# 1 Clipping (Découpage)

## 1.1 Introduction

Le clipping, ou découpage, est une étape fondamentale du pipeline graphique. Il consiste à éliminer les parties des objets géométriques qui se trouvent en dehors d'une région spécifiée, appelée fenêtre de clipping.

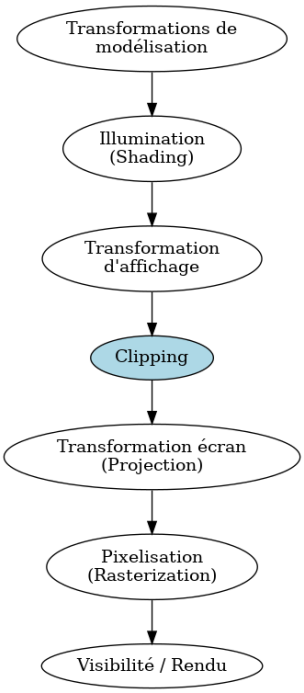


Figure 1: Le Clipping dans le Pipeline Graphique.

Le processus implique souvent :

- Le passage en coordonnées normalisées (NDC - Normalised Device Coordinates) : Les coordonnées des objets sont transformées dans un espace canonique, souvent un cube ou un carré unitaire, indépendant de la résolution de l'écran.
- La suppression des parties hors du volume de vision : Seules les parties de la scène visibles depuis le point de vue de la caméra et situées à l'intérieur du volume de vision défini sont conservées.

*Schéma conceptuel :*
Espace Oeil (Pyramide/Tronc de vision) → Transformation → Espace NDC (Cube [-1,1] ou [0,1])

Figure 2: Passage de l'espace Oeil à l'espace NDC.

Le clipping d'écran est un 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 :

- Carré : Écran standard
- Trapèze : Pare-brise / Rétroviseur
- Circulaire : Lunette / Jumelle

C'est un traitement de base de l'Infographie qui permet d'économiser des opérations inutiles en ne traitant pas les primitives ou parties de primitives invisibles. En 3D, cela permet d'éliminer les calculs en dehors de l'espace visible.

## 1.2 Familles d'algorithmes

Plusieurs familles d'algorithmes existent pour différents types de primitives :

- Clipping Point / Fenêtre
- Clipping Segment / Fenêtre
  - Cohen-Sutherland
  - Liang-Barsky
- Clipping Polygone / Fenêtre
  - Sutherland-Hodgeman

### 1.2.1 Clipping de points : Point / Fenêtre

Le test est très simple : un point $P(x, y)$ est conservé si et seulement s'il se trouve à l'intérieur des limites de la fenêtre rectangulaire définie par $(x_{min}, y_{min})$ et $(x_{max}, y_{max})$.

$$(x_{min} < x < x_{max}) \quad \& \quad (y_{min} < y < y_{max})$$

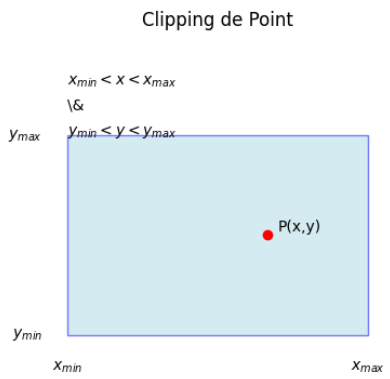


Figure 3: Test de clipping pour un point.

### 1.2.2 Clipping de segments : Segment / Fenêtre

L'objectif est de déterminer les portions de segments de droite qui se trouvent à l'intérieur de la fenêtre de clipping.

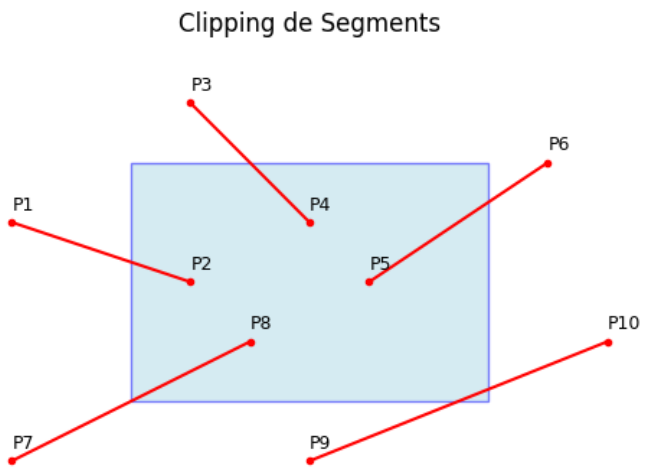


Figure 4: Exemples de segments par rapport à une fenêtre.

**Algorithme de Cohen & Sutherland** Cet algorithme (dû à Ivan Sutherland et collaborateurs) accélère le clipping de segments en utilisant des "codes de zones" (outcodes). L'espace est divisé en 9 régions par les lignes prolongeant les bords de la fenêtre. Chaque région a un code binaire de 4 bits.

- Bit 1 (droite) : 1 si $x > x_{max}$
- Bit 2 (gauche) : 1 si $x < x_{min}$
- Bit 3 (bas) : 1 si $y < y_{min}$
- Bit 4 (haut) : 1 si $y > y_{max}$

Le code pour la fenêtre elle-même est 0000.

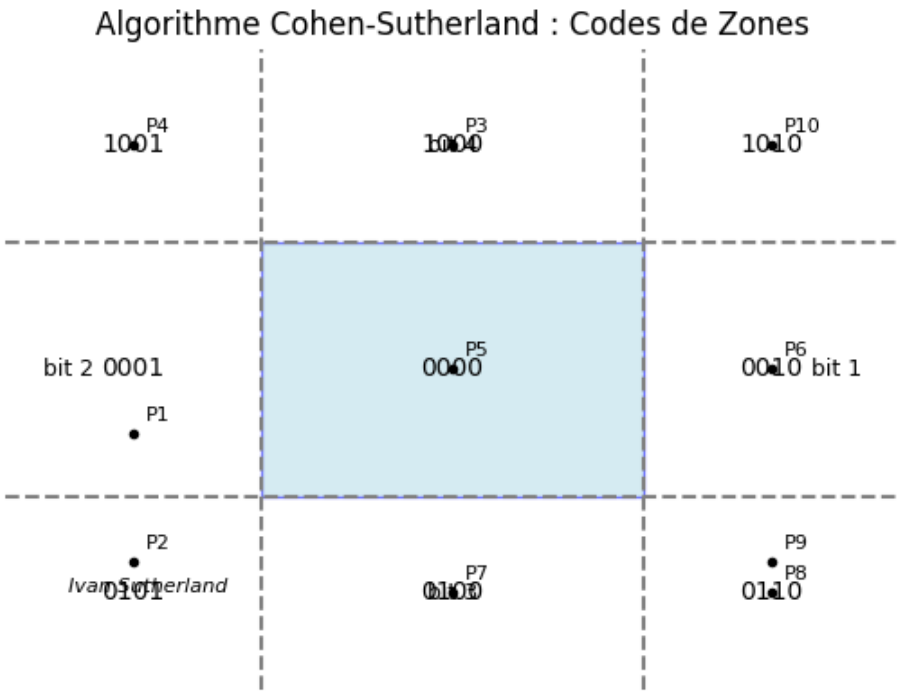


Figure 5: Codes de zones utilisés par l'algorithme de Cohen-Sutherland.

L'algorithme traite chaque segment $[P_1, P_2]$ en calculant les codes $c_1 = \text{code}(P_1)$ et $c_2 = \text{code}(P_2)$. Il y a 4 cas :

- **Cas 1 : Acceptation triviale.** Si $c_1 = 0$ et $c_2 = 0$ (ou $c_1|c_2 = 0$), les deux points sont dans la fenêtre. Le segment est entièrement visible.
- **Cas 3 : Rejet trivial.** Si $c_1 \& c_2 \neq 0$ (ET logique bit à bit), les deux points sont dans la même région extérieure (gauche, droite, haut, ou bas). Le segment est entièrement invisible.
- **Cas 2 : Partiellement dedans.** Si un point est dehors ($c_1 \neq 0$ ou $c_2 \neq 0$) et l'autre est dedans ($c_1 = 0$ ou $c_2 = 0$), le segment coupe la fenêtre. Il faut calculer l'intersection.
- **Cas 4 : Potentiellement dedans.** Si $c_1 \& c_2 = 0$ et aucun des points n'est dans la fenêtre, les extrémités sont dans des zones différentes, mais le segment pourrait traverser la fenêtre. Il faut calculer l'intersection.

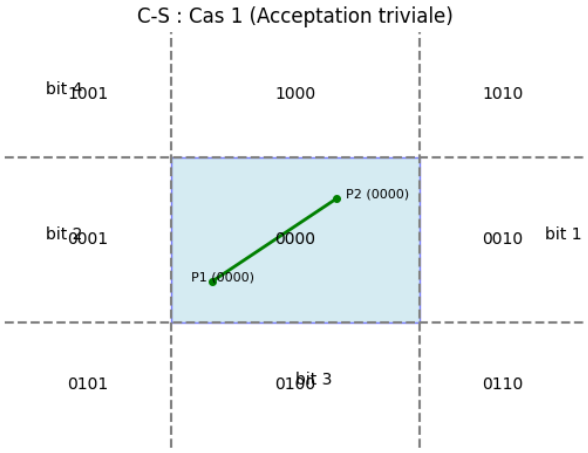


Figure 6: Cas 1 : Segment entièrement dans la fenêtre.

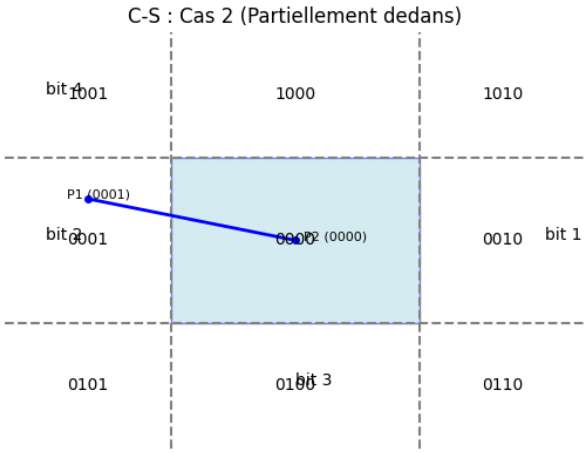


Figure 7: Cas 2 : Segment partiellement dedans.

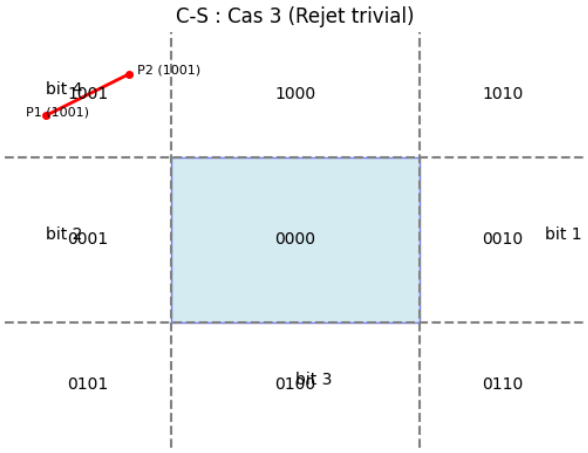


Figure 8: Cas 3 : Segment entièrement dehors.

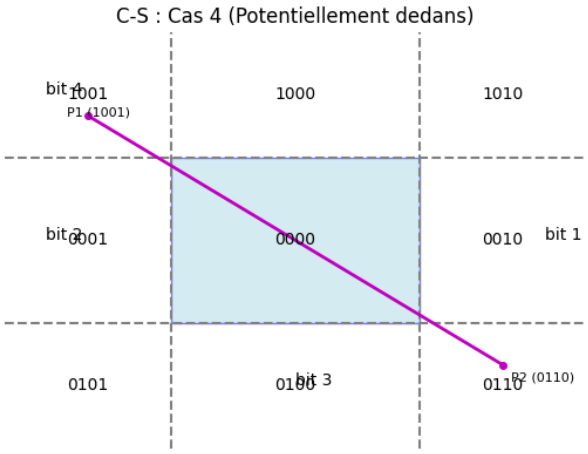


Figure 9: Cas 4 : Extrémités dans des zones différentes.

**Calcul d'intersection (Cas 2 et 4) :** Si un segment n'est ni trivialement accepté ni rejeté, on calcule son intersection avec les bords de la fenêtre. On choisit un point extérieur $P_{ext}(x, y)$ et on calcule l'intersection avec une ligne de clipping (par exemple, $y = y_{max}$). L'équation paramétrique du segment $[P_1(x_1, y_1), P_2(x_2, y_2)]$ est :

$$\begin{aligned} x(t) &= (1-t)x_1 + tx_2 \\ y(t) &= (1-t)y_1 + ty_2 \quad \text{avec } t \in [0,1] \end{aligned}$$

Pour l'intersection avec $y = y_{max}$ :

$$(1-t)y_1 + ty_2 = y_{max} \implies t = \frac{y_{max} - y_1}{y_2 - y_1}$$

On calcule ensuite le $x$ correspondant : $x = (1-t)x_1 + tx_2$. Le point $(x, y_{max})$ remplace $P_{ext}$. On recalcule le code du nouveau point et on réitère le test (Cas 1, 2, 3, 4) avec le segment réduit. Ce processus est répété jusqu'à ce que le segment soit trivialement accepté ou rejeté.

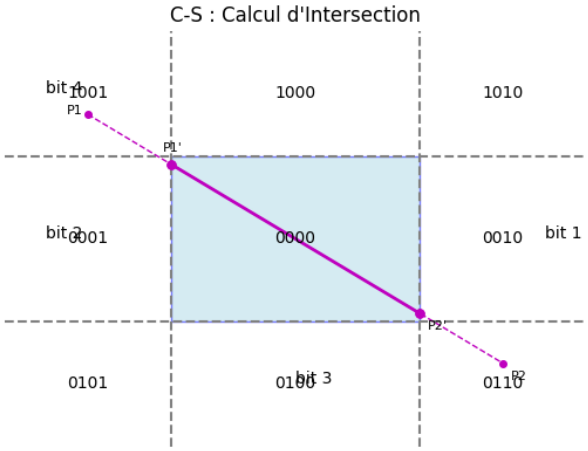


Figure 10: Calcul des intersections et réitération des tests.

**Extension 3D :** L'algorithme se généralise en 3D avec 6 bits pour le code de zone (outcode) :

- Bit 1: $x > x_{max}$ (RIGHT)
- Bit 2: $x < x_{min}$ (LEFT)
- Bit 3: $y > y_{max}$ (TOP)
- Bit 4: $y < y_{min}$ (BOTTOM)
- Bit 5: $z > z_{max}$ (FAR)
- Bit 6: $z < z_{min}$ (NEAR)

Cela nécessite des pré-calculs supplémentaires pour les intersections avec les 6 plans du volume de vue.

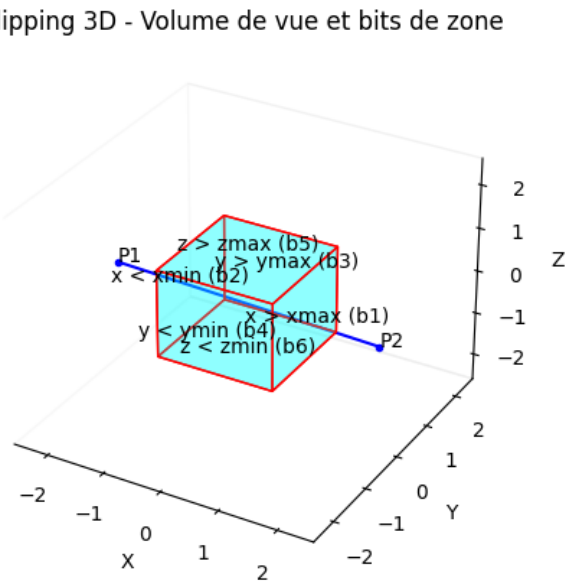


Figure 11: Extension 3D de Cohen-Sutherland avec 6 bits.

**Algorithme de Liang & Barsky** Cette approche découpe une droite en exploitant sa forme paramétrique $P(\alpha) = (1-\alpha)P_1 + \alpha P_2$.

$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$
$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2$$

Le paramètre $\alpha$ indique la position sur la droite :

- $\alpha \in [0,1]$ : $P(\alpha)$ est sur le segment $[P_1, P_2]$.
- $\alpha < 0$ : $P(\alpha)$ est sur la demi-droite à gauche de $P_1$.
- $\alpha > 1$ : $P(\alpha)$ est sur la demi-droite à droite de $P_2$.

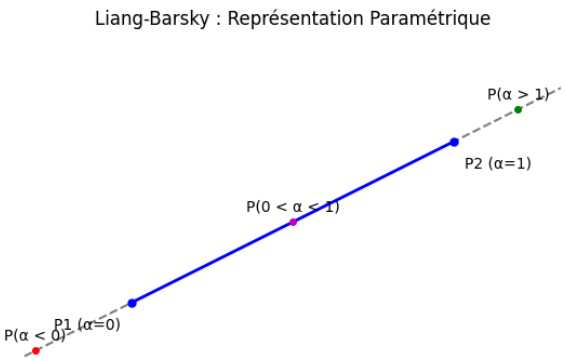


Figure 12: Signification du paramètre $\alpha$ dans l'algorithme de Liang-Barsky.

L'algorithme trouve les intersections de la droite infinie $P(\alpha)$ avec les quatre lignes de la fenêtre ($x = x_{min}, x = x_{max}, y = y_{min}, y = y_{max}$). Cela donne jusqu'à quatre valeurs de $\alpha$. Ces valeurs définissent l'intervalle $[\alpha_{min}, \alpha_{max}]$ correspondant à la partie visible du segment. Initialement, $[\alpha_{min}, \alpha_{max}] = [0, 1]$. Deux cas peuvent se présenter pour les intersections avec les bords de la fenêtre :

- La droite n'est parallèle à aucun côté de la fenêtre : 4 intersections potentielles.
- La droite est parallèle à un des côtés de la fenêtre : 2 intersections potentielles.

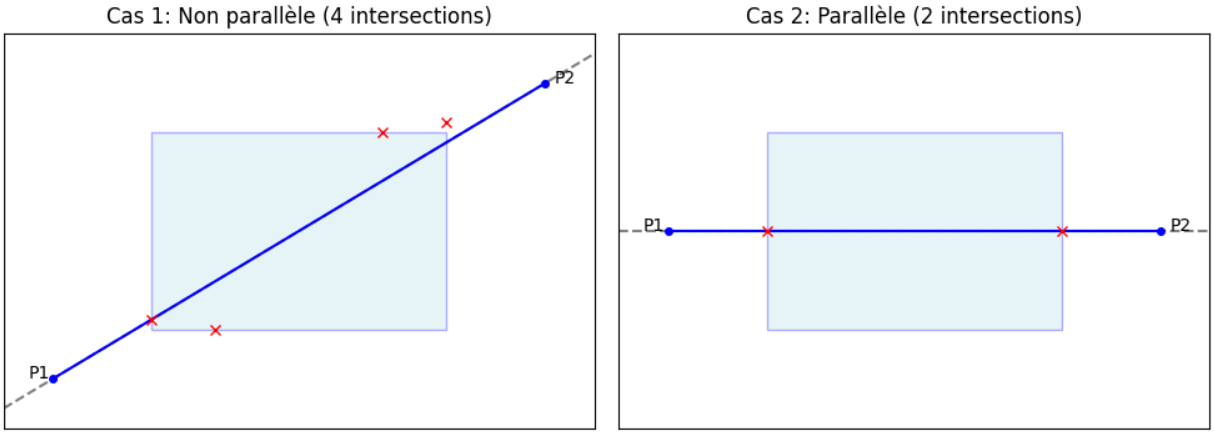


Figure 13: Cas d'intersection dans l'algorithme de Liang-Barsky.

**Exemple d'exécution :** Considérons le segment $P_0 = (30, 20)$ à $P_1 = (280, 160)$ et une fenêtre [70, 230] x [60, 150]. L'algorithme teste les intersections avec chaque bord, mettant à jour l'intervalle $[\alpha_{min}, \alpha_{max}]$ (initialement [0, 1]).

*(Note : Les diagrammes suivants illustrent le processus, mais les calculs exacts ne sont pas détaillés ici.)*

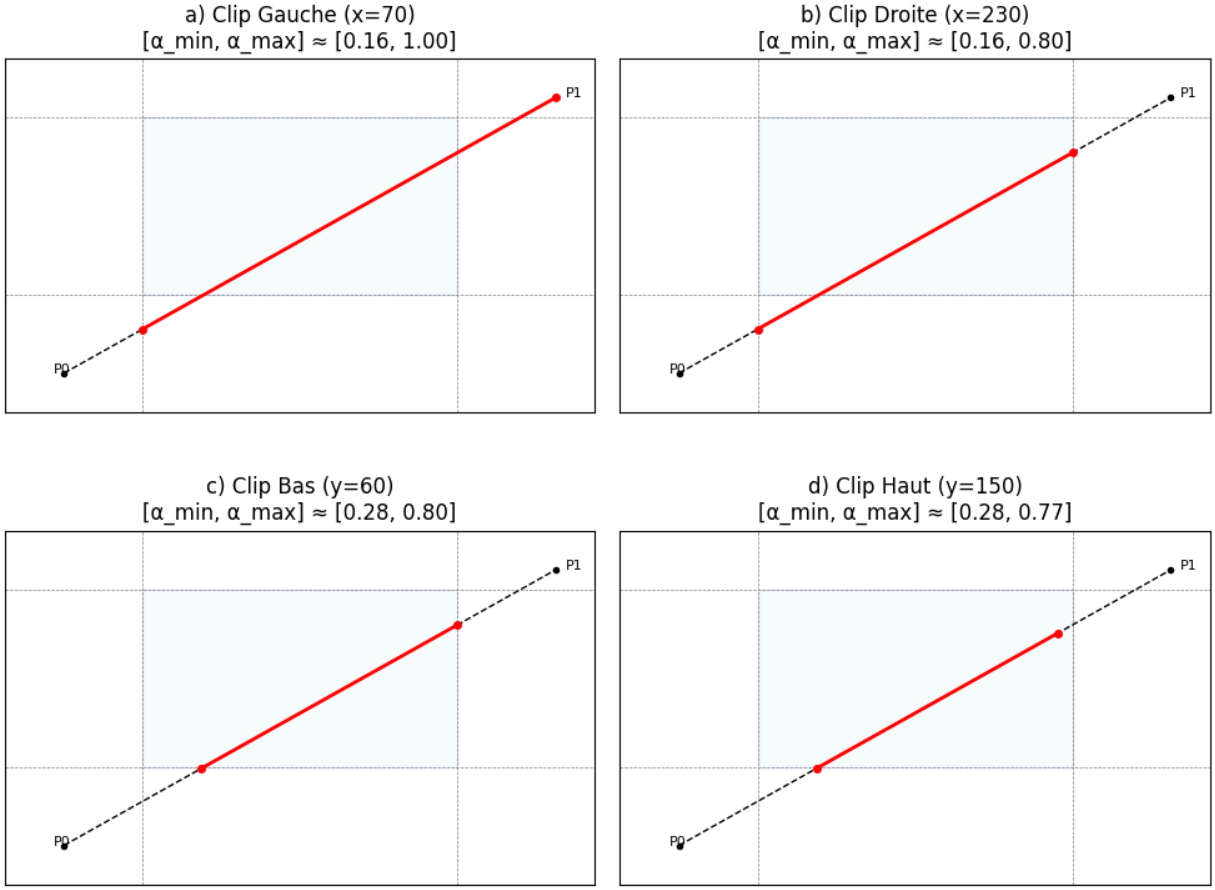


Figure 14: Exemple de clipping successif avec Liang-Barsky.

L'ordre d'intersection avec les frontières peut varier (ex: Bas, Gauche, Haut, Droite). L'algorithme calcule les 4 paramètres d'intersection $\alpha_1, \alpha_2, \alpha_3, \alpha_4$. Il trie ensuite ces paramètres et les compare avec l'intervalle [0, 1] pour déterminer la portion visible finale $[\alpha_{vis\_min}, \alpha_{vis\_max}]$. Si $\alpha_{vis\_min} < \alpha_{vis\_max}$, le segment est (partiellement) visible, sinon il est rejeté.

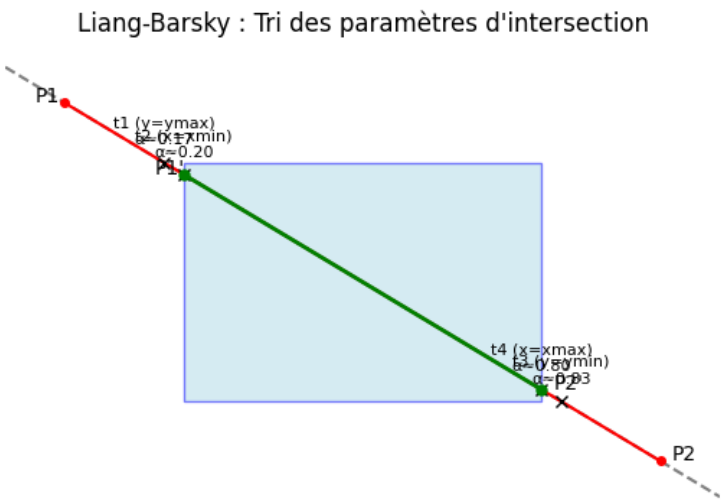


Figure 15: Ordre des intersections et détermination du segment visible.

### 1.2.3 Clipping de polygones : Polygone / Fenêtre

L'objectif est de déterminer la ou les parties d'un polygone qui se trouvent à l'intérieur de la fenêtre de clipping. Le résultat peut être un ou plusieurs polygones.

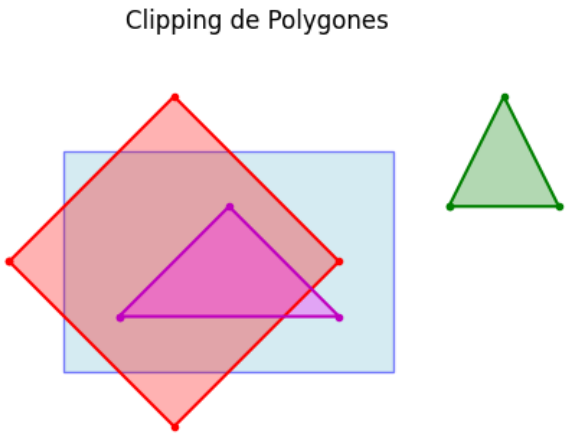


Figure 16: Exemples de polygones par rapport à une fenêtre.

**Algorithme de Sutherland & Hodgeman** Cet algorithme clippe un polygone en le testant successivement contre chaque frontière (bord) de la fenêtre de clipping (Gauche, Droite, Bas, Haut). Pour chaque frontière, l'algorithme prend une liste de sommets en entrée (le polygone original ou le résultat du clipping précédent) et produit une nouvelle liste de sommets en sortie.

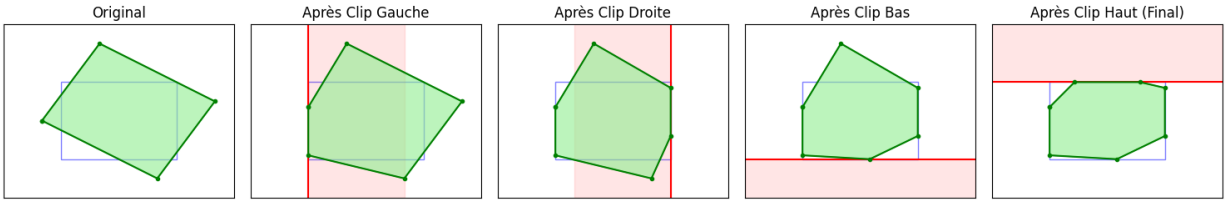


Figure 17: Étapes successives du clipping avec Sutherland-Hodgeman.

Pour chaque frontière et chaque arête $(P_i, P_{i+1})$ du polygone en entrée :

- Tester l'appartenance des extrémités $P_i$ et $P_{i+1}$ par rapport à la frontière (dedans ou dehors).
- Il y a 4 cas pour l'arête $(P_i, P_{i+1})$:
  1. $P_i$ dedans, $P_{i+1}$ dedans : Garder $P_{i+1}$.
  2. $P_i$ dedans, $P_{i+1}$ dehors : Calculer l'intersection $P'$, garder $P'$. (Cas 2)
  3. $P_i$ dehors, $P_{i+1}$ dehors : Ne rien garder.
  4. $P_i$ dehors, $P_{i+1}$ dedans : Calculer l'intersection $P'$, garder $P'$ puis $P_{i+1}$. (Cas 4)
- Insérer les nouveaux points calculés (intersections) et les points conservés dans la liste de sortie.
- Supprimer les points en dehors de la fenêtre.

La liste de sortie d'une étape devient l'entrée de l'étape suivante. Le résultat final est le polygone clippé.

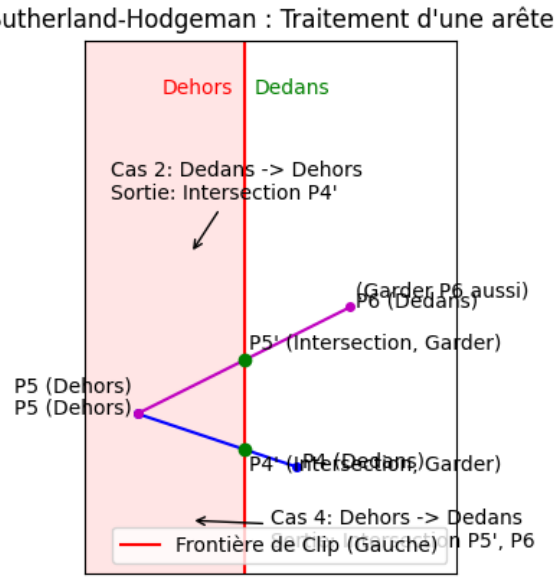


Figure 18: Détail du traitement d'une arête par Sutherland-Hodgeman (Cas 2 et 4).

# 2 Élimination des Parties Cachées (Visibilité / Rendu)

## 2.1 Introduction

L'élimination des parties cachées (ou détermination de la surface visible, HSR - Hidden Surface Removal) est une étape cruciale du rendu 3D. Elle consiste à déterminer quelles lignes, arêtes, surfaces ou volumes d'une scène 3D sont visibles depuis le point de vue de l'observateur (caméra).

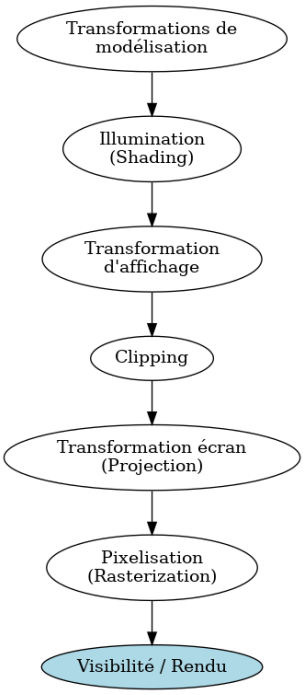


Figure 19: La Visibilité / Rendu dans le Pipeline Graphique.

Le but est d'assurer la cohérence visuelle de la scène et de réduire le nombre de primitives traitées par le pipeline graphique, en ne dessinant que ce qui est réellement visible. Cela implique souvent de remplir le framebuffer (tampon d'image) avec la bonne couleur pour chaque pixel, en tenant compte de l'occultation.

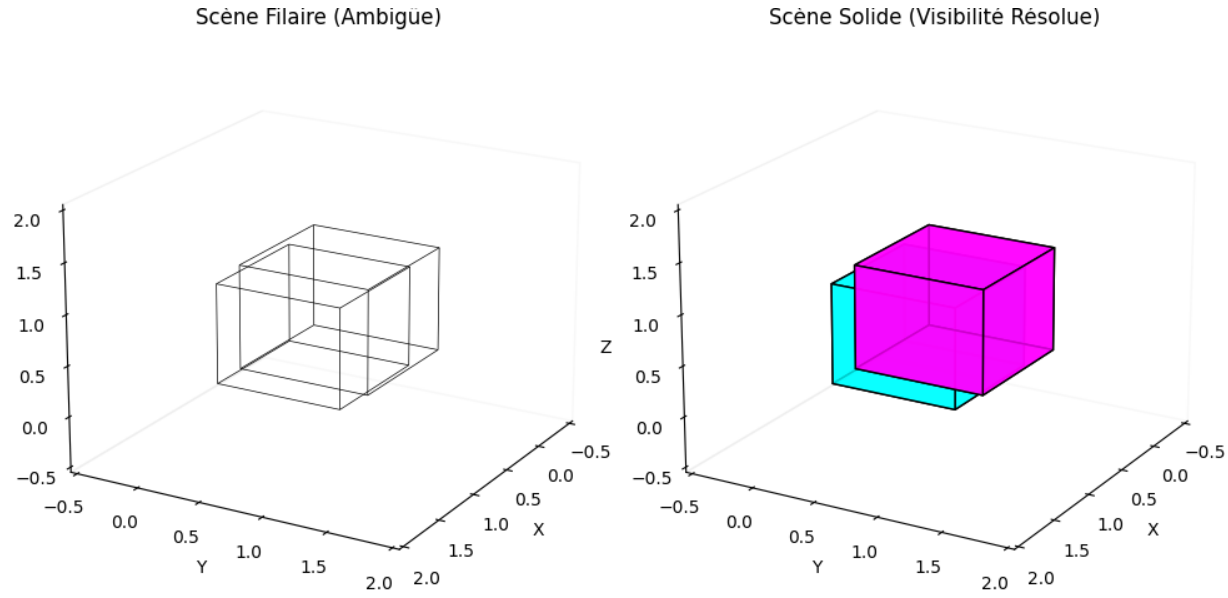


Figure 20: Objectif de l'élimination des parties cachées.

## 2.2 Familles d'algorithmes

Il existe deux grandes familles d'algorithmes HSR :

- **Algorithmes Objet (Espace Scène) :**
  - Le masquage se fait sur le modèle de données physiques (calcul en coordonnées du monde).
  - La détermination de la visibilité est valable quelle que soit l'échelle du dessin (résolution de l'image).
  - Caractéristiques : Test objet par objet (potentiellement $O(n^2)$), précision dépend de la résolution des objets, techniques souvent plus complexes à mettre en œuvre.
  - Méthodes : Backface Culling, Algorithme du peintre, Arbres BSP.
- **Algorithmes Image (Espace Image) :**
  - Le masquage se fait au niveau des pixels de l'écran (calcul en coordonnées fenêtre).
  - Caractéristiques : Test de visibilité en chaque pixel (souvent $O(n \times \text{pixels})$), précision dépend de la résolution de l'espace image, recalcul nécessaire si la caméra ou la scène change, techniques souvent plus simples.
  - Méthodes : Warnock, Ligne de balayage (Scan-line Watkins, Scan-line Z-buffer), Test de profondeur (Z-buffer).

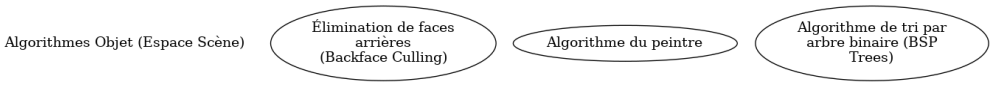


Figure 21: Principaux algorithmes HSR dans l'espace objet.

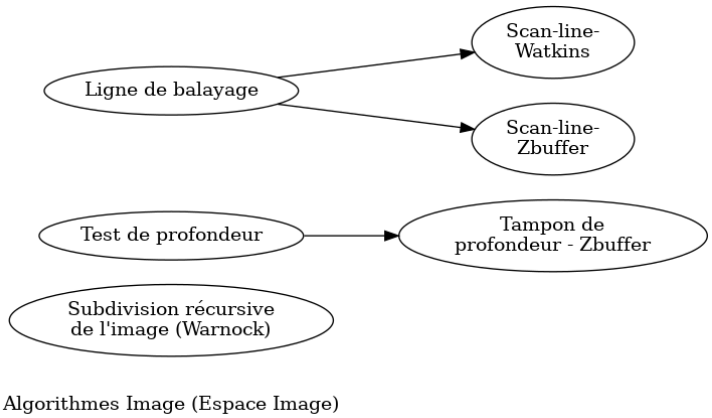


Figure 22: Principaux algorithmes HSR dans l'espace image.

### 2.2.1 Algorithmes Objet (Espace Scène)

**Élimination des Faces Arrières (Backface Culling)**

**Definition 2.1.** Principe : Garder uniquement les parties de la surface (facettes) dont la normale pointe *vers* l'observateur (ou dans une direction opposée au vecteur de vue). Les faces dont la normale pointe à l'opposé de l'observateur sont considérées comme "arrière" et sont éliminées.

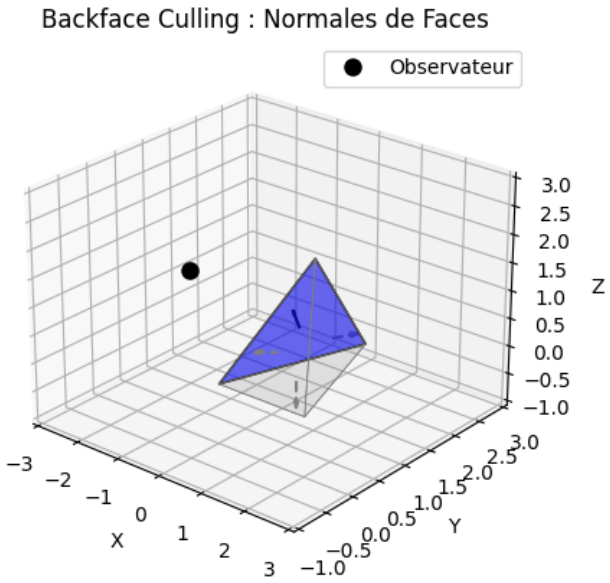


Figure 23: Principe du Backface Culling.

**Remark 2.2.** Conditions : La suppression des faces arrières suffit à éliminer les parties cachées si :

- L'objet est seul dans la scène (pas d'occultation par d'autres objets).
- L'objet est convexe. Pour un objet concave, des faces avant peuvent être masquées par d'autres parties de l'objet lui-même.

**Definition 2.3.** Calcul : On utilise le produit scalaire entre la normale à la face $\vec{N}$ et le vecteur allant d'un point de la face (ou son centre) vers l'oeil (ou la caméra) $\vec{V} = \text{Oeil} - \text{Face}$.

- Si $\vec{N} \cdot \vec{V} > 0$ : La face est visible (orientée vers l'oeil). On la garde. *(Note: La diapo dit $\leq 0$ on garde, ce qui implique que V va de l'oeil vers la face ou que N est la normale intérieure. En suivant la convention usuelle Oeil-¿Face et normale extérieure, N.V ¿ 0 est visible. Si V = Face - Oeil, N.V ¡ 0 est visible. La diapo utilise (Oeil - Face) . Normale $\leq 0$ =¿ on garde. Adoptons cette convention.)*
- Si $(\text{Oeil} - \text{Face}) \cdot \vec{N} \leq 0$ : On garde le polygone.
- Si $(\text{Oeil} - \text{Face}) \cdot \vec{N} > 0$ : On élimine le polygone (face arrière).

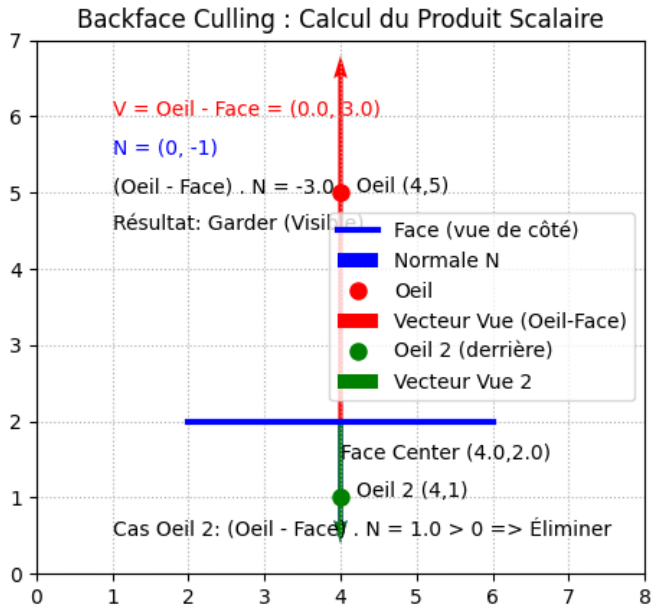


Figure 24: Calcul pour déterminer si une face est visible.

**Remark 2.4.** Avantages :

- Économise en moyenne 50% du temps de calcul (pour des objets fermés).
- Faible coût par polygone.
- Souvent utilisé comme étape préliminaire pour d'autres algorithmes HSR plus complexes.

### Algorithme du Peintre (Depth-Sort)

**Definition 2.5.** Aussi appelé "Algorithme de Tri par la Profondeur" (Newell, Newell & Sancha 1972). Principe :

1. Trier les polygones de la scène en fonction de leur distance à l'observateur (profondeur, souvent coordonnée Z après projection). On trie du plus éloigné au plus proche.
2. Peindre (dessiner) les polygones dans la mémoire vidéo (framebuffer) dans cet ordre : du plus loin au plus près.

Ainsi, les polygones plus proches recouvrent correctement les polygones plus éloignés qu'ils occultent.

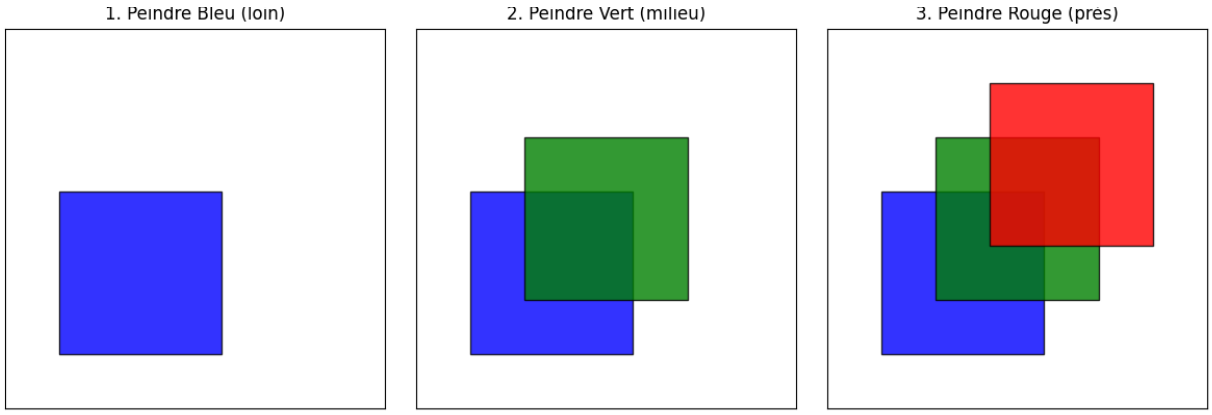


Figure 25: Principe de l'algorithme du peintre : peindre du fond vers l'avant.

**Remark 2.6.** Cas ambigu (chevauchement) : Le tri simple basé sur une seule valeur de profondeur (ex: Zmax) n'est pas toujours suffisant. Des polygones peuvent avoir des étendues en Z qui se chevauchent, ou des dépendances cycliques.

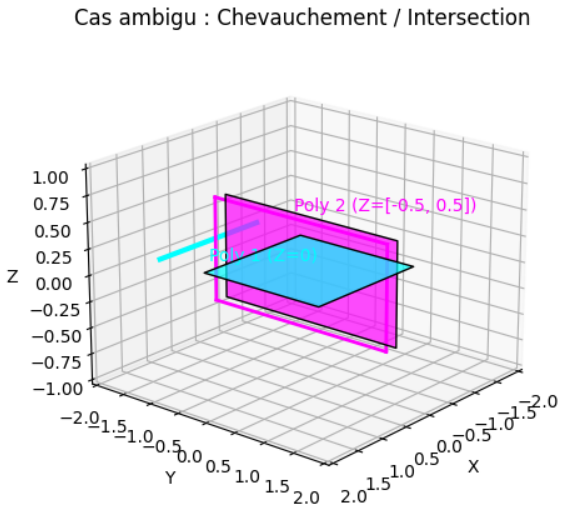


Figure 26: Exemple de cas ambigu pour l'algorithme du peintre.

**Remark 2.7.** Résolution des cas ambigus :

- Trier les polygones en fonction de leur plus grande coordonnée en Z (distance max à la caméra).
- Si deux polygones P et Q ont des étendues en Z qui se recouvrent ($[Z^P_{min}, Z^P_{max}]$ et $[Z^Q_{min}, Z^Q_{max}]$ overlap) :
  - Test des boîtes englobantes : Si les projections 2D (XY) des boîtes englobantes sont séparées, l'ordre n'importe pas.

– Test d'orientation : Tester si P est entièrement "derrière" Q ou vice-versa par rapport au plan de l'autre polygone.

– Test de projection : Si les projections 2D se chevauchent, vérifier si P occulte Q ou Q occulte P.

– Si rien ne marche (intersection ou dépendance cyclique) : Couper l'un des polygones (ou les deux) par le plan de l'autre. On obtient de nouveaux polygones plus petits qu'il faut réinsérer dans la liste triée.

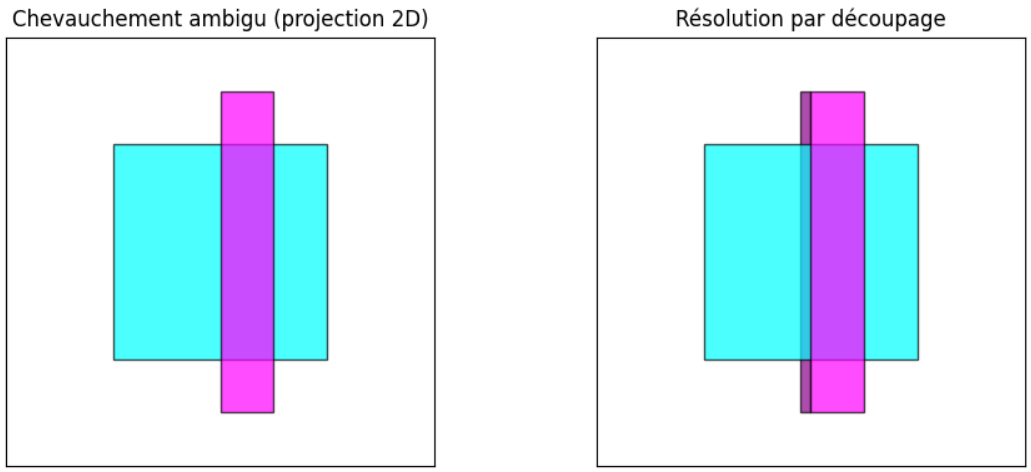


Figure 27: Résolution des ambiguïtés par découpage de polygones.

**Remark 2.8.** Caractéristiques :

- Le plus intuitif des algorithmes.
- Coût en mémoire : Peut nécessiter de stocker tous les polygones. L'affichage direct à l'écran évite un grand buffer ($O(p)$ où p est le nombre de polygones).
- Coût de calcul : Dominé par le tri ($O(n \log n)$ ou $O(n^2)$ en cas de tests complexes) et potentiellement le découpage.
- Efficace surtout sur des petites scènes où les découpages sont rares.

### Partition Binaire de l'Espace (BSP-Trees)

**Definition 2.9.** Un arbre BSP (Binary Space Partition) est un arbre binaire utilisé pour trier des primitives (polygones) dans l'espace 3D. Il permet une détermination efficace de la visibilité.

### Principe (Construction de l'arbre - Schumaker 1969, Fuchs 1980)

1. Choisir un polygone (ou une face) de la scène comme plan séparateur (nœud racine).
2. Ce plan divise l'espace en deux demi-espaces : "avant" (in front) et "arrière" (behind) par rapport à l'orientation du plan/polygone.
3. Répartir tous les autres polygones de la scène dans ces deux demi-espaces :
   - Ceux entièrement "devant" vont dans le sous-arbre "avant" (fils droit/gauche).

- Ceux entièrement "derrière" vont dans le sous-arbre "arrière" (fils gauche/droit).
- Ceux qui sont coupés ("à cheval") par le plan sont divisés en deux nouveaux polygones, chacun étant placé dans le demi-espace correspondant.

4. Réitérer le processus récursivement sur les ensembles de polygones dans chaque demi-espace, jusqu'à ce que chaque feuille de l'arbre contienne au plus un polygone (ou fragment).

*Schémas conceptuels (voir slides 42-47) :*
1. Ensemble initial d'objets.
2. Choix d'un plan (associé à un objet/polygone) → Nœud racine.
3. Répartition des objets : 'avant' (P - proche/positif) / 'arrière' (L - loin/négatif).
4. Objet coupé ('Ouch') → Division en deux, répartition.
5. Répétition récursive pour les sous-ensembles avant/arrière → Construction de l'arbre.

Figure 28: Illustration conceptuelle de la construction d'un arbre BSP.

**Utilisation pour le rendu (Parcours de l'arbre)** Le parcours de l'arbre BSP pour le rendu dépend de la position de l'observateur par rapport aux plans séparateurs des nœuds. L'ordre de parcours garantit que les objets sont dessinés du plus loin au plus près (similaire à l'algorithme du peintre, mais sans tri explicite à chaque frame).

**Algorithme de rendu récursif ('renderBSP(node, eye$_pos$)') :**

1. Si 'node' est vide (feuille), retourner.
2. Déterminer si 'eye$_p$os'est"devant"ou"derrière"leplandu'node'.Si'eye$_p$os'estdevant :
3. 4. 'Proche = node-¿avant'
   5. 'Loin = node-¿arriere'

Sinon ('eye$_p$os'est$derrière$) :
'Proche = node-¿arriere'

'Loin = node-¿avant'

Appeler récursivement 'renderBSP(Loin, eye$_p$os)'($traiterlesous - arbrelointaind'abord$).$Dessinerlaprimitive$($polygone$)$associéeau'node'courant$.$

Appeler récursivement 'renderBSP(Proche, eye$_p$os)'($traiterlesous - arbreprocheensuite$).

Listing 1: Pseudo-code du rendu avec un arbre BSP

```
renderBSP(BSPtree *T, Point eye_pos) {
  if (T == NULL) return; // Feuille vide

  BSPtree *Proche, *Loin;
  Plane plane = T->plane; // Plan du noeud courant

  // Determiner si l'oeil est devant ou derriere le plan
  if (is_in_front(eye_pos, plane)) {
    Proche = T->front_child; // Sous-arbre avant
    Loin = T->back_child;   // Sous-arbre arriere
  } else {
    Proche = T->back_child;   // Sous-arbre arriere
    Loin = T->front_child; // Sous-arbre avant
  }
```

```
    // 1. Traiter le sous-arbre lointain
    renderBSP(Loin, eye_pos);

    // 2. Dessiner la primitive du noeud courant
    if (T->polygon != NULL) { // Verifier si c'est un noeud interne ou
        ↪ feuille avec polygone
        renderObject(T->polygon);
    }

    // 3. Traiter le sous-arbre proche
    renderBSP(Proche, eye_pos);
}
```

*Schémas conceptuels (voir slides 49-64) :*
1. Position de l'observateur (Oeil) par rapport aux plans.
2. Parcours récursif : Loin → Dessin Nœud → Proche.
3. L'ordre de dessin final dépend du point de vue (Viewpoint A vs Viewpoint B).
4. Résultat : Scène rendue avec occultation correcte.

Figure 29: Illustration conceptuelle du rendu avec un arbre BSP.

### 2.2.2 Algorithmes Image (Espace Image)

Ces algorithmes opèrent au niveau des pixels dans l'espace image (fenêtre/écran).

**Remark 2.10.** Caractéristiques :

- Test de visibilité effectué pour chaque pixel.
- Précision limitée par la résolution de l'image.
- Doit être recalculé si la caméra ou la scène change.
- Souvent plus simples à implémenter que les algorithmes objet complexes.

**Méthodes principales (voir Figure 22) :**

- **Z-buffer (Tampon de profondeur)** : Le plus commun. Maintient un tampon stockant la profondeur (Z) du pixel visible le plus proche pour chaque pixel de l'écran. Chaque polygone est rastérisé (converti en pixels), et chaque pixel généré n'est écrit dans le framebuffer que si sa profondeur est inférieure à celle déjà stockée dans le Z-buffer.
- **Algorithmes Scan-line** : Traitent l'image ligne par ligne. Pour chaque ligne de balayage, ils déterminent les segments visibles en calculant les intersections des polygones avec la ligne et en gérant les informations de profondeur le long de la ligne. (Watkins, Scan-line Z-buffer).
- **Algorithme de Warnock** : Algorithme récursif de subdivision de l'image. Divise récursivement les régions de l'écran jusqu'à ce que la visibilité dans une région soit simple à déterminer (ex: région vide, couverte par un seul polygone, etc.).