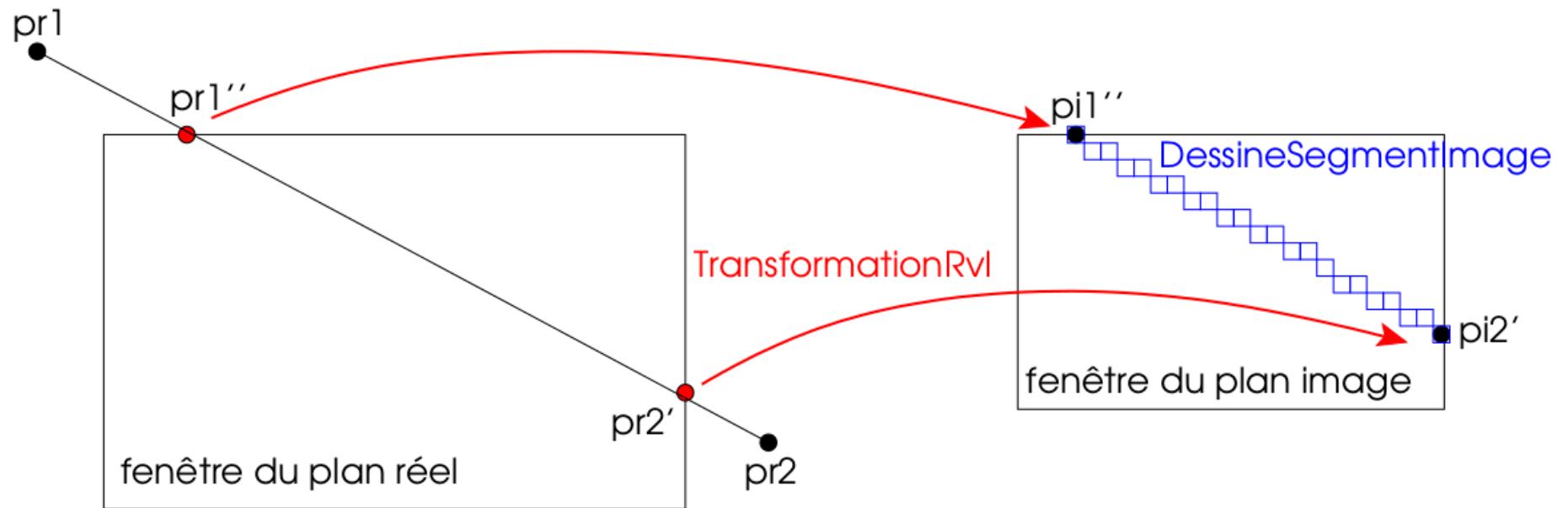
- Projections sans intersection avec la fenêtre



- Projections avec intersections

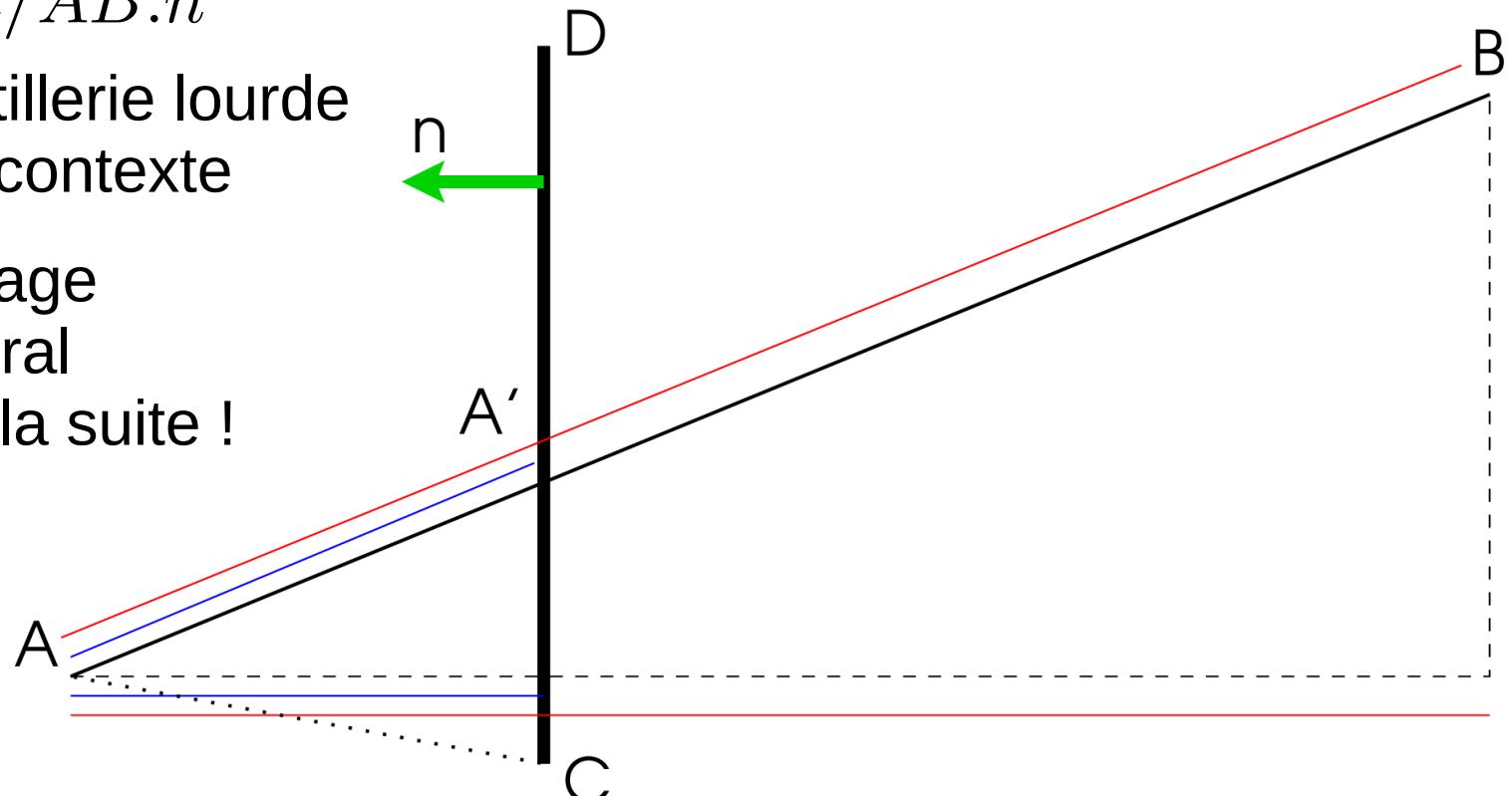


Processus complet réel → image



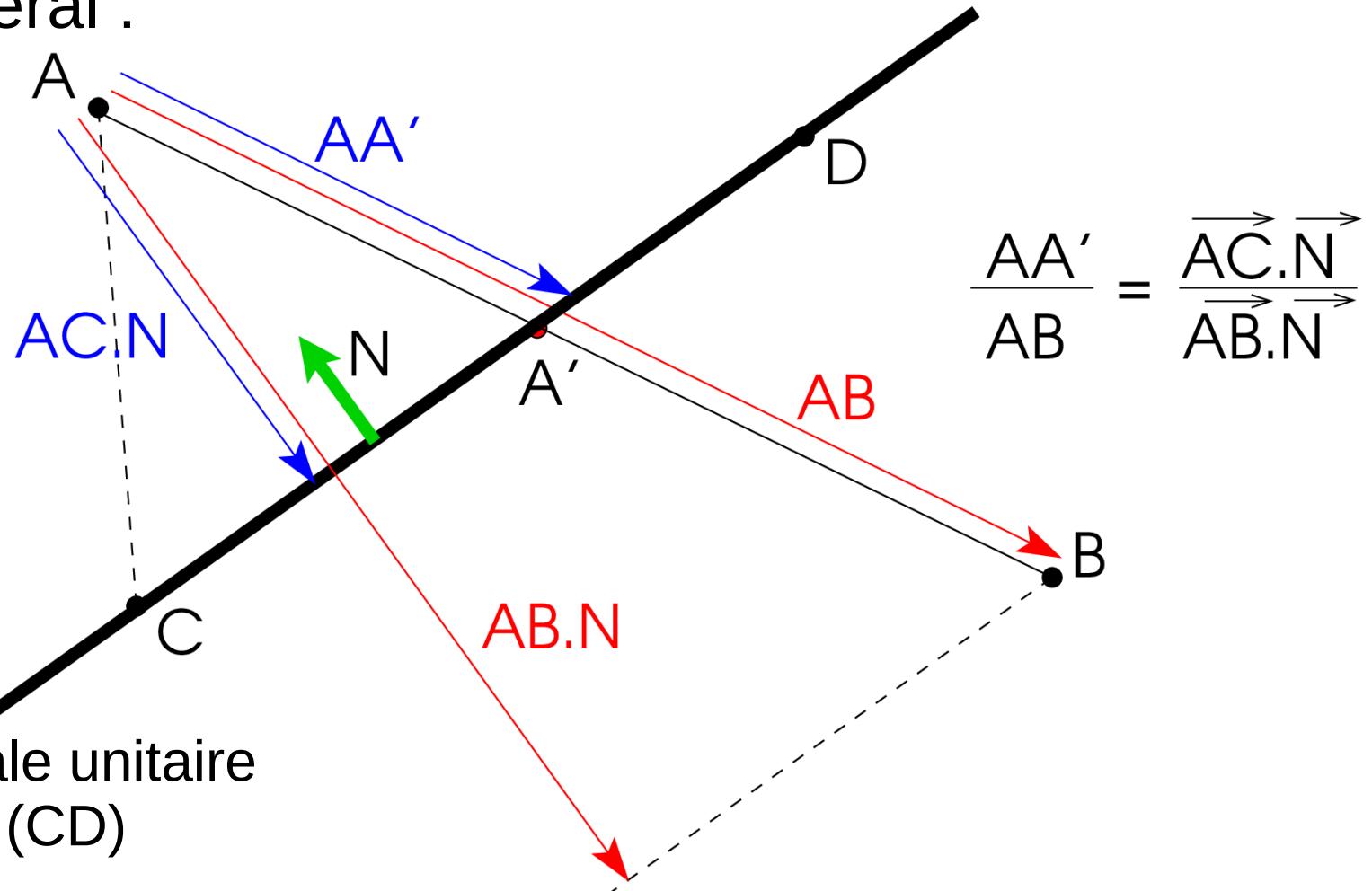
Projection sur un bord

- Version plus générique : approche vectorielle
- Formulation vectorielle de A' :
$$A' = A + \alpha \cdot \vec{AB} \text{ avec } 0 \leq \alpha \leq 1$$
- α calculé par Thalès : avec \vec{n} normale à $[CD]$
$$\alpha = \vec{AC} \cdot \vec{n} / \vec{AB} \cdot \vec{n}$$
- Un peu l'artillerie lourde pour notre contexte
- Mais avantage d'être général et utile par la suite !



Version vectorielle de la projection

- Cas général :



- N = normale unitaire
à la droite (CD)
- Produit scalaire avec N
→ projection sur direction N

Prise en compte du découpage

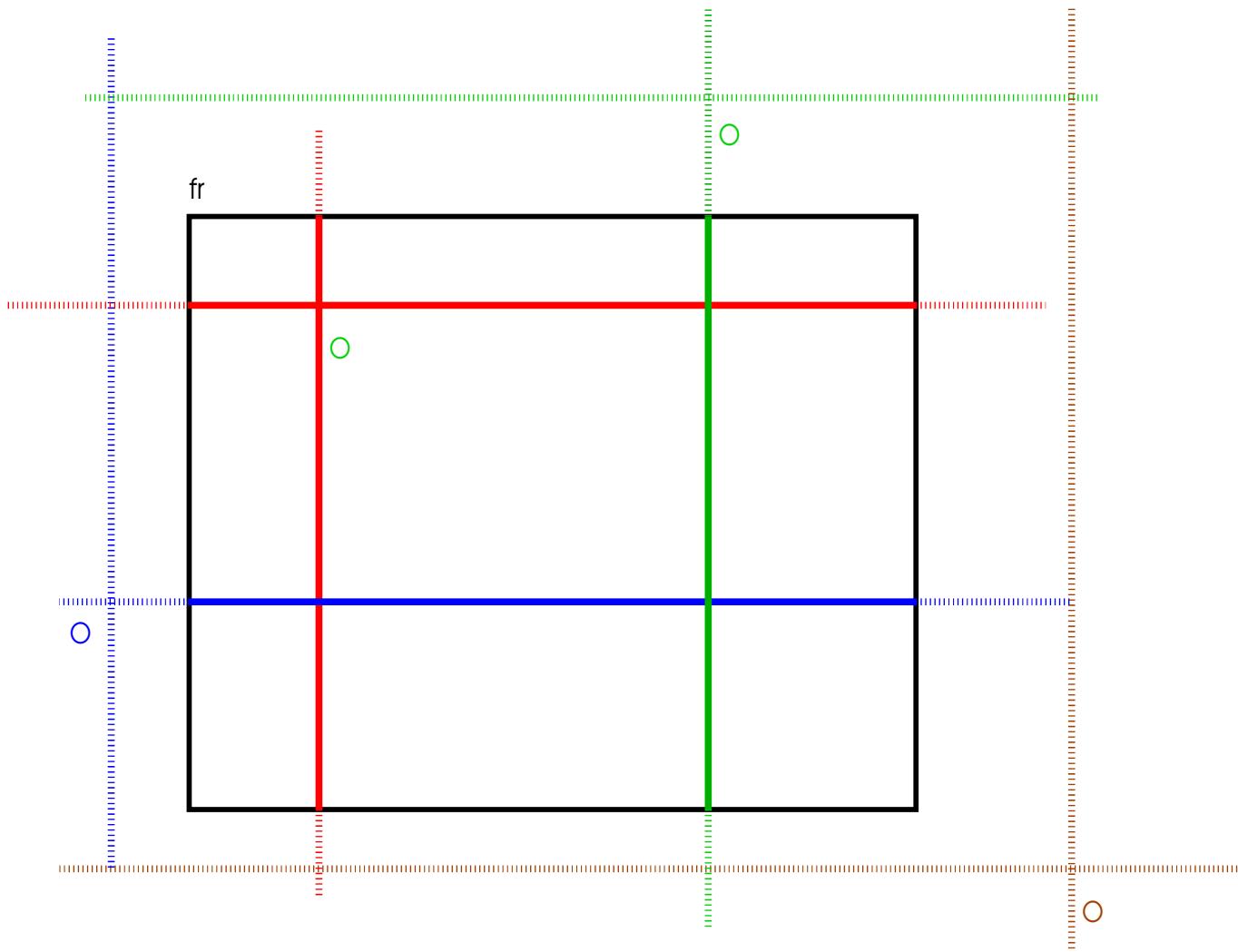
- Intégrer le découpage du segment dans la fonction de dessin d'un segment réel
- Comment détecter que le segment n'a pas d'intersection avec la fenêtre image ?
 - Renvoi d'un booléen en plus des deux points
OU
 - Renvoi de points à gauche de la fenêtre réelle
 - Permet de détecter qu'il n'y a rien à afficher
 - L'abscisse du premier point suffit

Tracé des axes du repère

- Deux segments :
 - Axe des abscisses : horizontal et d'ordonnée 0
 - Axe des ordonnées : vertical et d'abscisse 0
- Pour chaque axe, il faut :
 - Déterminer l'intervalle à afficher
 - Dépend de la fenêtre
- Traitement dans le plan réel utilise le découpage de segment
 - Élimine les cas où les axes sont hors fenêtre

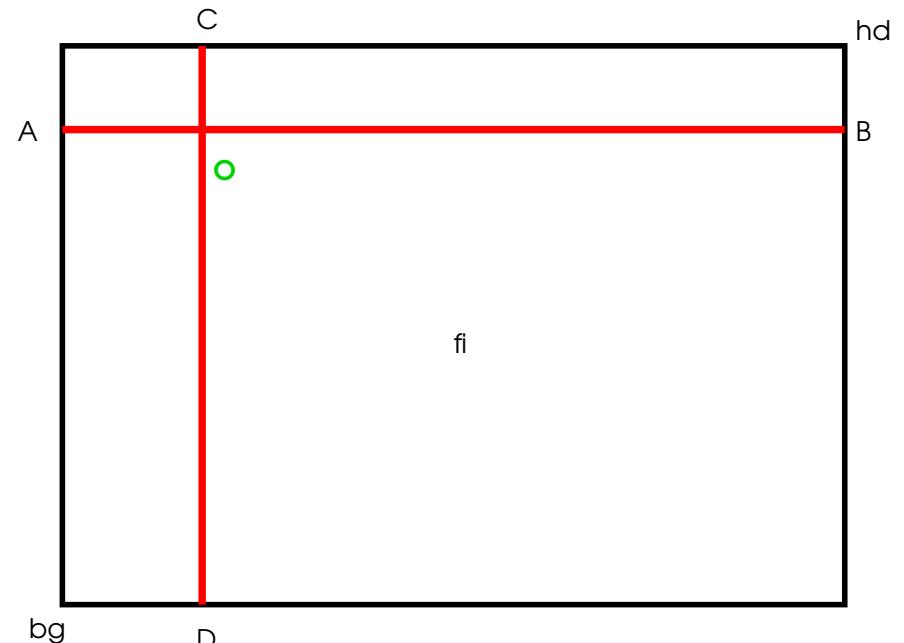
Tracé des axes du repère

- Deux segments (horizontal et vertical)
 - Position liée à la fenêtre du plan réel (origine O)



Tracé des axes du repère

- Autre solution :
 - Projection dans le plan image de l'origine du repère réel
 - Donne directement la ligne et la colonne des deux axes
 - Tracer les deux segments image dans l'intervalle de fi
 - Pour chaque axe, vérifier que le point est dans l'intervalle de fi
 - Utiliser les sommets de fi pour déduire les extrémités du segment sur l'autre dimension

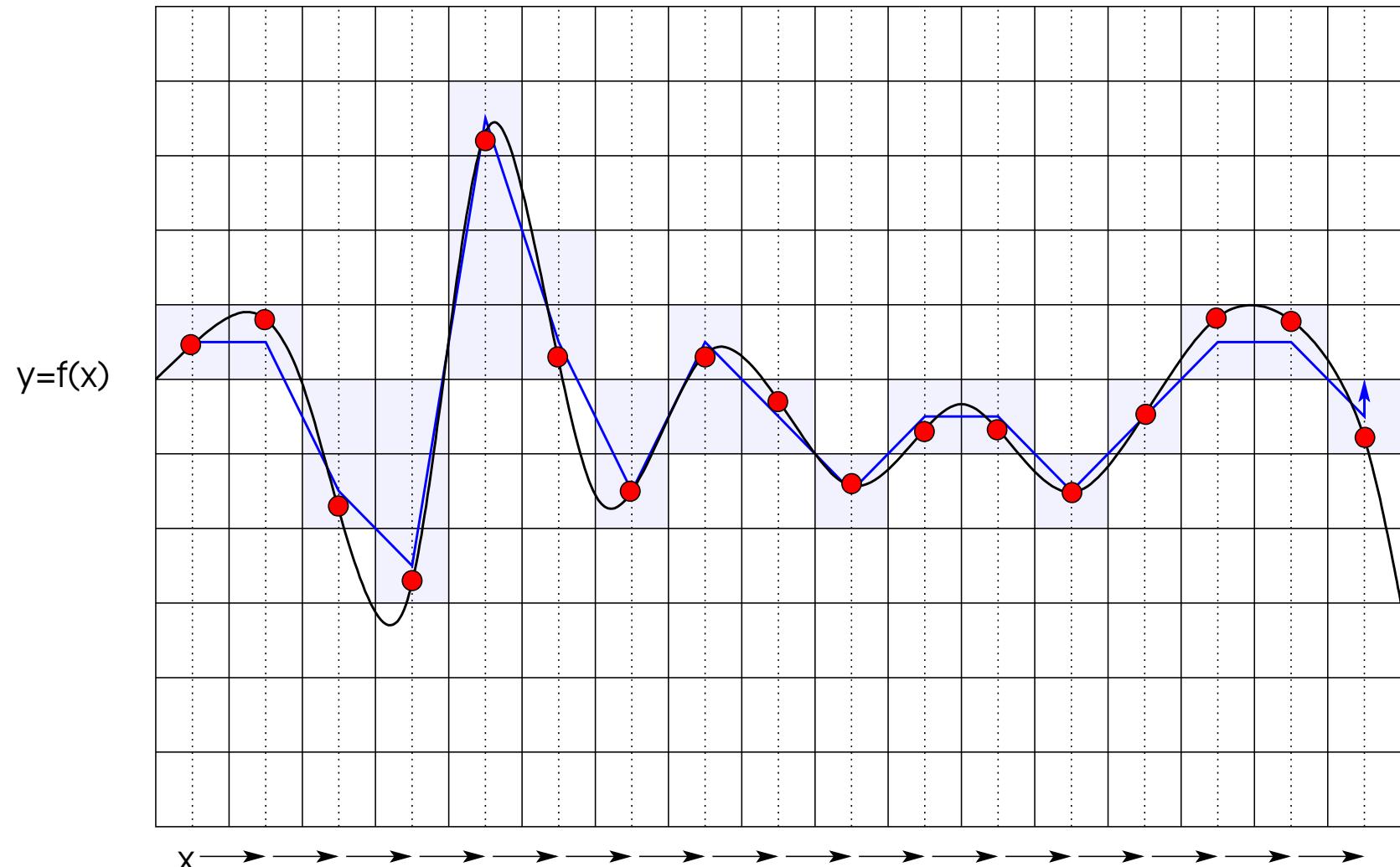


Tracé des graduations

- Il faut définir l'intervalle de celles-ci :
 - Dans le plan réel
- Tracé selon chaque axe :
 - On part de l'origine
 - On s'arrête lorsque l'on sort de la fenêtre réelle
 - On projette le point dans l'image :
 - On obtient les coordonnées d'un pixel sur cet axe
 - On en déduit les extrémités de la graduation
 - On passe d'une graduation à la suivante en se déplaçant de l'intervalle défini

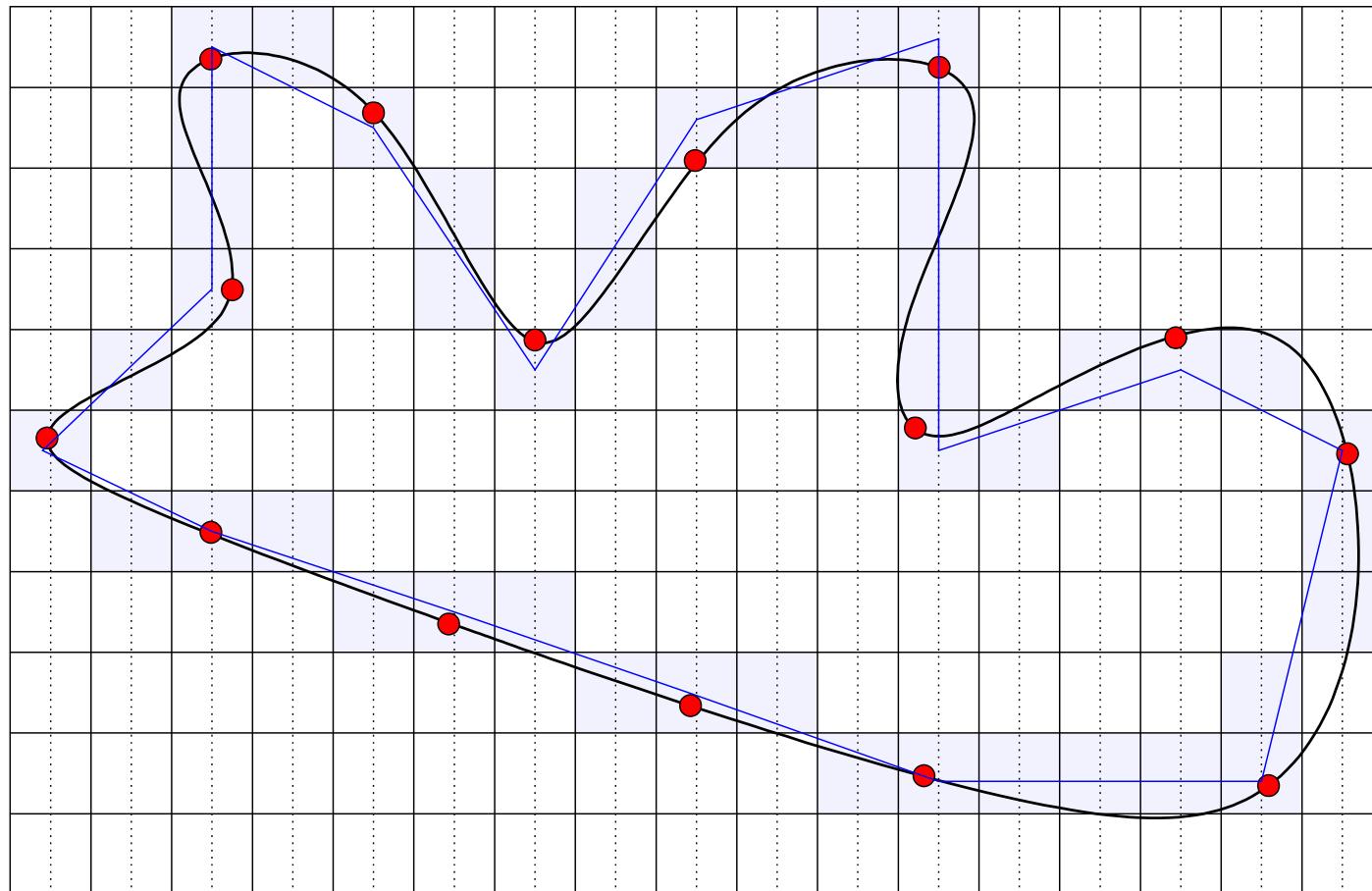
Tracé de fonction

- Parcours des abscisses discrets (pixels)



Tracé de courbe paramétrique

- Parcours du paramètre t



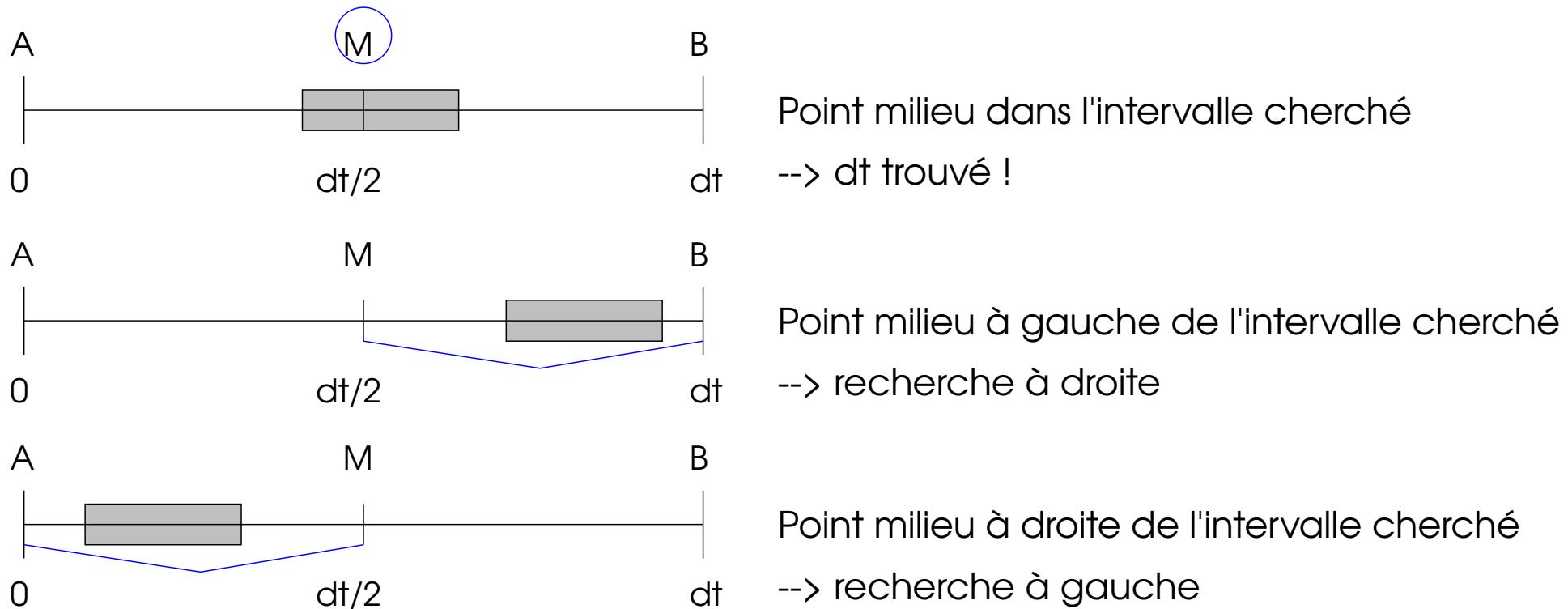
$$P(t)=(x(t), y(t))$$

Pas variable

- Problème :
 - Trouver dt tel que pour $A=P(t)$ et $B= P(t+dt)$, on a :
 - $d(A,B)$ ne doit pas dépasser la taille d'un pixel dans fr
 - $d(A,B)$ ne doit pas être inférieure à $1-\varepsilon$ fois un pixel
 - Taille d'un pixel dans fr déduite des rapports d'échelles dans la transformation des fenêtres :
 - $irA = 1/riA$ et $irC = 1/riC$
 - Pour être général, on prend le min des deux :
 - $d_p = \min(irA, irC)$
 - On cherche donc dt tel que :
$$(1-\varepsilon).d_p \leq d(P(t),P(t+dt)) \leq d_p$$

Processus dichotomique

- Itérations en changeant dt selon que l'on obtient une distance trop grande ou trop petite



Version récursive à pas variable

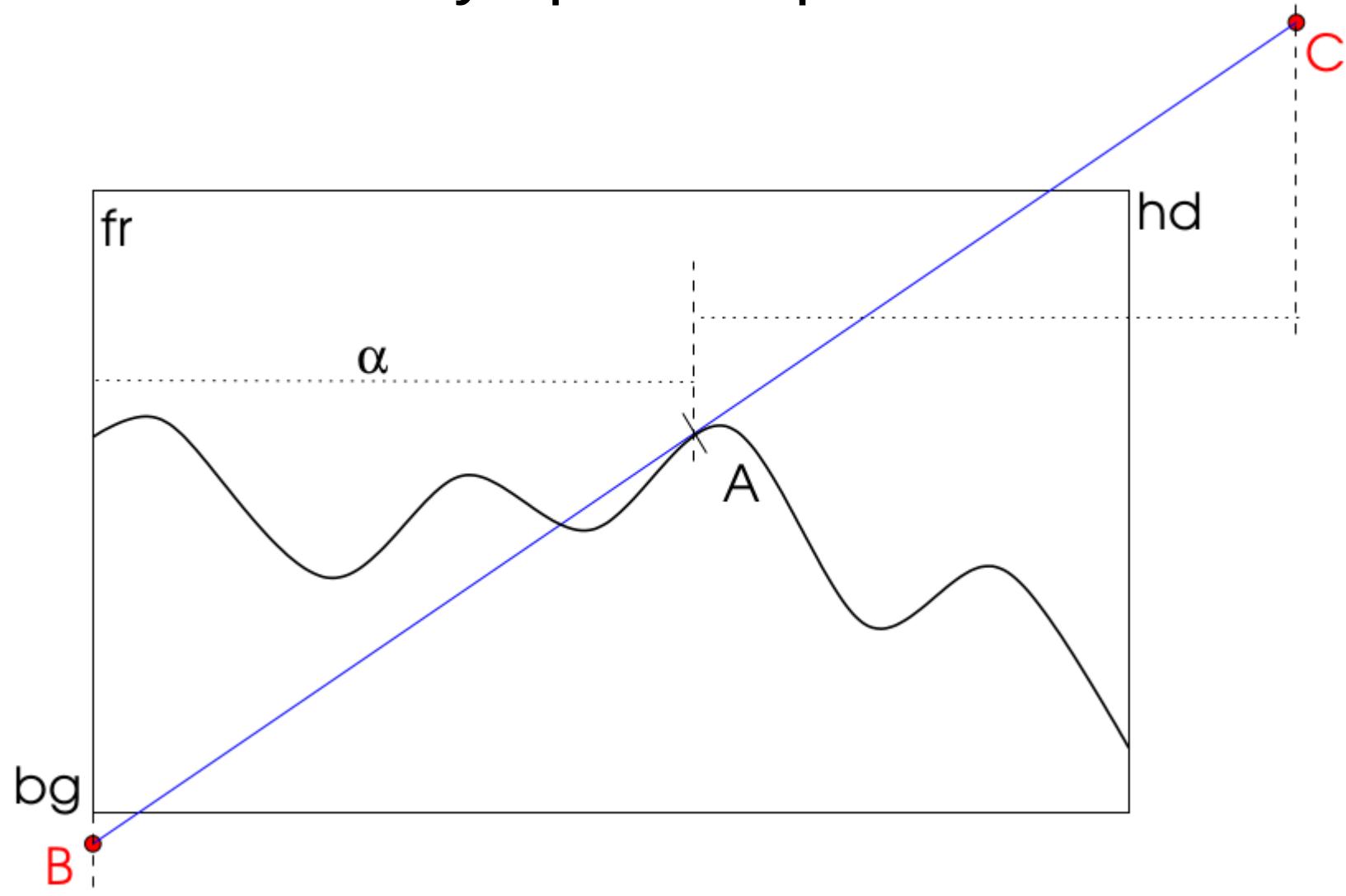
- Calculer dt à chaque fois est coûteux
- On décompose le problème initial en deux sous-problèmes → récursion :
 - Tracer $C(t_{\min}, t_{\max}) = \text{Tracer } C(t_{\min}, t_{\text{milieu}}) + \text{Tracer } C(t_{\text{milieu}}, t_{\max})$
 - On arrête la récursion lorsque :
 - Les deux points extrêmes sont à une distance < pixel
 - Évite le recours au tracé de segment
 - **!! Attention !!** risque de rater des parties de la courbe
 - Exemple du cercle !
 - Critère supplémentaire : seuil sur dt ou distances avec le point au milieu de l'intervalle

Objets paramétriques utiles

- Segment :
 - Défini entre 2 points P_0, P_1 sur l'intervalle $[0:1]$ par :
 - $P(t) = (1-t).P_0 + t.P_1$
- Cercle :
 - Défini par un centre C et un rayon r sur l'intervalle $[0:2\pi]$:
 - $P(t) = C + r.(\cos(t), \sin(t))$
- Ellipse parallèle aux axes du repère :
 - Définie par un centre C et 2 rayons rx, ry sur l'intervalle $[0:2\pi]$:
 - $P(t) = C + (rx.\cos(t), ry.\sin(t))$

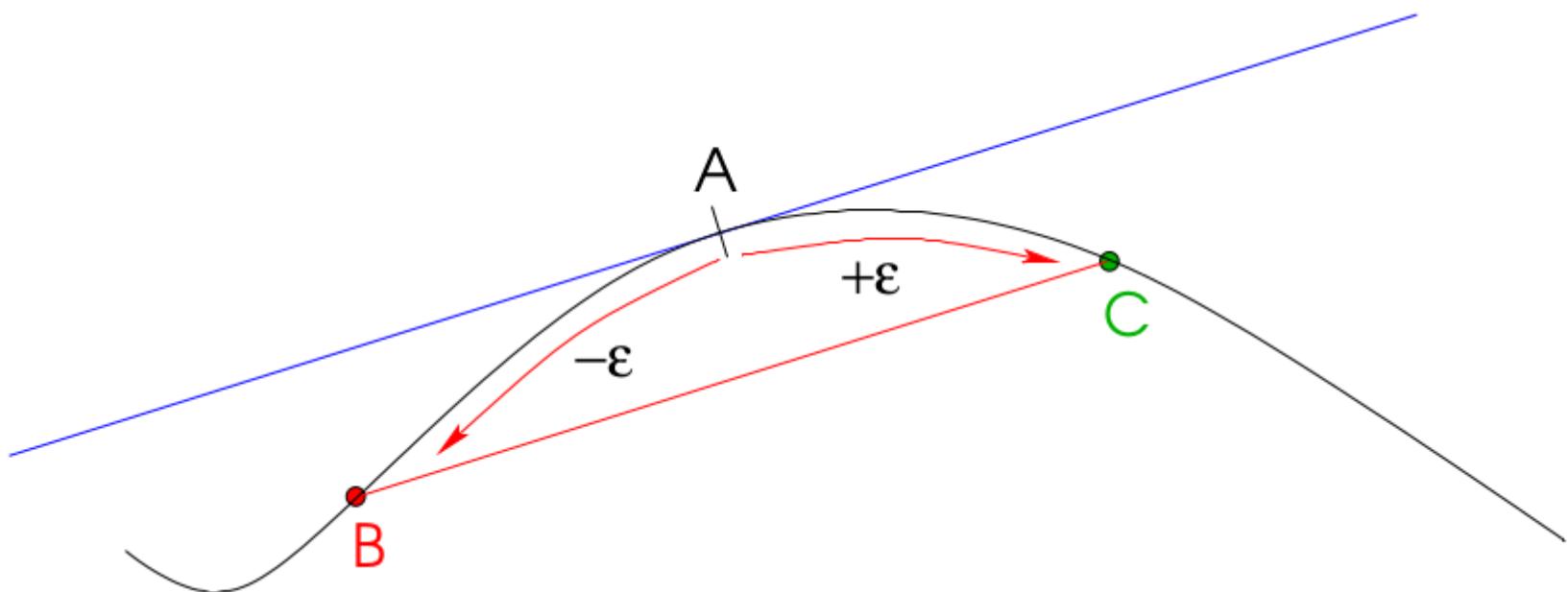
Calcul de la tangente

- Construction analytique des points B et C



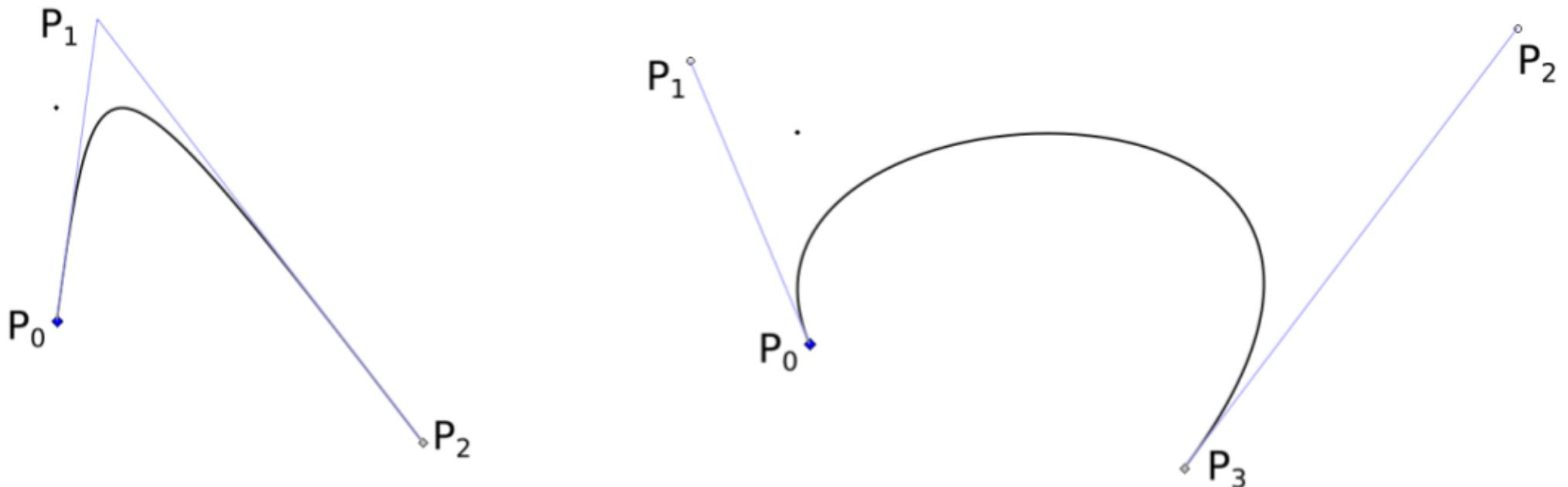
Calcul de la tangente

- Version différentielle :



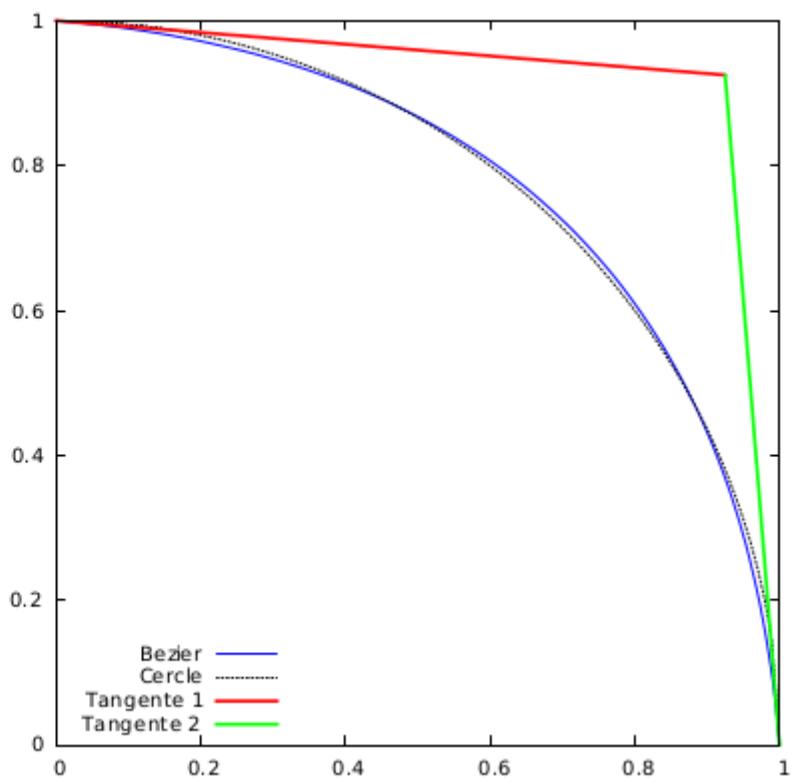
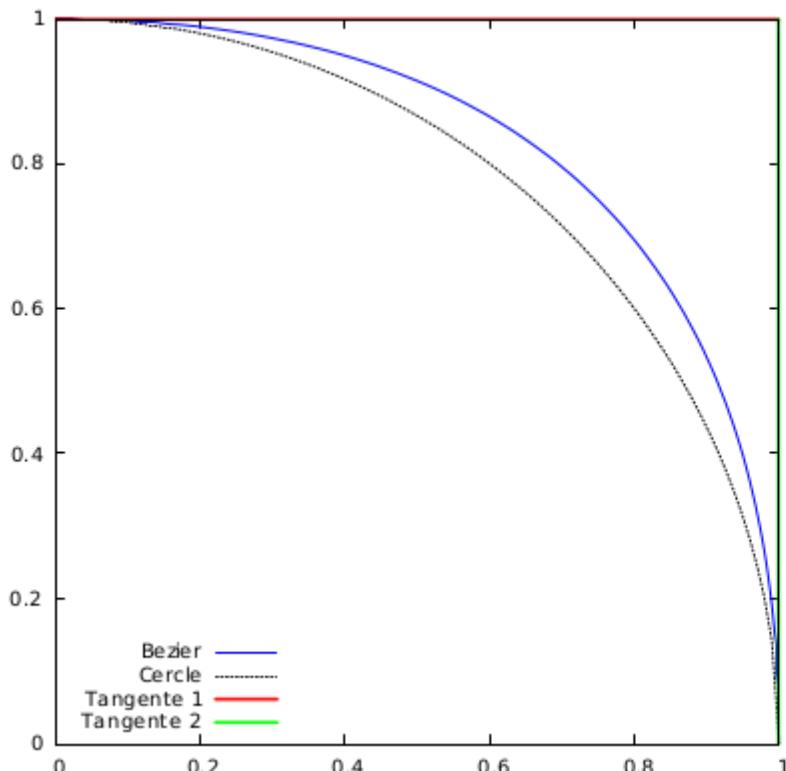
Courbes de Bézier

- Linéaire (degré 1) : $B(t) = (1 - t).P_0 + t.P_1 \quad \forall t \in [0, 1]$
- Quadratique (degré 2) :
$$B(t) = (1 - t)^2.P_0 + 2.(1 - t).t.P_1 + t^2.P_2 \quad \forall t \in [0, 1]$$
- Cubique (degré 3) :
$$B(t) = (1 - t)^3.P_0 + 3.(1 - t)^2.t.P_1 + 3.(1 - t).t^2.P_2 + t^3.P_3 \quad \forall t \in [0, 1]$$



Différence avec cercle

- Pas d'équivalence de courbure



Formulation générale

- Degré n : n+1 points de contrôle
- Courbe exprimée par combinaison polynomiale des points de contrôle :

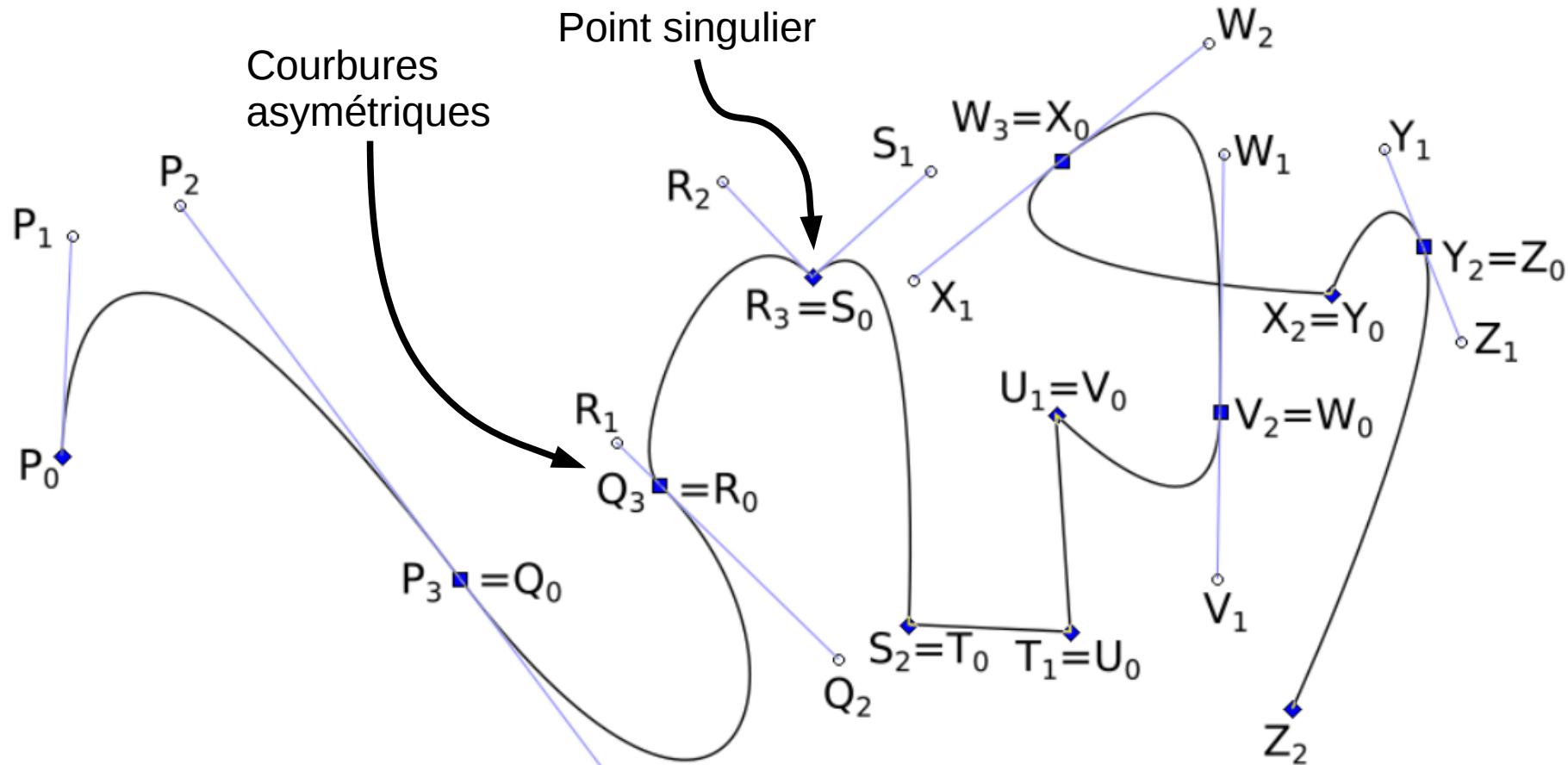
$$B(t) = \sum_{i=0}^n \alpha_i.P_i \quad \forall i \in [0, n]$$

- où : $\alpha_i = C_n^i \cdot (1-t)^{n-i} \cdot t^i \quad \forall i \in [0, n]$
- Rarement plus que le degré 3 :
 - propriétés de contrôle moins flexibles que l'assemblage de courbes de degrés ≤ 3

Assemblage de courbes

- Formes complexes :
 - Faire coïncider le dernier point d'une courbe avec le premier point de la courbe suivante
 - Pas de contrainte sur les degrés des courbes :
 - Relier des courbes de degrés différents
 - Contrôle du lissage aux points d'assemblage :
 - Tangentes alignées en sens opposées
 - Courbure symétrique si longueur identiques
 - Point de rebroussement si tangentes non alignées ou dans le même sens
- Mais pas *toutes* les formes :
 - B-Splines, NURBS,...

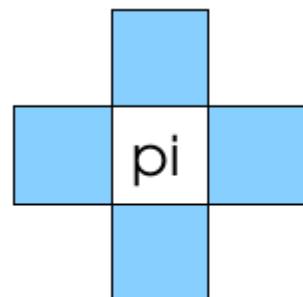
Assemblage de courbes



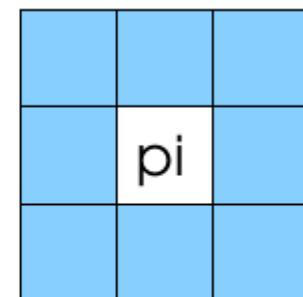
P : degré 3 Q : degré 3 (le point $P_3 = Q_0$ est C^1)
 R : degré 3 S : degré 2 (point non lisse $R_3 = S_0$)
 T : degré 1 U : degré 1
 V : degré 2 W : degré 3
 X : degré 2 Y : degré 2
 Z : degré 2

Remplissage de zone

- Données :
 - Une position de départ (pixel)
 - Une couleur de remplissage
- Résultat :
 - Colorier tous les pixels dans le **voisinage** du pixel de départ qui ont la même couleur que lui et ainsi de suite
- Voisinage :
 - Connexité = définition des *voisins directs*
 - Deux principaux types :



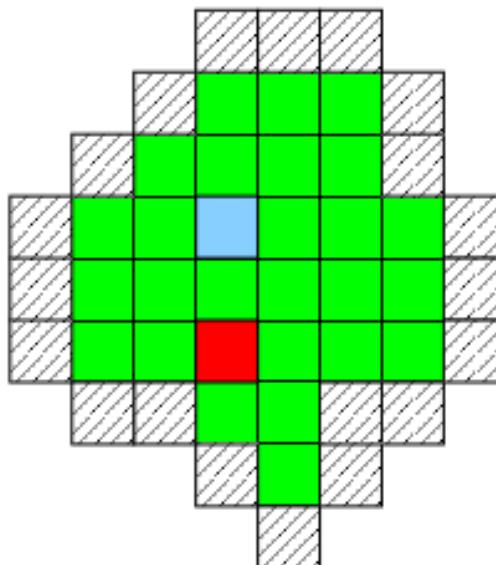
4-voisinage



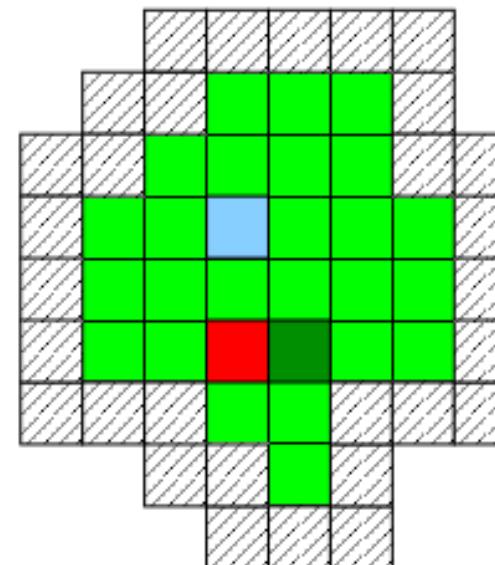
8-voisinage

Zones uniformes

- Composante connexe :
 - Ensemble de pixels de même couleur reliés entre eux selon un voisinage spécifié (4 ou 8)
 - Délimitée par contour de connexité complémentaire
 - Peut englober (encercler) d'autres pixels



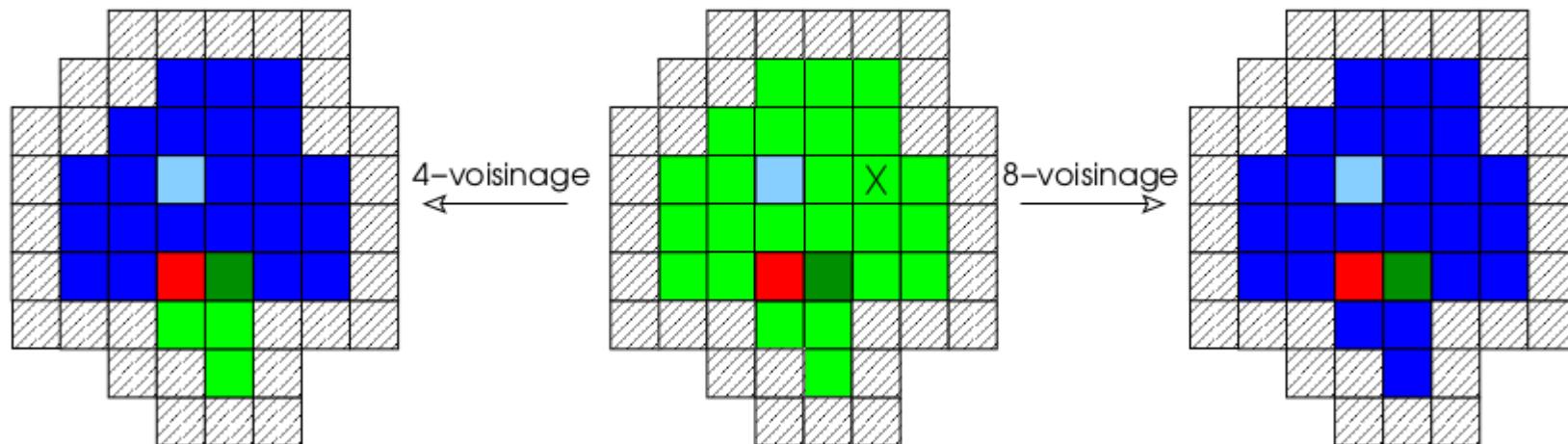
Zone verte 4-connectée délimitée
par un contour 8-connecté



Zone verte 8-connectée délimitée
par un contour 4-connecté

Remplissages

- Remplissage en bleu à partir de X selon les deux types de connexité :



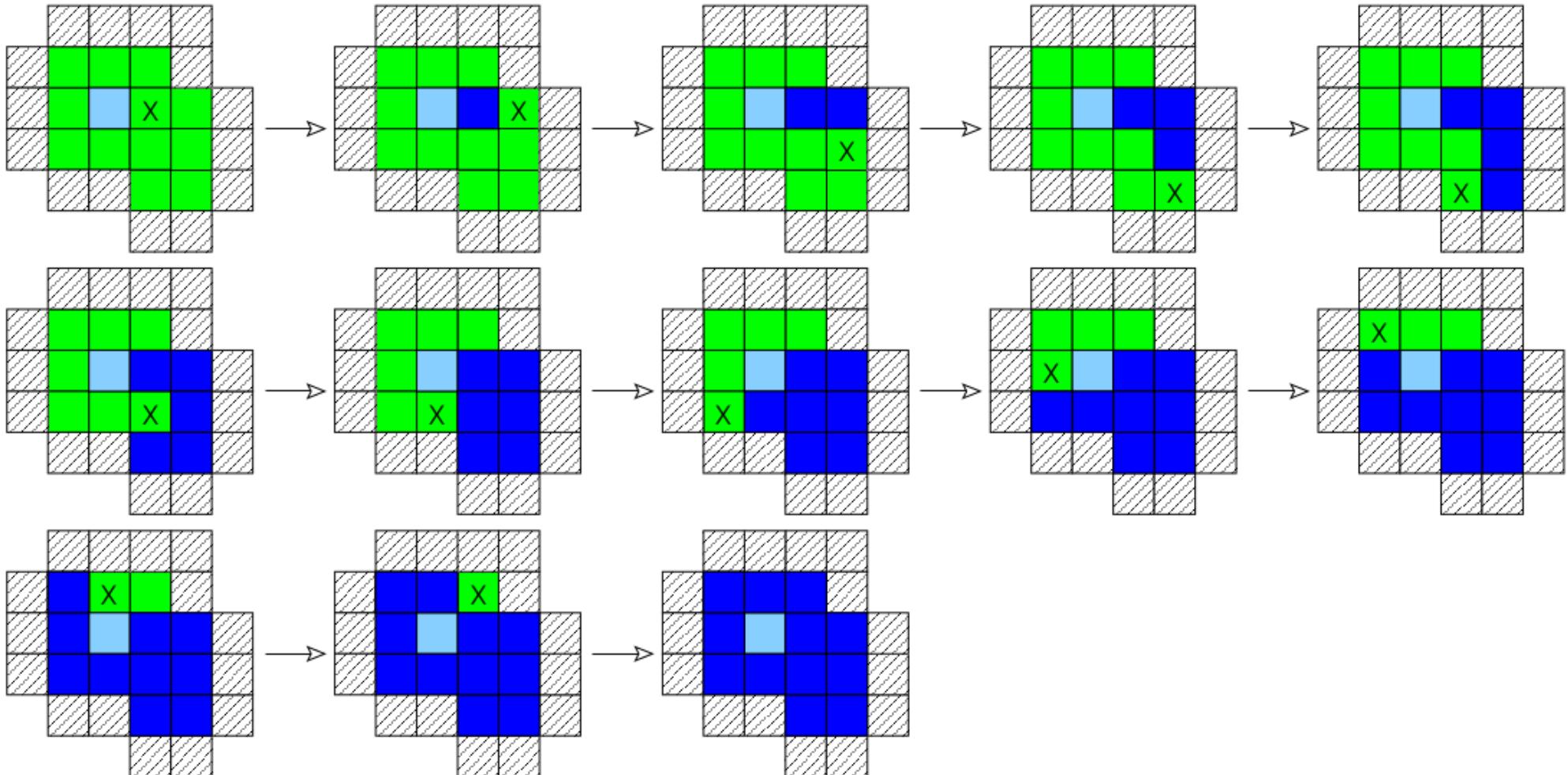
- Principe :
 - Propager le coloriage de proche en proche à partir du pixel de départ
 - Processus récursif !

Algorithme de remplissage

-
- 0) Récupérer la couleur c_i du pixel de départ p_i
 - 1) Appliquer 2) au pixel p_i
 - 2) $c_c = \text{couleur du pixel courant } p_c$
 - si $c_c = c_i$ alors // pixel p_c dans la composante connexe de p_i
 - Colorier p_c avec c_n // Application de la nouvelle couleur c_n
 - Pour chaque voisin v de p_c // Propagation
 - Appliquer 2) au pixel v

Algorithme de remplissage

- Déroulement à partir de X :



Tolérance de variation de couleur

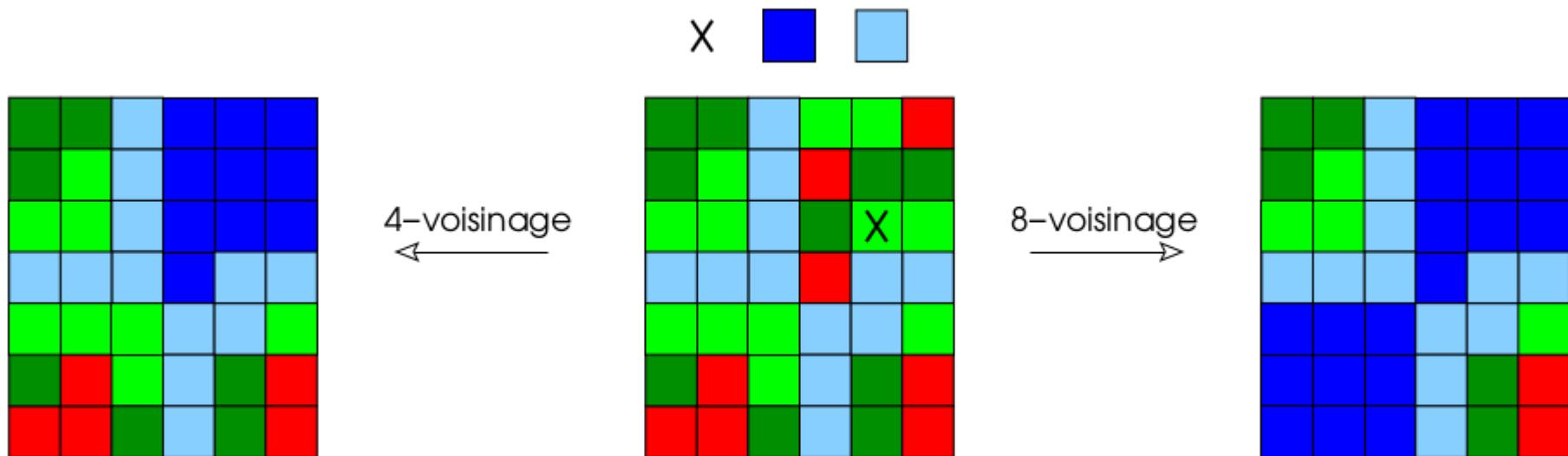
- Zones uniformes dans images naturelles très rares et limitées
- Utile de pouvoir étendre la propagation à des couleurs proches de la couleur initiale :
 - Différence maximale tolérée
 - Exprimée en pourcentage ou en unités d'intensité
 - Peut être différente pour chaque canal

Remplissage par contours

- Condition différente de remplissage :
 - Coloriage des *tous* les pixels dans une zone délimitée par un contour d'une couleur donnée
- Le choix de connexité change le résultat
- Schéma algorithmique similaire au précédent :
 - Échange des cas terminal et récursif
 - Propagation sur pixels de couleur différente du contour
 - Arrêt sur la couleur du contour
- Si le contour est ajouté à l'image avant remplissage, il faut bien choisir sa couleur
 - Pas déjà présente dans la zone concernée

Remplissage par contours

- Remplissage bleu foncé avec contour bleu clair

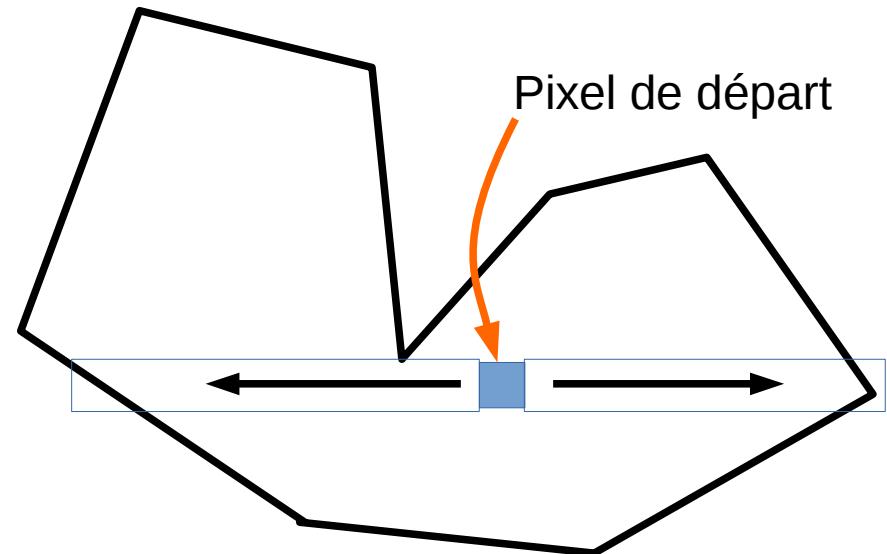


Problème avec la récursion

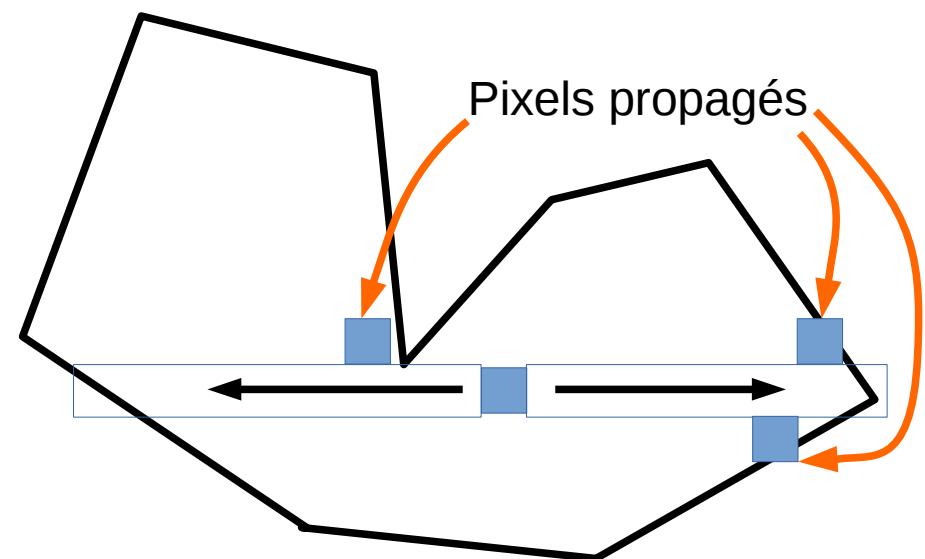
- Le remplissage récursif peut poser problème :
 - Profondeur de récursion = nombre de pixels remplis
 - Peut poser des problèmes de pile d'exécution lors du remplissage de très grandes zones !
- On doit donc :
 - Réduire la profondeur de récursion :
 - Version mixte : partie itérative et partie récursive
 - ou
 - Utiliser une version totalement itérative :
 - Transformation du récursif en itératif
 - Autre algorithme : parcours linéaire plutôt que propagation

Version mixte

- Partie itérative :
 - Pour un pixel de départ :
 - Colorier les pixels connexes sur la ligne courante
 - Parcours itératif vers la gauche
 - Parcours itératif vers la droite
 - Arrêt : couleur différente OU contour



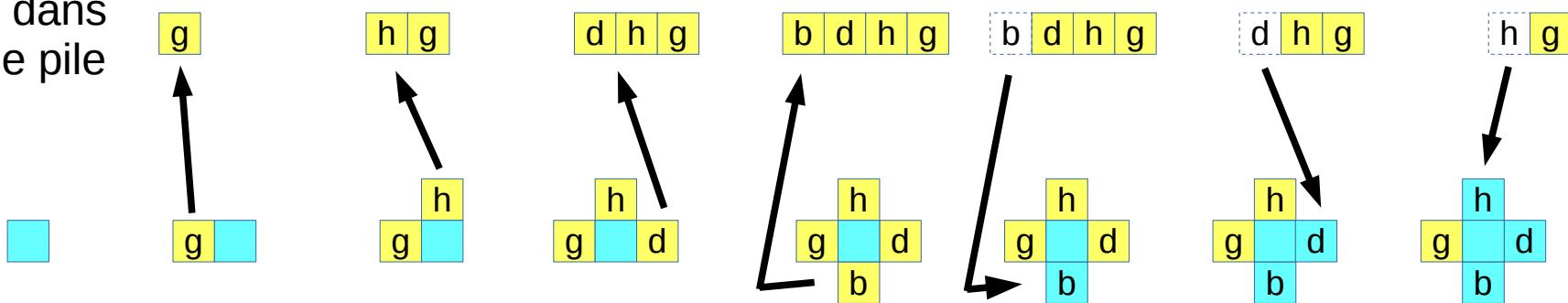
- Partie récursive :
 - Propager le remplissage aux 2 lignes voisines :
 - Aux pixels ayant un voisin à droite (on peut choisir gauche) de couleur différente (bord de zone)
 - Arrêt : couleur différente OU contour pour les pixels au-dessus et au-dessous du pixel parcouru



Version récursive transformée en itératif

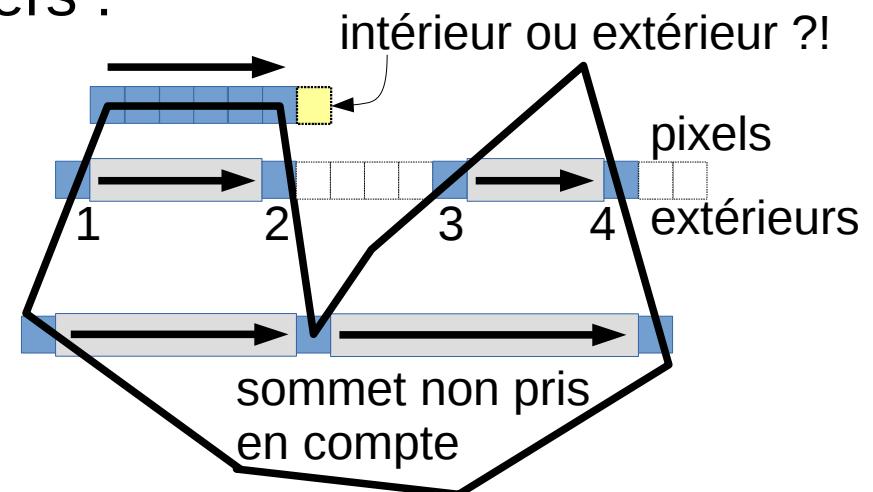
- Remplacer les appels récursifs par un *stockage* dans une *liste dynamique* (pile ou file) :
 - Stockage des pixels à traiter lors de la propagation
- Lorsque le traitement d'un pixel est terminé :
 - Récupérer un autre pixel à traiter depuis la liste
- L'algorithme s'arrête lorsqu'il n'y a plus de pixel à traiter dans la liste
- Exemple avec une pile :

Stockage dans
une pile



Parcours itératif

- Plutôt pour le remplissage par contours
- Parcours d'un rectangle englobant :
 - Pour chaque pixel, on détermine s'il est dans la forme ou non :
 - Compter le nombre de traversées du contour depuis le début de la ligne courante
 - Nombre impair pour l'intérieur du contour
 - MAIS il y a des cas particuliers :
 - Plusieurs pixels de contours consécutifs (bord horizontal)
 - Besoin d'infos sur l'historique du parcours
 - Un sommet du contour (il ne faut pas le compter)

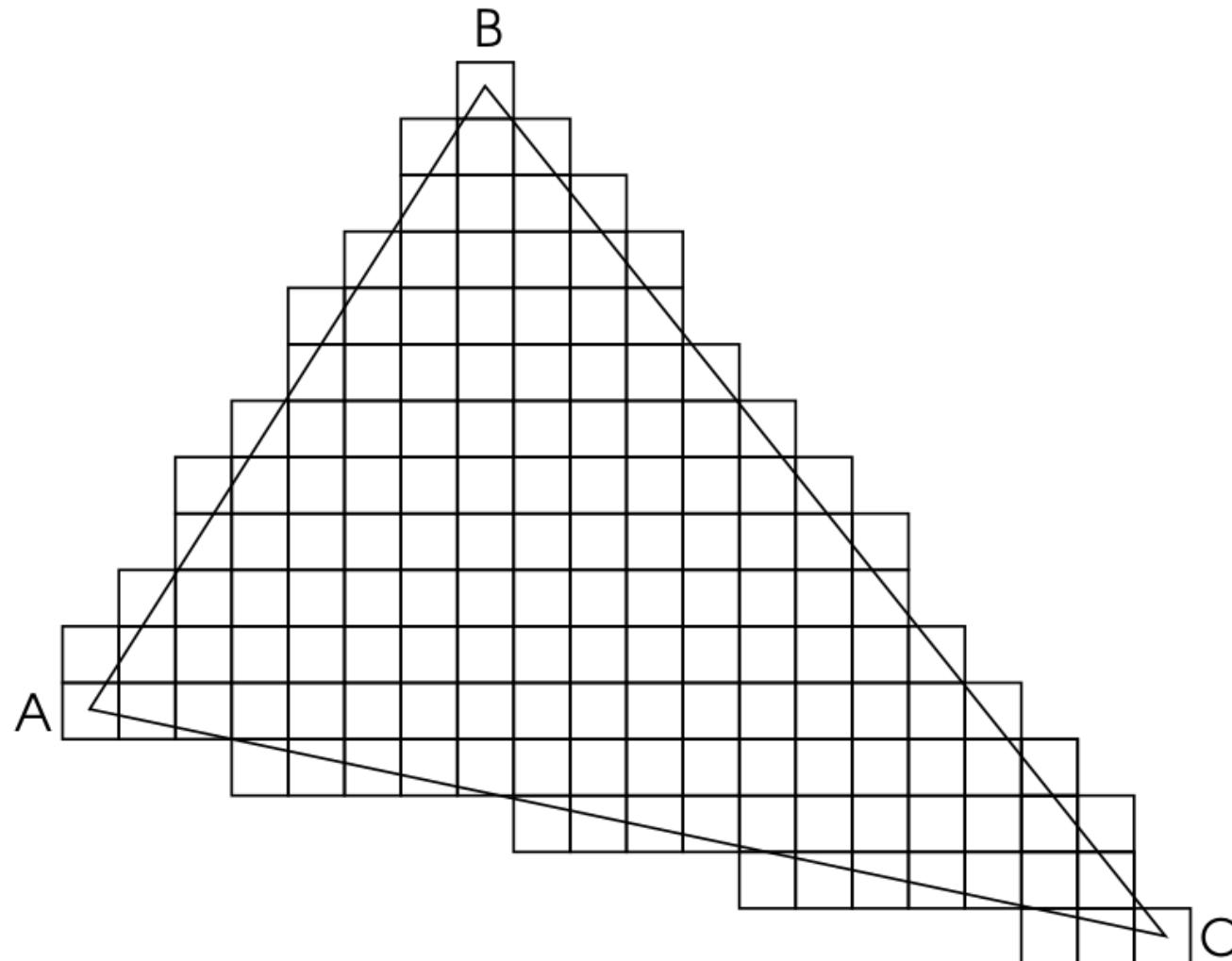


Remplissage de forme polygonale

- On considère une forme définie dans le plan réel
- Plusieurs approches possibles, notamment :
 - Tracer les segments du contour avec une couleur spécifique (non déjà présente)
 - +
 - Utiliser le remplissage par contours dans l'image
 - Décomposer le remplissage du polygone en un ensemble de tracés de formes remplies élémentaires :
 - Polygone le plus simple = le triangle
 - Il nous faut donc un tracé de triangle plein !

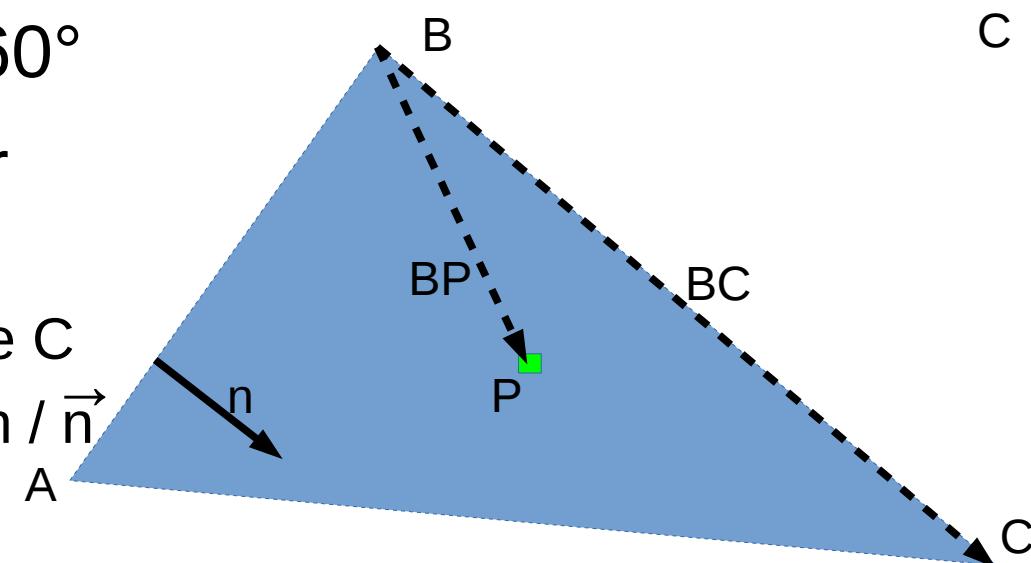
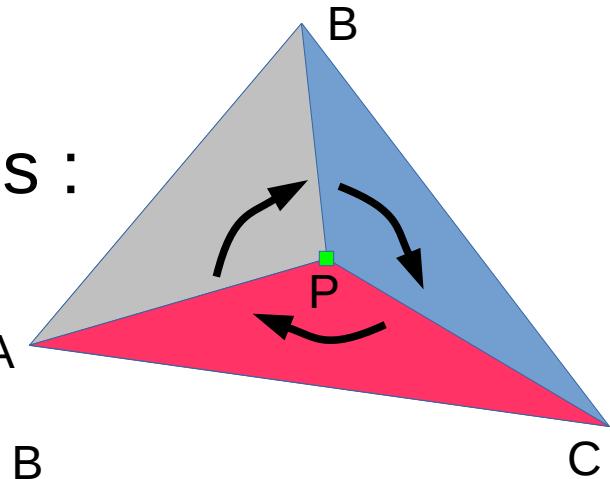
Dessin d'un triangle plein

- On a trois points A, B, C
- On doit colorier tous les pixels dans le triangle



Dessin d'un triangle plein

- On peut parcourir le rectangle englobant et tester pour chaque pixel s'il est dans le triangle
- Il faut choisir un test d'inclusion :
 - Basé sur les aires des sous-triangles :
 $ABC = APB + BPC + APC$
 - Basé sur les angles issus du pixel
 $\widehat{APB} + \widehat{BPC} + \widehat{CPA} = 360^\circ$
 - Basé sur la position par rapport à chaque côté :
 - P du même côté de AB que C
 - BP et BC même orientation / \vec{n}
 - $\vec{n} \cdot \vec{BP} \times \vec{n} \cdot \vec{BC} > 0$

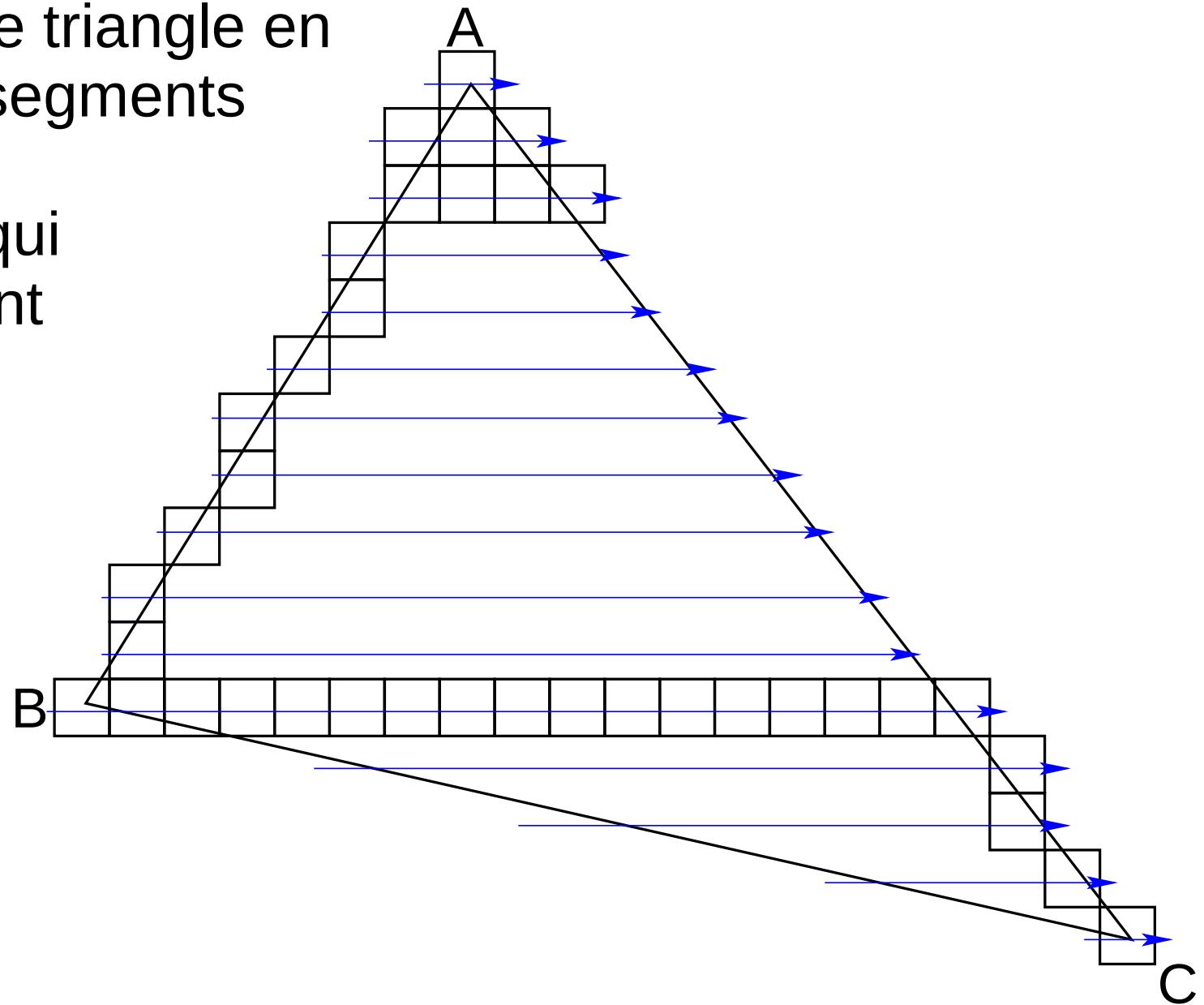


Dessin d'un triangle plein

- Cela donne le bon résultat MAIS :
 - Pixels hors triangle parcourus inutilement
 - Pas d'information sur la position relative du pixel courant dans le triangle
 - ⇒ Nécessaire pour certains traitements (texturage,...)
- Algorithme de dessin ligne par ligne :
 - Décomposer le triangle en lignes horizontales
 - Colorier une suite de segments horizontaux

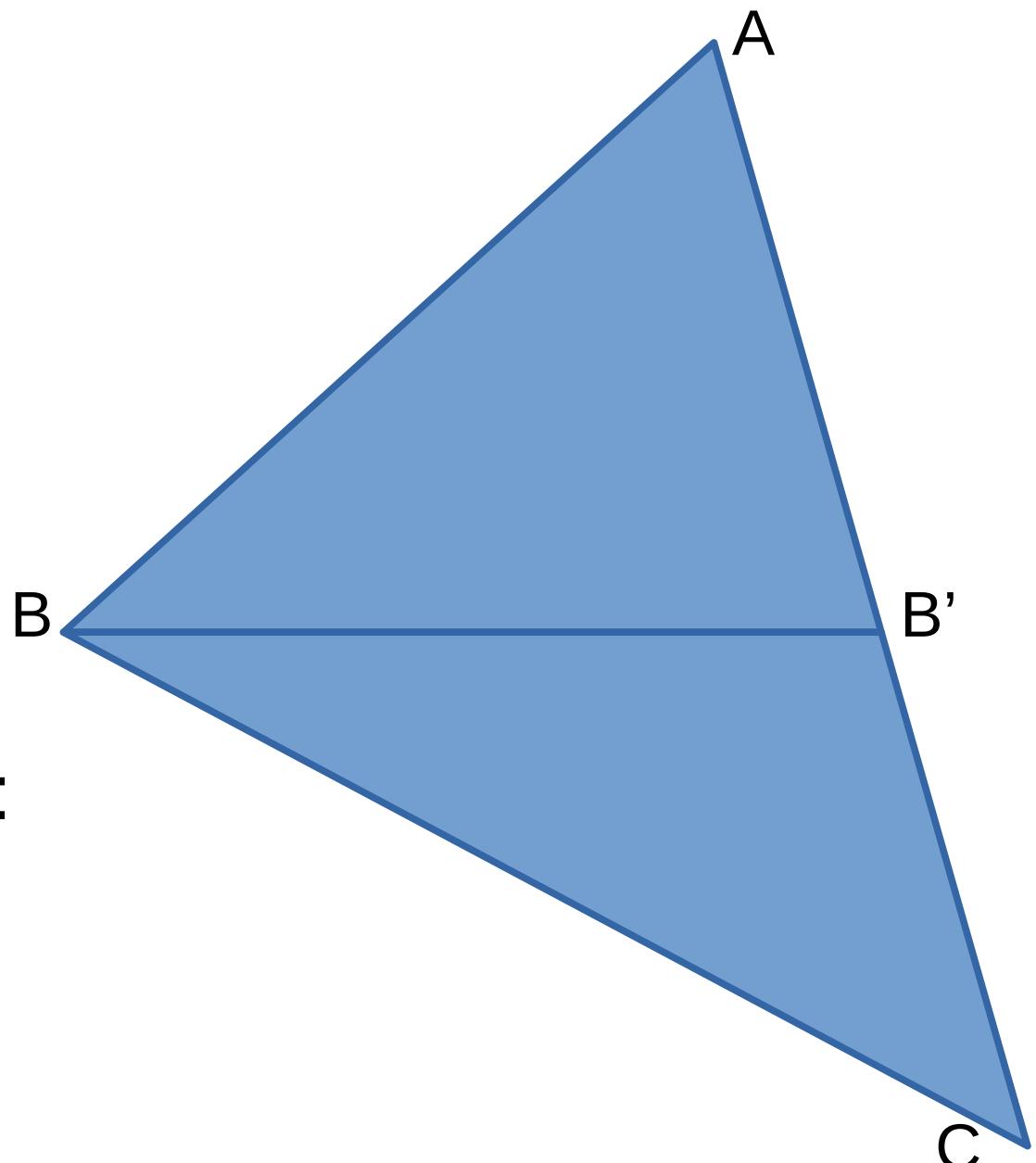
Algorithme ligne par ligne

- On remplit le triangle en traçant les segments horizontaux successifs qui le composent



Décomposition en deux parties

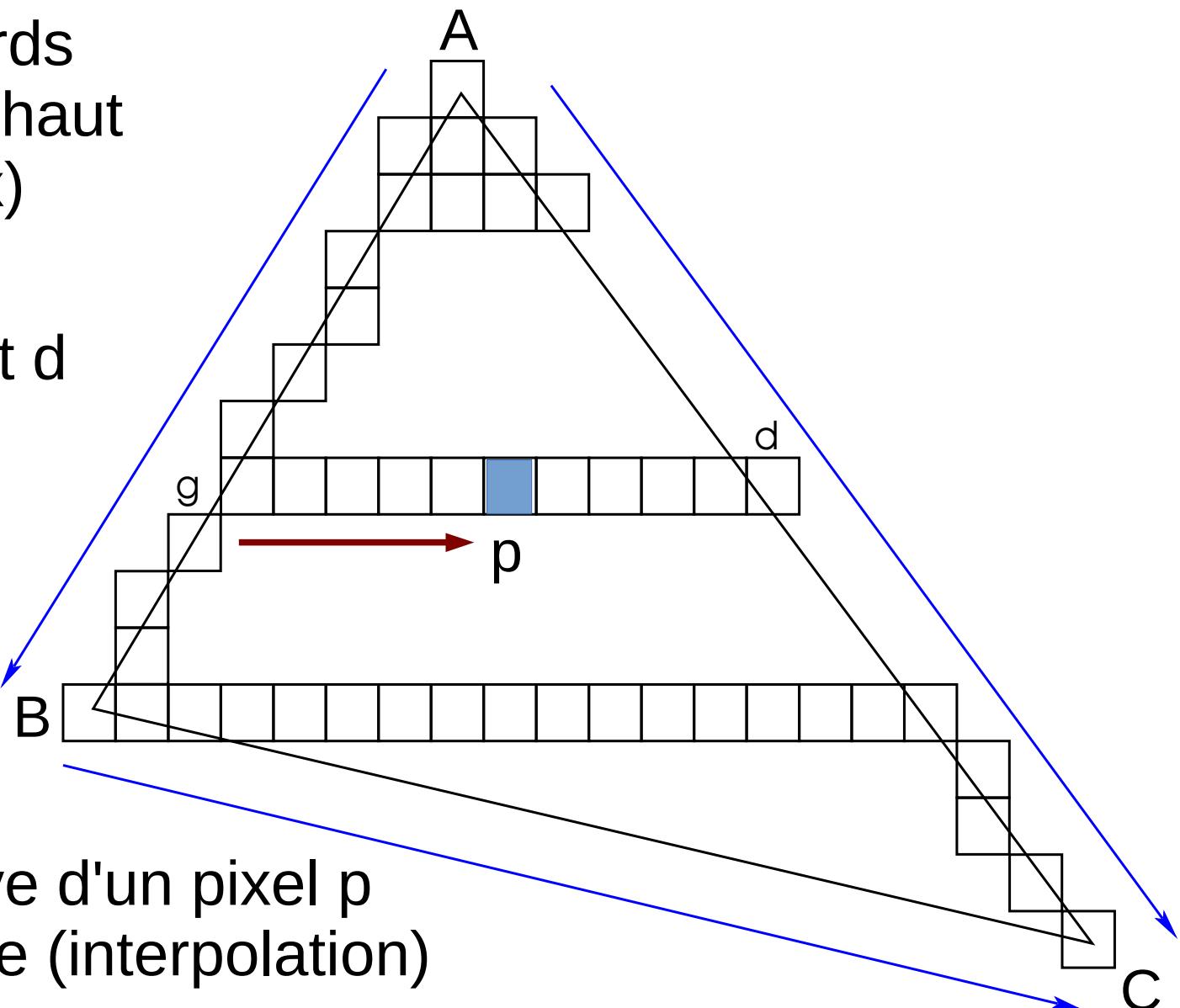
- Côté inférieur plat :



- Côté supérieur plat :

Algorithme ligne par ligne

- On suit les bords du triangle de haut en bas (par ex)
- On déduit les extrémités g et d du segment horizontal dans le triangle
- Permet de déduire la position relative d'un pixel p dans le triangle (interpolation)



Version classique par lignes

```

fonction DessineTrianglePlein(pi1 : PointImage, pi2 : PointImage,
    pi3 : PointImage, coul : Couleur) : Vide
// Remplit le triangle défini par les points pi1, pi2 et pi3 avec la couleur coul

A ← pi1 // On considère initialement les points dans le bon ordre
B ← pi2
C ← pi3

// Tri des 3 points dans l'ordre croissant des lignes
si A.lig > B.lig alors
    échange(A, B)
fsi
si B.lig > C.lig alors
    échange(B, C)
fsi
si A.lig > B.lig alors
    échange(A, B)
fsi

// Calcul des 3 vecteurs que l'on va parcourir
AB ← PointImage(B.col - A.col, B.lig - A.lig) // Vecteur AB
AC ← PointImage(C.col - A.col, C.lig - A.lig) // Vecteur AC
BC ← PointImage(C.col - B.col, C.lig - B.lig) // Vecteur BC

```

// Construction des 2 points extrêmes pour chaque ligne
 pAB ← PointImage() // Point sur le vecteur AB
 pAC ← PointImage() // Point sur le vecteur AC

```

// Parcours des lignes de A à B seulement si A et B sur lignes différentes
si AB.lig > 0 alors
    pour l de 0 à AB.lig-1 faire
        pAB.col ← A.col + AB.col * l / AB.lig // Calcul du point courant sur AB
        pAB.lig ← A.lig + l
        pAC.col ← A.col + AC.col * l / AC.lig // Calcul du point courant sur AC
        pAC.lig ← pAB.lig
        DessineSegmentImage(pAB, pAC, coul) // Dessin du segment horizontal
    fpour
sinon
    DessineSegmentImage(A, B, coul) // Dessin du segment horizontal
fsi

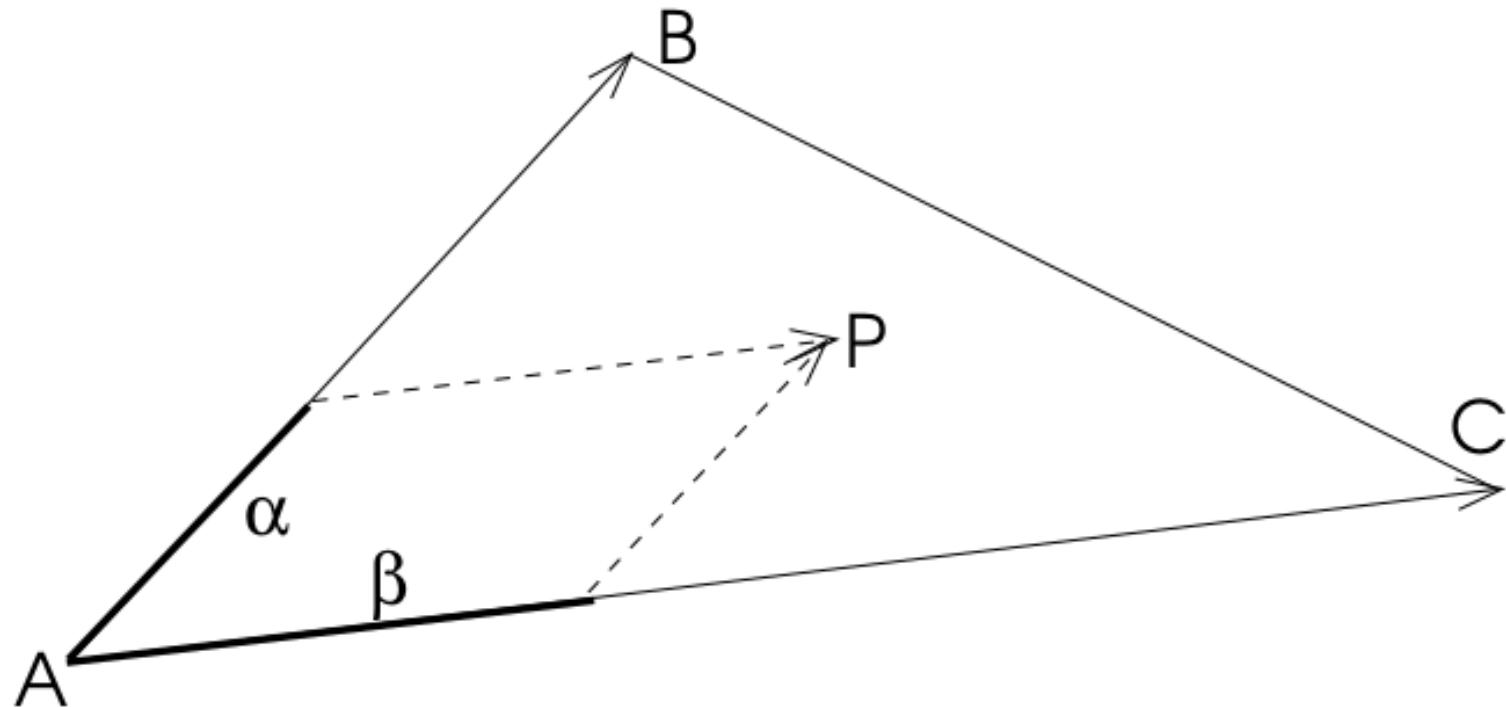
pBC ← PointImage() // Point sur le vecteur BC

// Parcours des lignes de B à C seulement si B et C sur lignes différentes
si BC.lig > 0 alors
    pour l de AB.lig à AC.lig faire // Boucle dans la continuité de la 1ère
        pBC.col ← B.col + BC.col * (l-AB.lig) / BC.lig // Calcul du pt crt sur BC
        pBC.lig ← A.lig + l
        pAC.col ← A.col + AC.col * l / AC.lig // Calcul du point courant sur AC
        pAC.lig ← pBC.lig
        DessineSegmentImage(pBC, pAC, coul) // Dessin du segment horizontal
    sinon
        DessineSegmentImage(B, C, coul) // Dessin du segment horizontal
    fsi

```

Version bi-linéaire

$$P = A + \alpha \cdot \overrightarrow{AB} + \beta \cdot \overrightarrow{AC} \quad \text{avec} \quad 0 \leq \alpha + \beta \leq 1$$



Algorithme bi-linéaire

```
fonction DessineTrianglePlein(pi1 : PointImage, pi2 : PointImage,
                               pi3 : PointImage, coul : Couleur) : Vide
    // Remplit le triangle défini par les points pi1, pi2 et pi3 avec la couleur coul

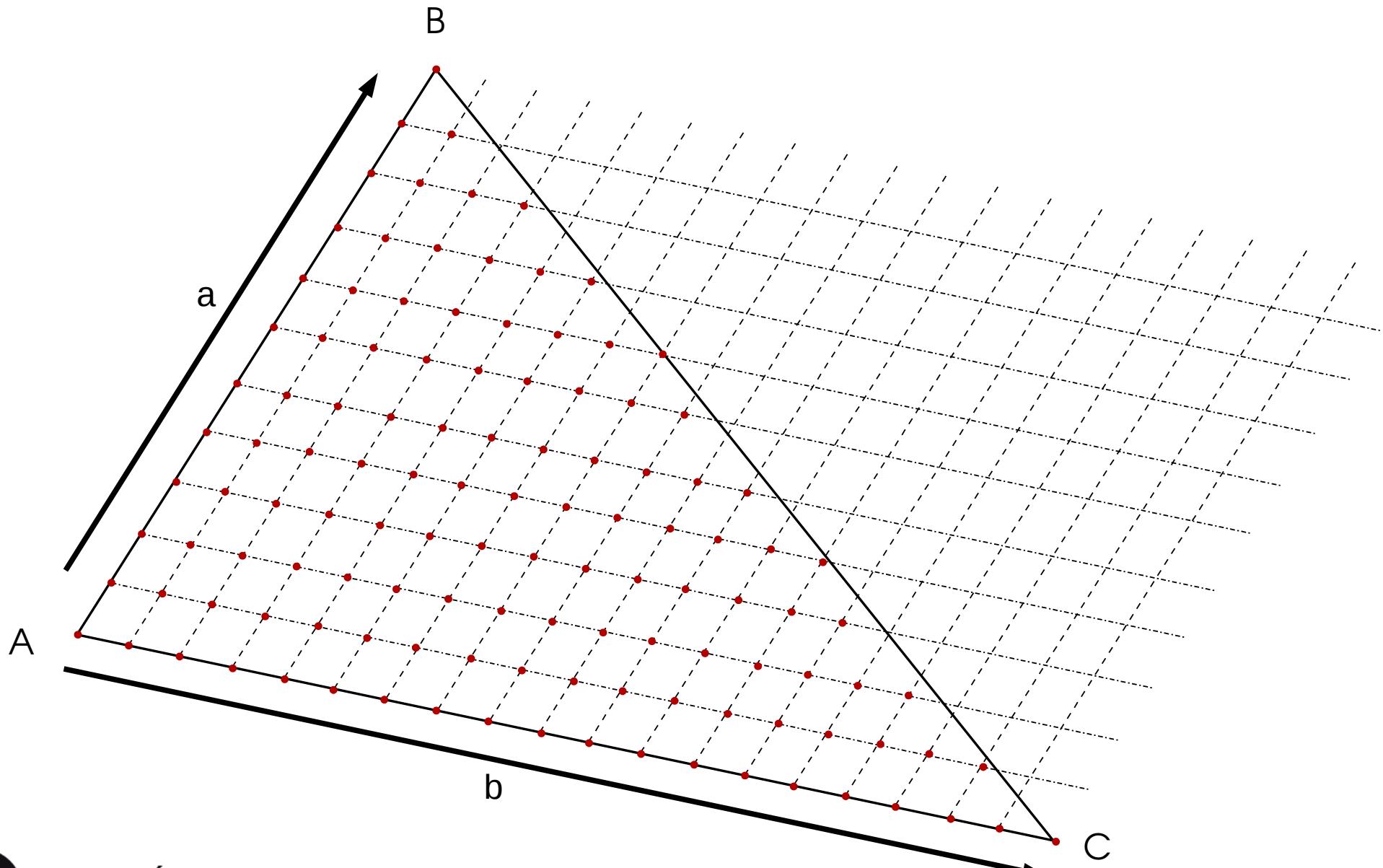
    // Construction des 2 vecteurs issus des sommets 1-2 et 1-3
    AB ← PointImage(pi2.col - pi1.col, pi2.lig - pi1.lig)
    AC ← PointImage(pi3.col - pi1.col, pi3.lig - pi1.lig)
    P ← PointImage()

    // Calcul du delta nécessaire pour parcourir AB
    mAB ← max(abs(AB.col), abs(AB.lig))
    dAB ← 1
    si mAB ≠ 0 alors
        dAB ← 1.0 / mAB

    // Calcul du delta nécessaire pour parcourir AC
    mAC ← max(abs(AC.col), abs(AC.lig))
    dAC ← 1
    si mAC ≠ 0 alors
        dAC ← 1.0 / mAC

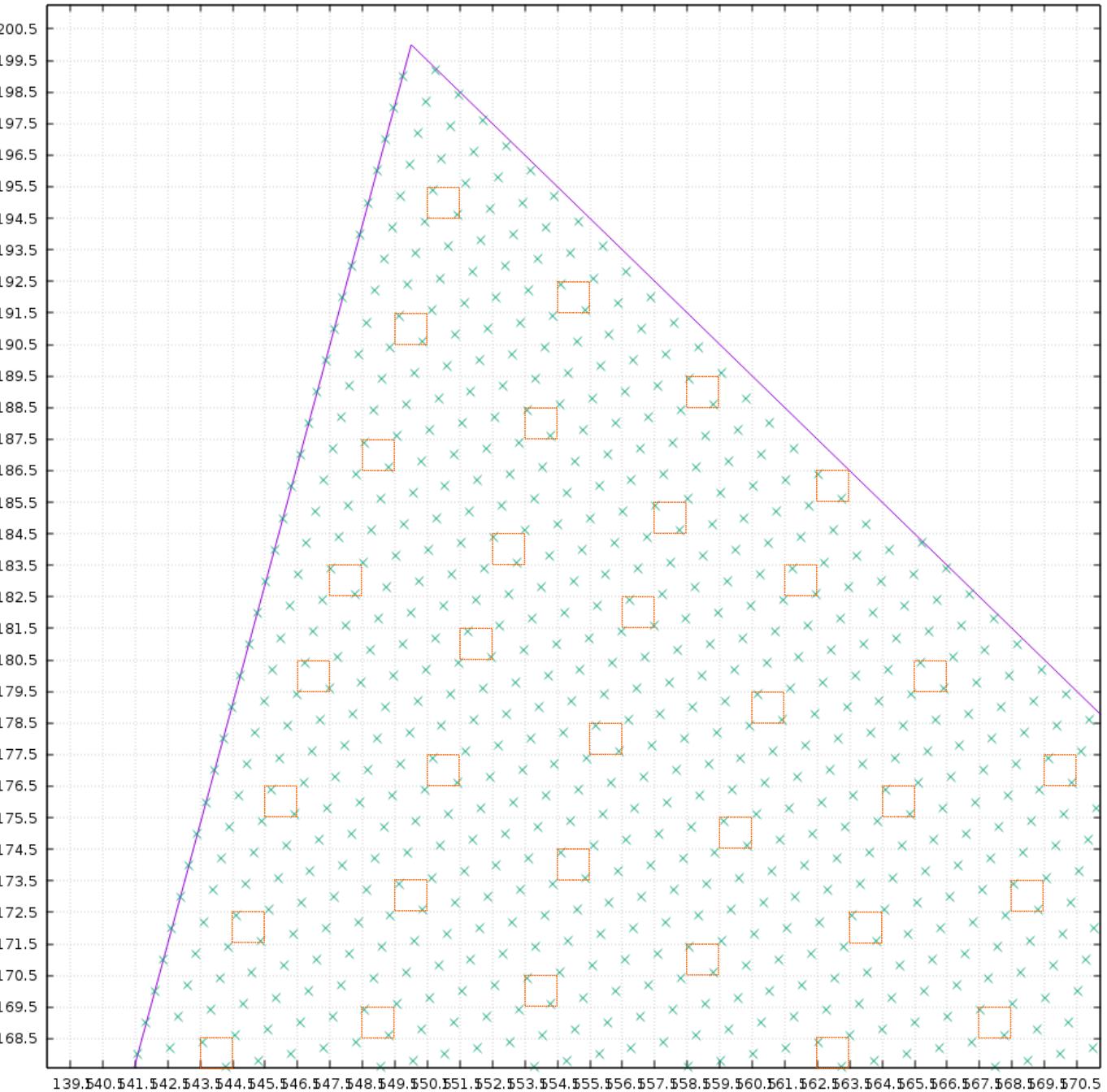
    // Double boucle de parcours
    a ← 0.0
    tant que a ≤ 1.0 faire // Parcours selon AB
        b ← 0.0
        tant que a+b ≤ 1.0 faire // Parcours selon AC sans dépasser le triangle
            P.x ← int(round(pi1.col + a * AB.col + b * AC.col))
            P.y ← int(round(pi1.lig + a * AB.lig + b * AC.lig))
            ColoriePixel(P.x, P.y, coul)
            b ← b + dAC
        ftant
        a ← a + dAB
    ftant
```

Exemple de parcours bi-linéaire

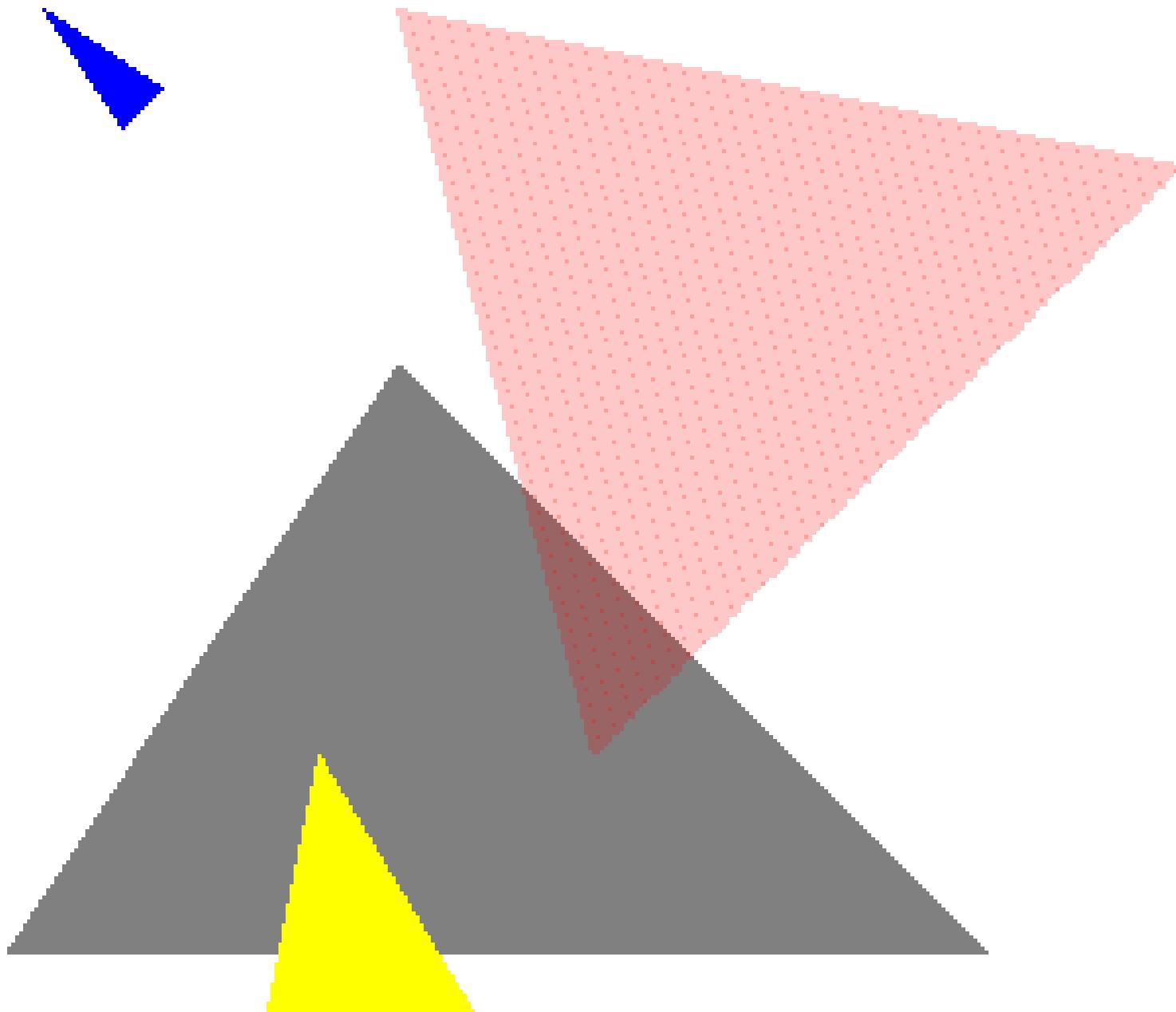


Problème des parcours multiples

- Certains pixels sont parcourus plusieurs fois
→ Carrés rouges
- Pas gênant dans la plupart des cas
- Pose problème si on calcule un cumul des pixels :
→ Transparence,...



Problème avec la transparence

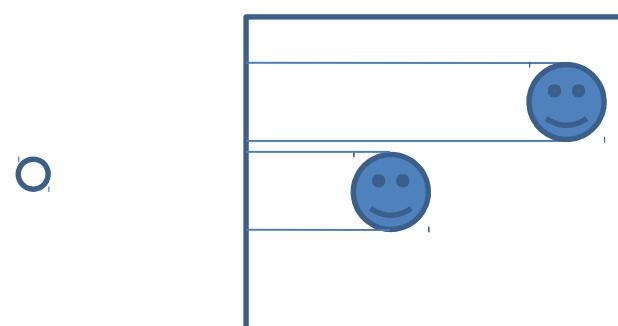
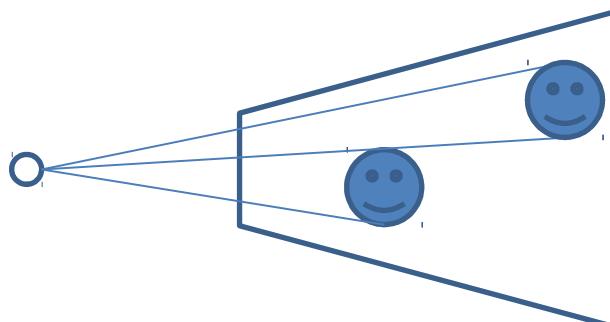
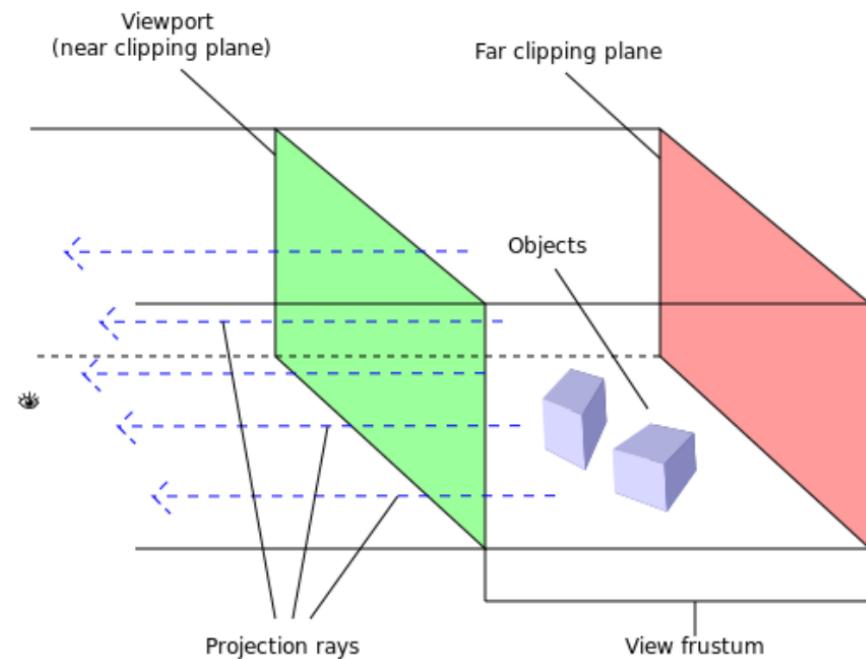
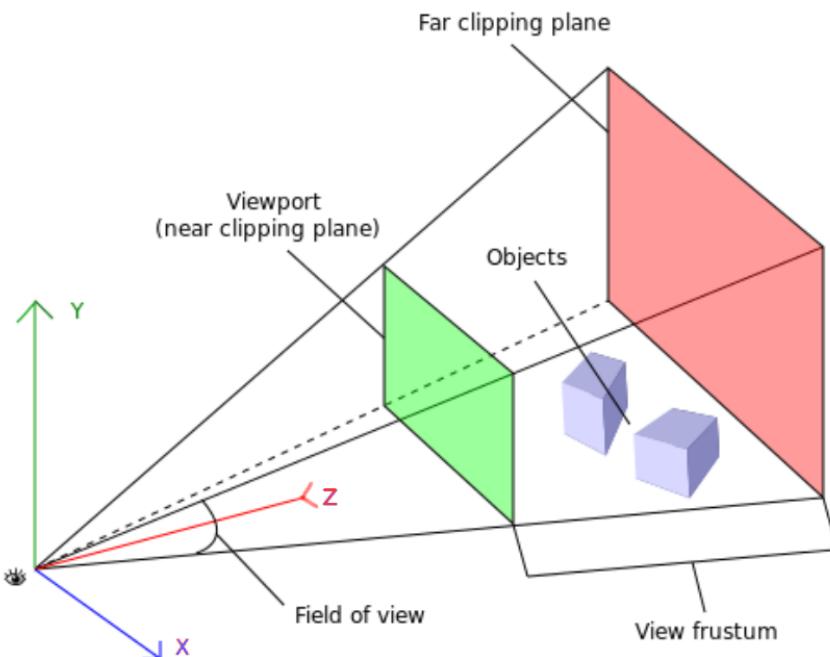


Interpolation linéaire dans le triangle

- On peut déduire des informations à l'intérieur du triangle à partir de ses trois sommets (interpolation)
- Positions relatives de g et d sur AB et AC :
 - $\alpha = Ag/AB$ et $\beta = Ad/AC$
- Position relative du pixel p entre g et d :
 - $\delta = gp/gd$
- Si A, B et C sont dans R^3 alors le z de p est donné par :
 - $g.z = A.z + \alpha * (B.z - A.z)$
 - $d.z = A.z + \beta * (C.z - A.z)$
 - $p.z = g.z + \delta * (d.z - g.z)$
- Utilisable aussi pour déduire d'autres infos :
 - couleur, lumière, normale,...

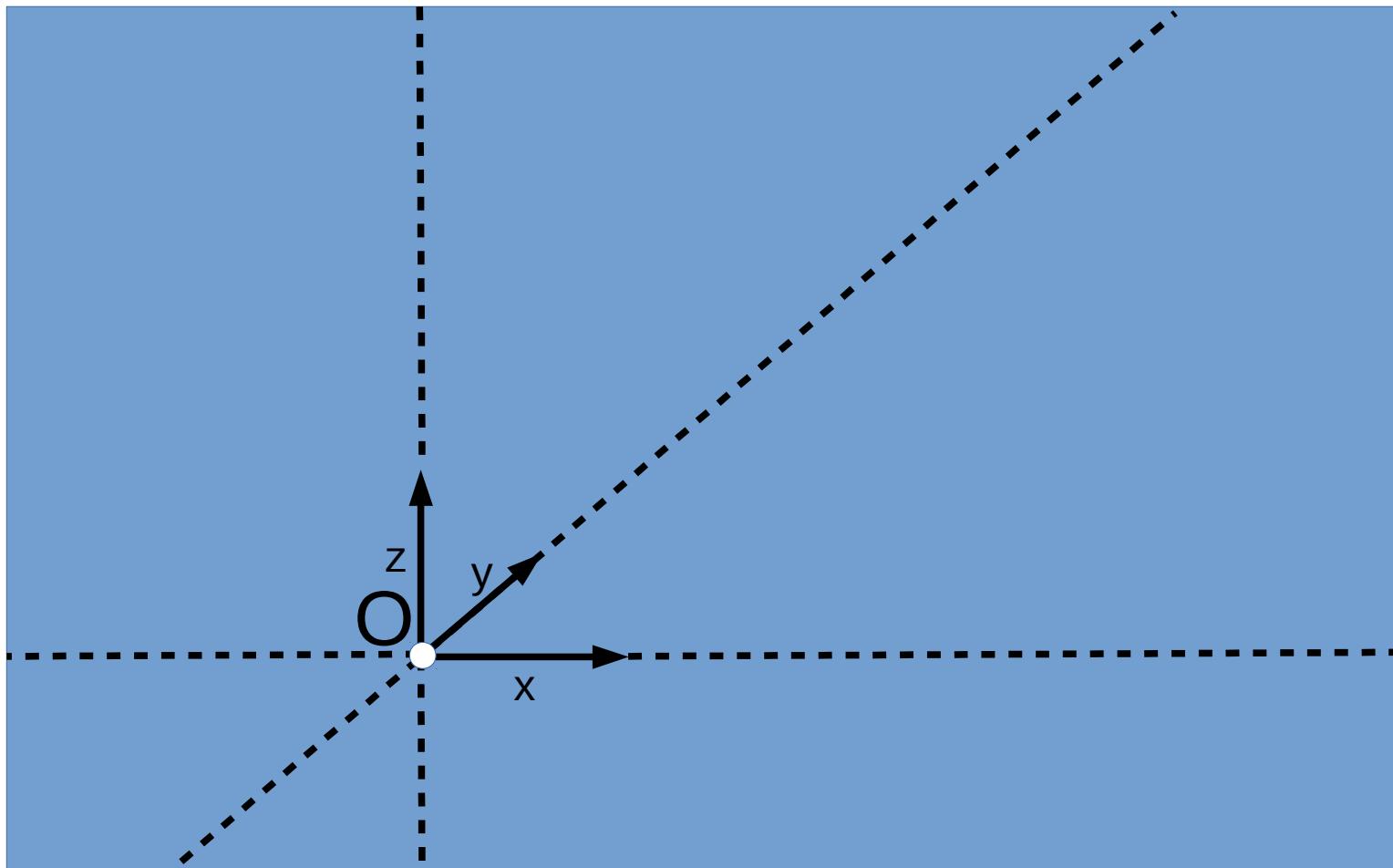
3D Isométrique

- Différence entre perspective et isométrie :



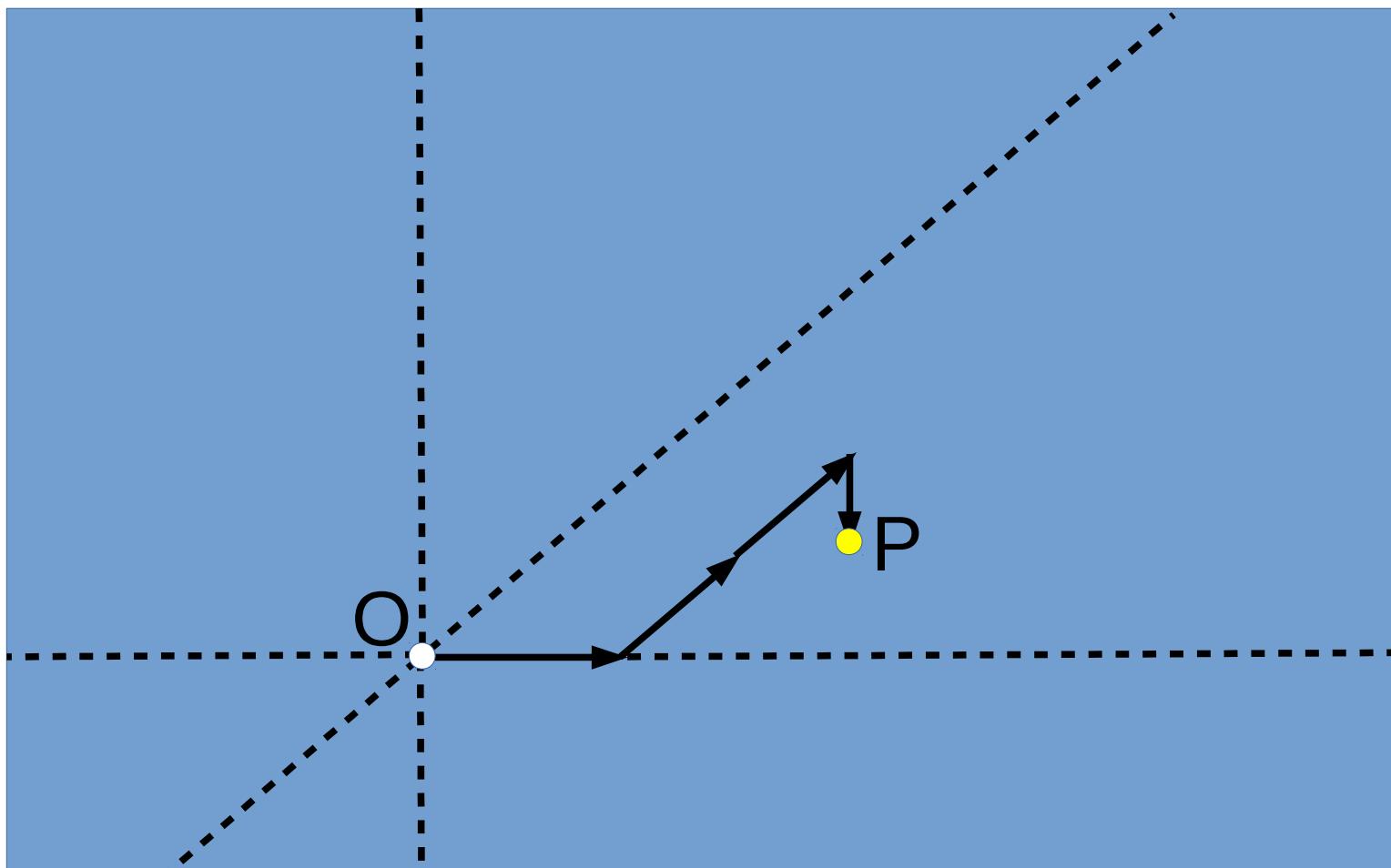
Principe de la 3D Isométrique

- Définition de l'origine O et des 3 vecteurs unitaires x, y, z du repère 3D, dans le plan *image* (2D) :

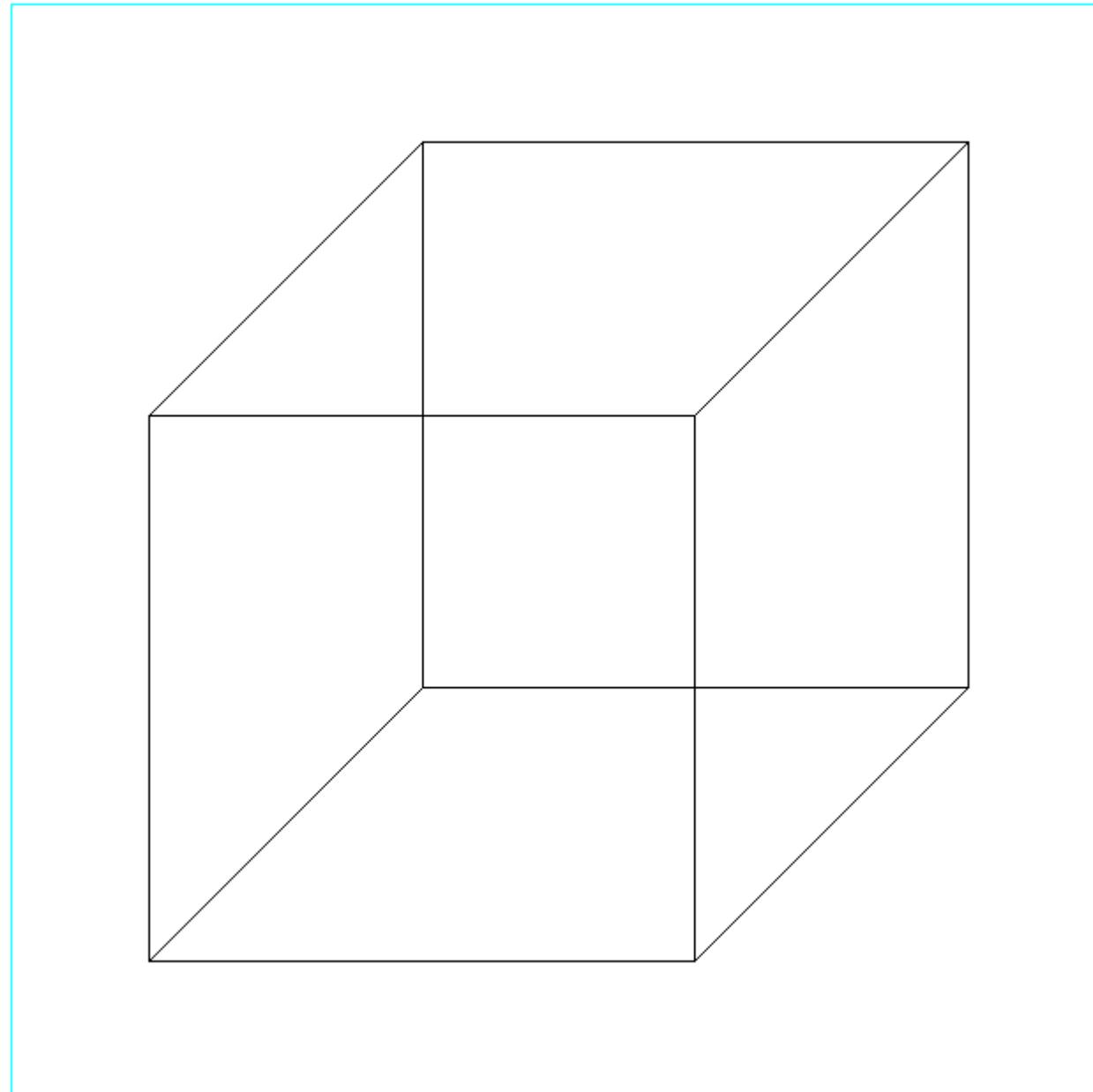


Principe de la 3D Isométrique

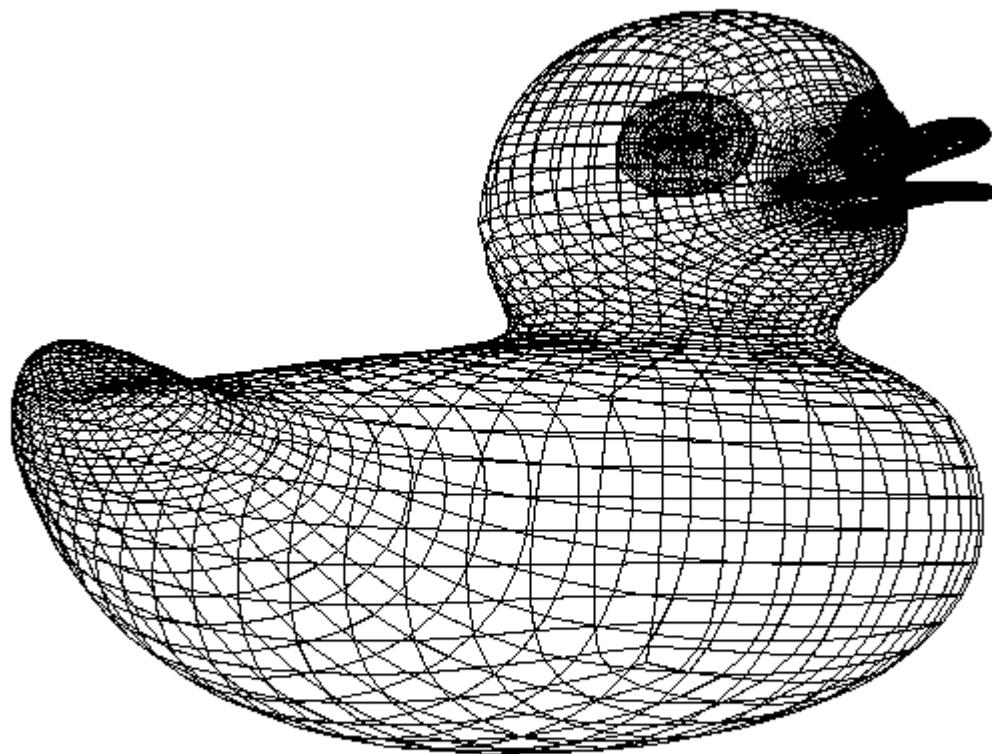
- P est le résultat du déplacement $(1, 2, -0.5)$ depuis O :



Exemple d'affichage « fil de fer »



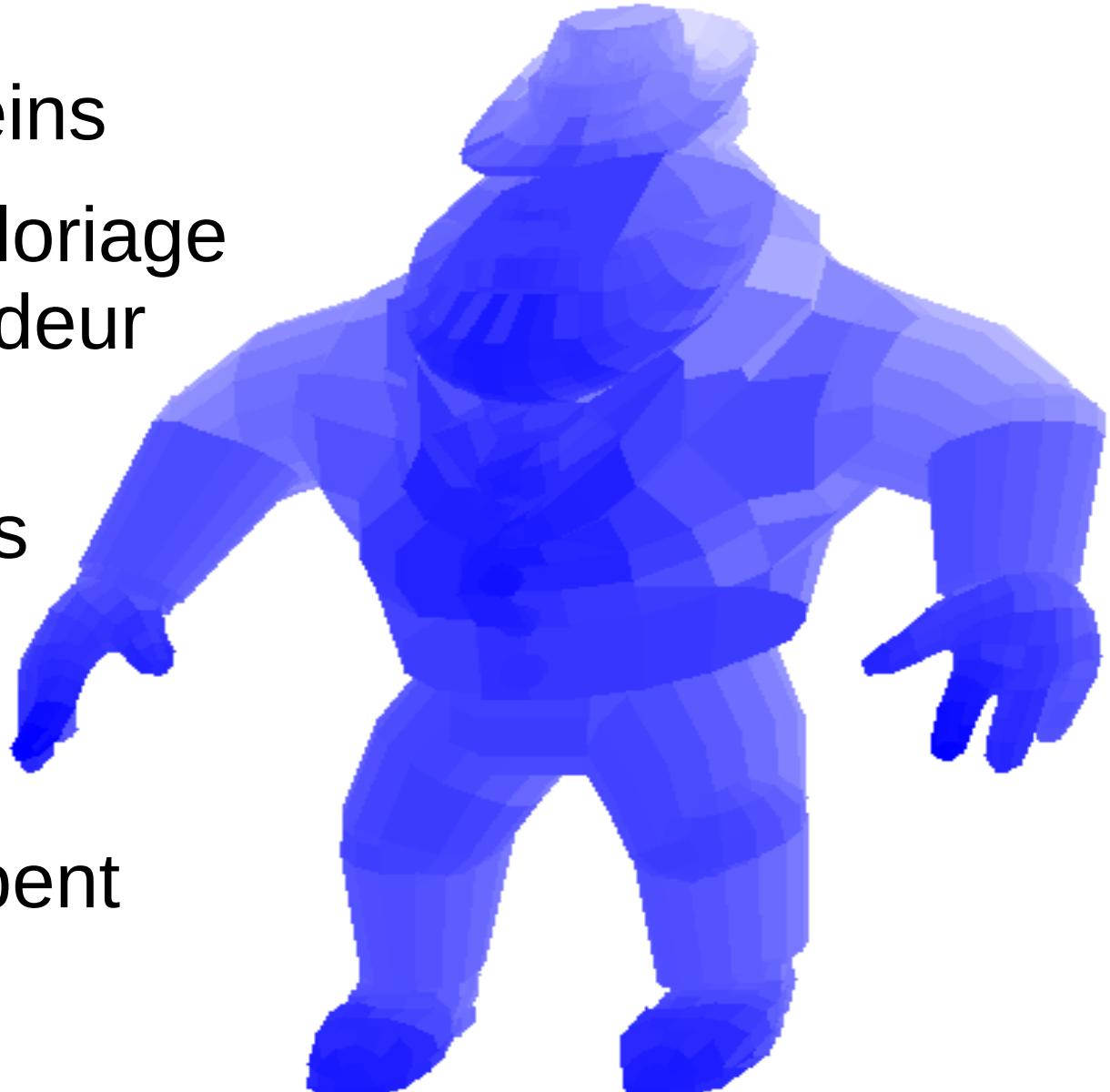
Exemple d'affichage « fil de fer »



Exemple d'affichage « plein »

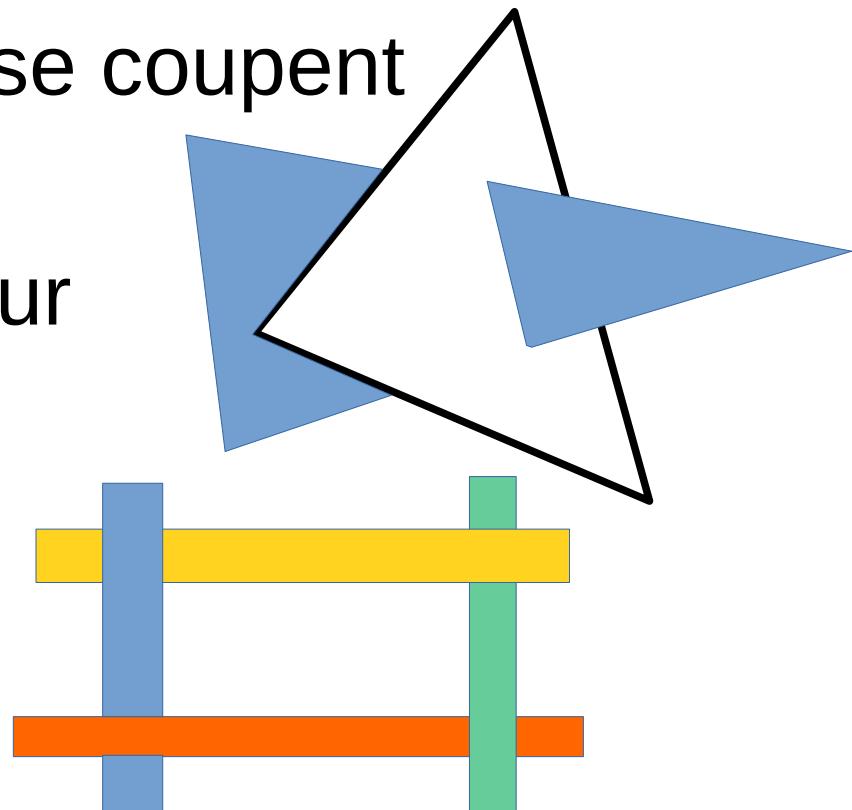
- Utilise le tracé de triangles pleins
- Exemple de coloriage selon la profondeur des triangles
- Mais problèmes selon l'ordre d'affichage des triangles ou s'ils se coupent

→ Z-buffer

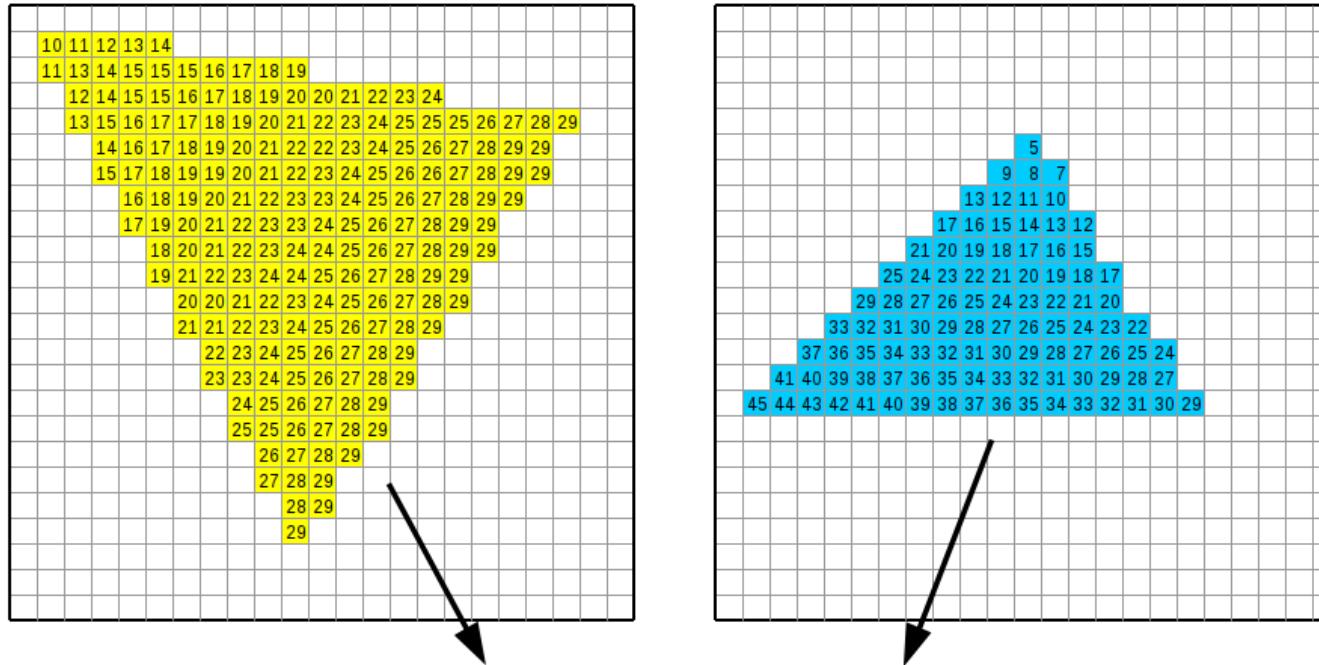


Principe du Z-buffer

- Tableau de mêmes dimensions que l'image
- Chaque case du tableau :
 - Correspond à un pixel de l'image
 - Contient la profondeur du dernier coloriage du pixel
- Utile quand des polygones se coupent ou se chevauchent
- Interpolation de la profondeur
- Écrasement seulement si le nouveau pixel est devant l'ancien



Application du Z-buffer



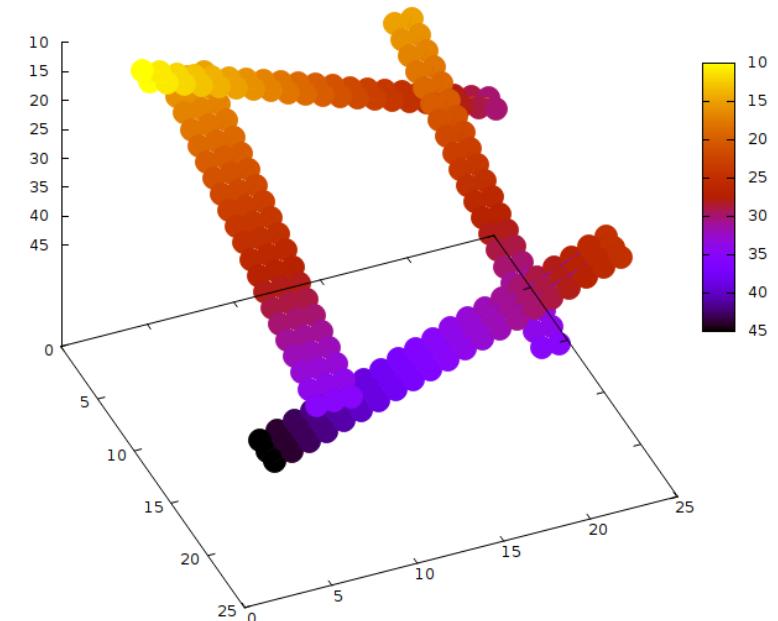
10 11 12 13 14									
11 24 25 26 27 15 16 17 18 19									
12 25 26 27 27 28 29 30 31 20 21 22 23 24									
13 26 27 28 29 30 31 32 33 33 34 35 36 37 25 26 27 28 29									
14 28 29 30 31 32 33 34 5 36 37 37 38 39 40 41 29									
15 28 29 30 31 32 33 9 8 7 36 37 38 39 40 40 29									
16 30 31 32 33 13 12 11 10 37 38 39 40 41 29									
17 31 32 33 17 16 15 14 13 12 39 40 41 29									
18 32 33 20 19 18 17 16 15 39 40 41 29									
19 33 34 34 22 21 20 19 18 17 41 29									
29 20 20 21 22 23 23 22 21 20 28 29									
33 32 21 21 22 23 24 25 25 24 23 22									
37 36 35 34 22 23 24 25 26 27 27 26 25 24									
41 40 39 38 37 23 23 24 25 26 27 28 29 28 27									
45 44 43 42 41 40 39 24 25 26 27 28 29 32 31 30 29									
25 25 26 27 28 29									
26 27 28 29									
27 28 29									
28 29									
29									

Application du Z-buffer

10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25

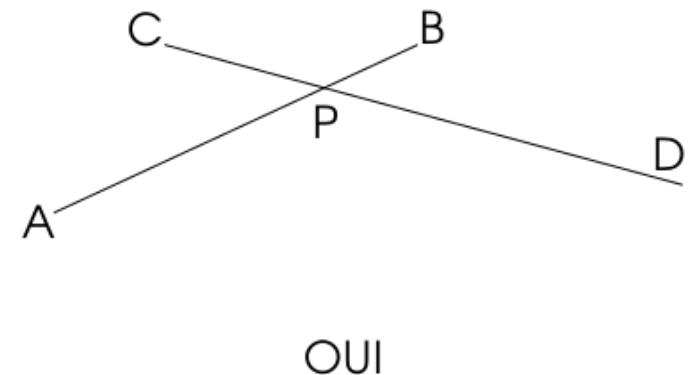
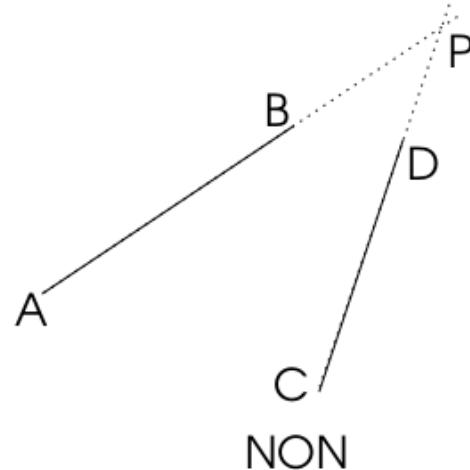
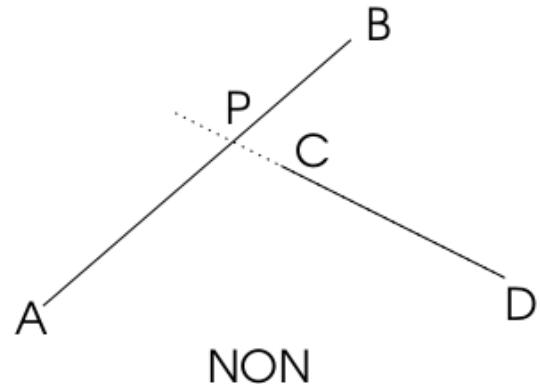
15 15 15
16 16 16
17 17 17
18 18 18
19 19 19
20 20 20
21 21 21
22 22 22
23 23 23
24 24 24
25 25 25
26 26 26
27 27 27
28 28 28
29 29 29
30 30 30
31 31 31
32 32 32
33 33 33
34 34 34
35 35 35
15 15

15 15 15
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 16 16 27 28 29 30
10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 17 17 27 28 29 30
18 18 18
19 19 19
20 20 20
21 21 21
22 22 22
23 23 23
24 24 24
25 25 25
26 26 26
27 27 27
28 28 28
29 29 29
30 30 30
31 31 31
45 44 32 32 32 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
45 44 33 33 33 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
45 44 34 34 34 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25
35 35 35
35 35 35

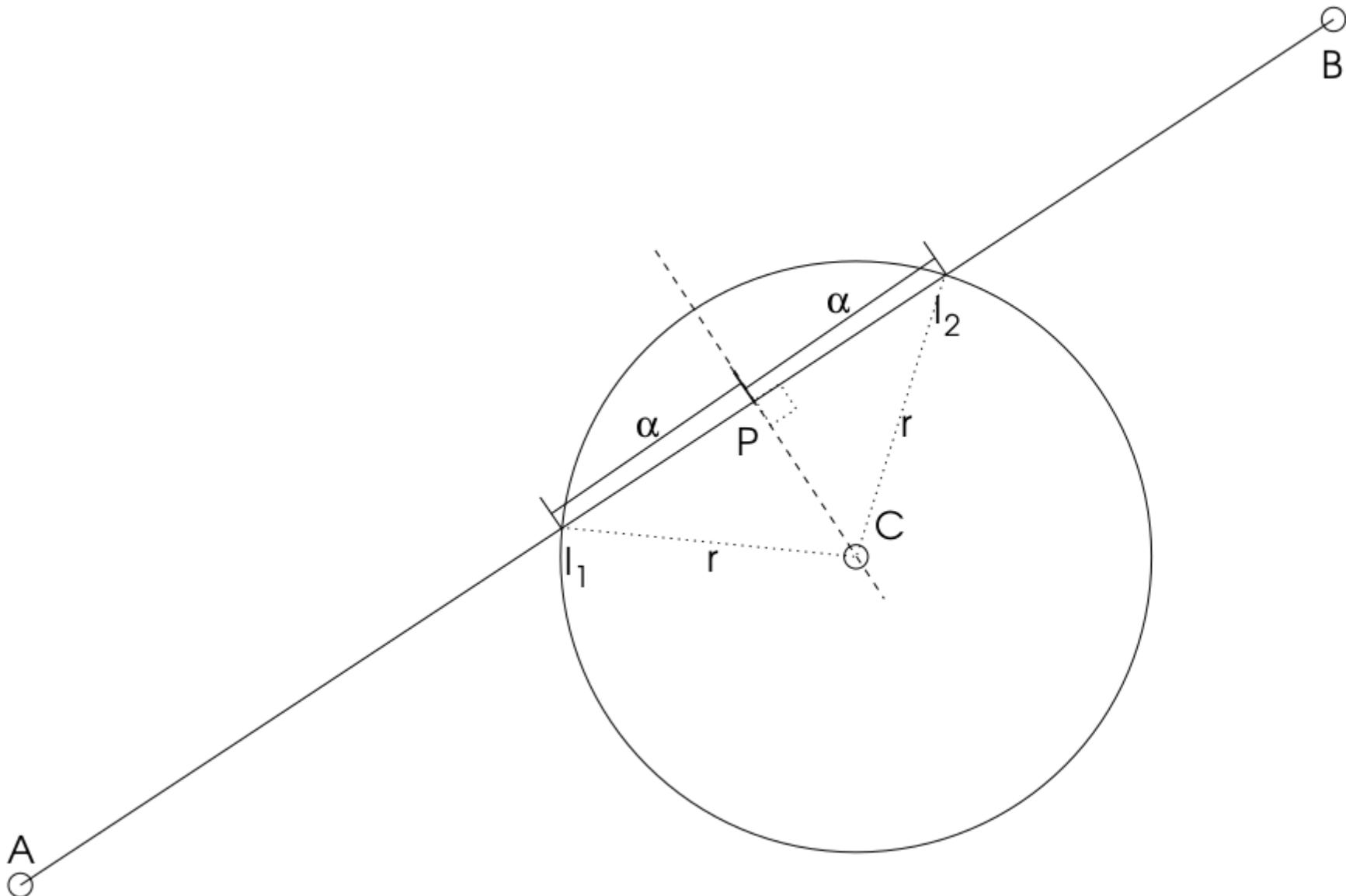


Intersection de segments

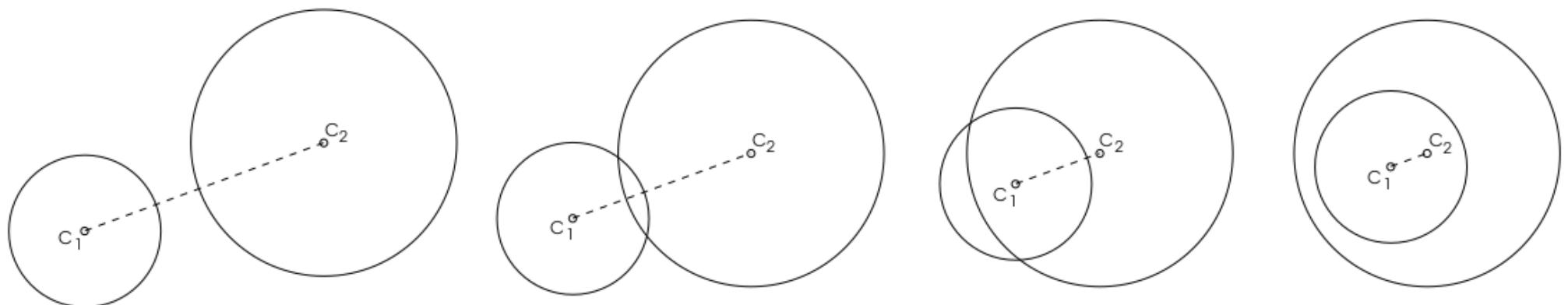
- Configurations possibles :



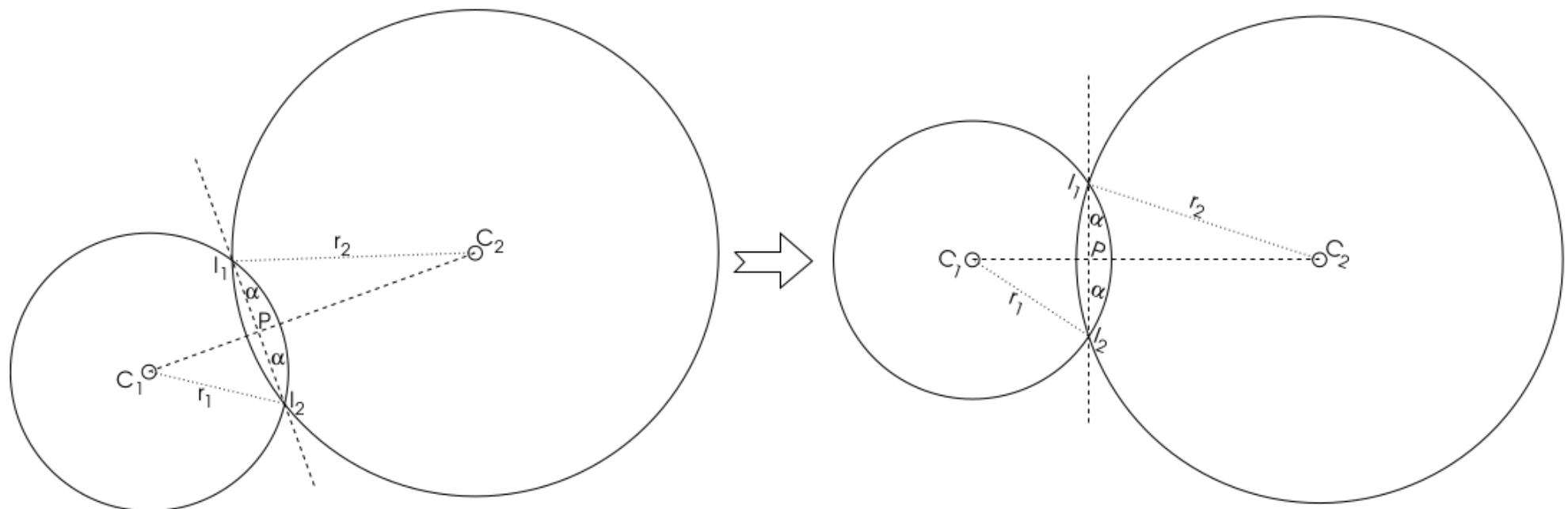
Intersection segment - cercle



Intersections entre cercles



Repositionnement des cercles



Intersection segment - Bézier

- Approximation linéaire

