1 Introduction : Élimination des Parties Cachées

Lors de la création d'images de synthèse tridimensionnelles, plusieurs objets peuvent occuper la même position sur l'écran (pixel). Le problème de l'élimination des parties cachées consiste à déterminer quelles parties de quels objets sont visibles depuis le point de vue de l'observateur (la caméra) afin de ne dessiner que celles-ci. Sans ce processus, les objets les plus éloignés pourraient être dessinés par-dessus les objets plus proches, créant une image incorrecte. Plusieurs approches algorithmiques existent pour résoudre ce problème. On les classe souvent en deux catégories : les algorithmes opérant dans l'espace objet et ceux opérant dans l'espace image. Ce chapitre se concentre sur les **algorithmes images**, qui déterminent la visibilité au niveau de chaque pixel de l'image finale.

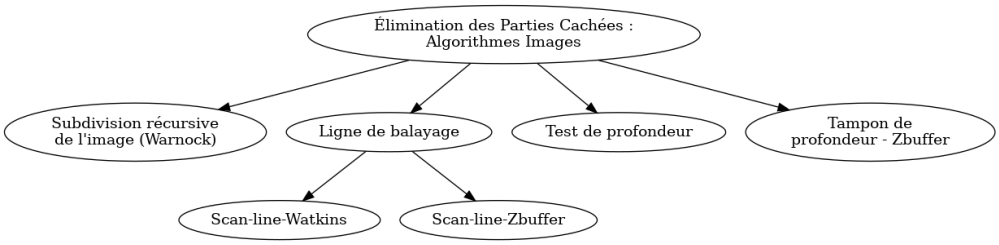

Figure 1: Classification des algorithmes d'élimination des parties cachées opérant dans l'espace image.

Parmi les algorithmes images, nous allons détailler l'algorithme de subdivision récursive de l'image (Warnock) et l'algorithme du tampon de profondeur (Z-buffer). Les algorithmes basés sur la ligne de balayage (Scan-Line), tels que ceux de Watkins ou utilisant un Z-buffer par ligne, seront brièvement évoqués.

2 Subdivision Récursive de l'Image : Algorithme de Warnock

L'algorithme de Warnock, développé par John Warnock, est un algorithme d'élimination des parties cachées basé sur le principe "diviser pour régner" (divide and conquer). L'idée est de subdiviser récursivement l'image en quadrants (structure de quadtree) jusqu'à ce que le contenu de chaque quadrant soit suffisamment simple pour être affiché directement.

2.1 Principe "Diviser pour Régner"

L'algorithme commence avec la fenêtre d'affichage entière.

1. On examine le contenu du quadrant courant.
2. Si le contenu est simple (voir cas simples ci-dessous), on affiche le quadrant et on arrête la subdivision pour cette branche.
3. Si le contenu est complexe, on subdivise le quadrant en quatre sous-quadrants de taille égale.
4. On applique récursivement l'algorithme à chacun des sous-quadrants.

Ce processus est illustré ci-dessous, montrant les étapes successives de subdivision de l'image.

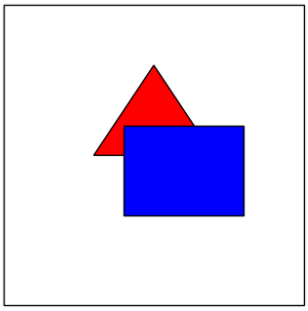

Figure 2: Image initiale avec deux polygones.

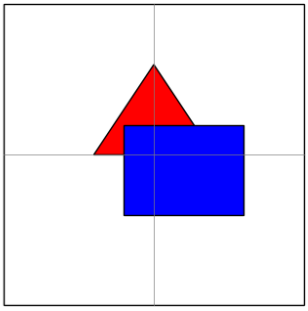

Figure 3: Première subdivision en quadtree.

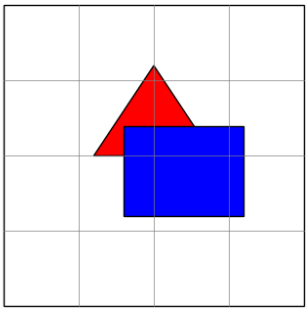

Figure 4: Deuxième niveau de subdivision dans les quadrants complexes.

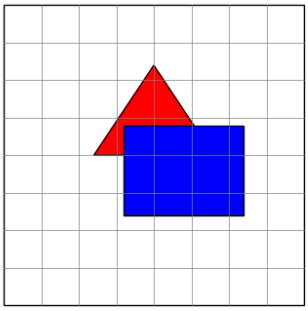

Figure 5: Troisième niveau de subdivision.

2.2 Traitement des Quadrants

Pour chaque quadrant, l'algorithme détermine la liste des polygones potentiellement visibles (ceux qui intersectent le quadrant). Ensuite, il analyse la situation :

2.2.1 Cas Simples

Le contenu du quadrant est considéré comme simple et peut être tracé directement dans les cas suivants :

- **Rien dans le quadrant :** Le quadrant est vide. On le remplit avec la couleur du fond.
- **Un seul polygone :**
  - Si le polygone **couvre totalement** le quadrant, on remplit le quadrant avec la couleur du polygone (éventuellement après calcul d'ombrage).
  - Si le polygone **couvre partiellement** le quadrant, on remplit le quadrant avec la couleur du fond, puis on dessine la partie du polygone contenue dans le quadrant. (Note : une approche alternative est de subdiviser même si un seul polygone est partiellement dedans, mais le cas simple ici suggère de le dessiner directement).
  - Si le polygone est **contenu** dans le quadrant, on le dessine avec sa couleur sur le fond.

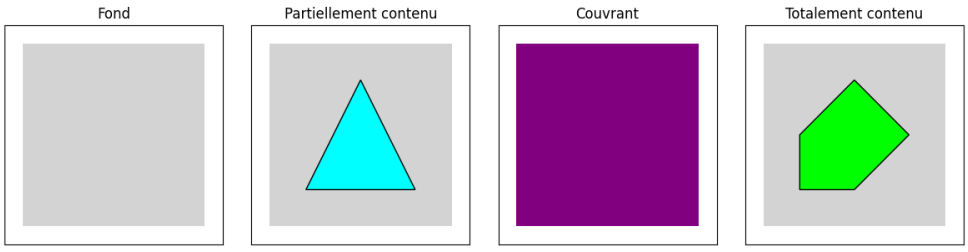

Figure 6: Cas simples de traitement d'un quadrant dans l'algorithme de Warnock.

2.2.2 Cas Complexes : Plusieurs Polygones

Si plusieurs polygones sont présents dans le quadrant, la situation est complexe et nécessite une analyse plus poussée ou une subdivision :

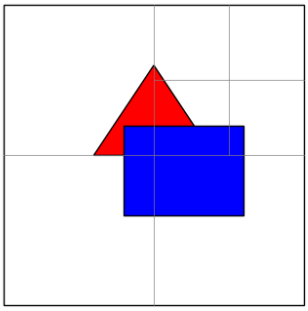

Figure 7: Cas complexe : quadrant contenant plusieurs polygones.

1. **Trier les polygones :** Les polygones dans le quadrant sont triés selon leur profondeur (distance à l'observateur).
2. **Vérifier le masquage complet :** On vérifie si le polygone le plus proche (après tri) masque complètement tous les autres polygones à l'intérieur du quadrant.
    - Si un polygone P couvre entièrement le quadrant et est plus proche que tous les autres polygones intersectant le quadrant, alors on remplit le quadrant avec la couleur de P. Ce test peut parfois être effectué sans calculs complexes de profondeur si P est clairement devant les autres.
3. **Subdiviser :** Si aucun polygone ne masque entièrement les autres de manière simple, on subdivise le quadrant en quatre sous-quadrants et on réitère le processus pour chacun.

2.2.3 Critère d'Arrêt

La subdivision récursive s'arrête lorsque le quadrant devient plus petit qu'un pixel. À ce niveau de résolution :

- On calcule la profondeur de chaque polygone intersectant le pixel (au centre du pixel, par exemple).
- On détermine le polygone le plus proche de l'observateur.
- On affiche le pixel avec la couleur de ce polygone le plus proche.

2.3 Découpage Récursif et Problèmes Potentiels

Le découpage récursif continue jusqu'à ce qu'une décision puisse être prise pour chaque région.

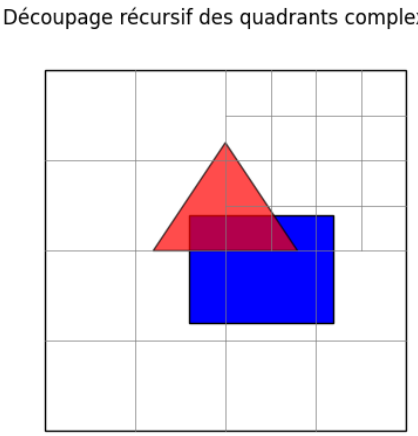

Figure 8: Exemple de découpage récursif pour résoudre les cas complexes.

Un problème peut survenir si les tests simples de masquage échouent à détecter une situation de masquage réel. Par exemple, un polygone S1 peut masquer entièrement d'autres polygones S2, S3, S4 à l'intérieur d'un quadrant donné, mais si S1 ne remplit pas *entièrement* le quadrant, les tests simples (comme vérifier si un polygone couvre le quadrant) peuvent ne pas le détecter. Le test de profondeur au niveau du pixel résout ce problème, mais le but de la subdivision est d'éviter ces calculs coûteux autant que possible.

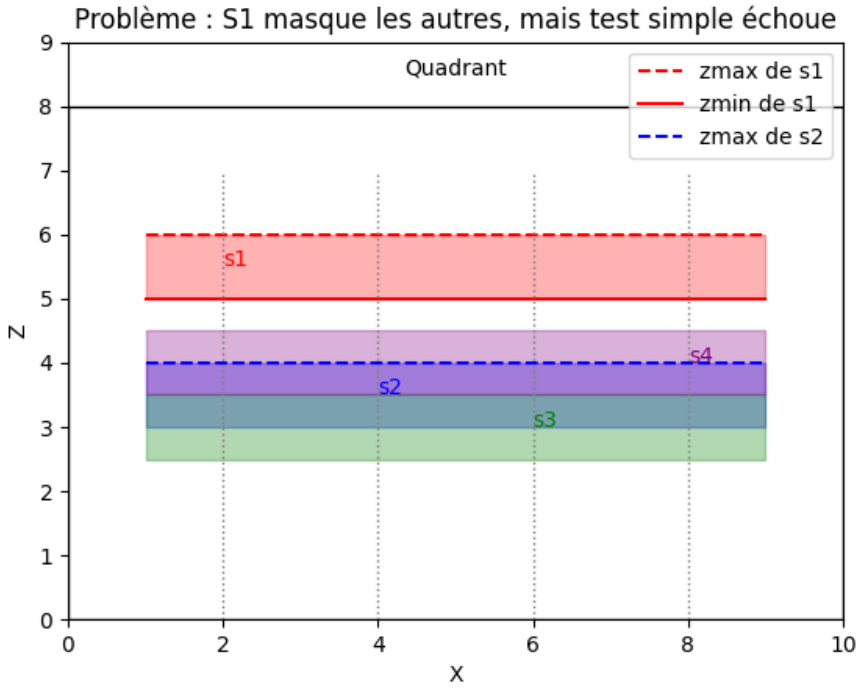

Figure 9: Illustration du problème où un polygone S1 masque d'autres polygones (S2, S3, S4) dans le quadrant selon l'axe Z, mais les tests simples peuvent ne pas le détecter si S1 ne couvre pas entièrement le quadrant.

Dans ce cas (Figure 9), la subdivision est nécessaire car le test simple (S1 couvre-t-il le quadrant ?) échoue, même si S1 est effectivement le seul polygone visible dans cette zone.

3 Algorithme de Balayage (Scan-Line) - Aperçu

Les algorithmes de balayage traitent l'image ligne par ligne (horizontalement, le long de l'axe y). Le principe général est d'éliminer les parties cachées pour chaque ligne de balayage individuellement.

- **Principe :** Pour une ligne de balayage donnée (y constant), on détermine tous les segments de polygones qui intersectent cette ligne.
- **Visibilité :** On calcule ensuite, pour chaque segment le long de la ligne (en x), quel segment est le plus proche de l'observateur (en utilisant la coordonnée z).
- **Affichage :** Seuls les segments visibles sont affichés sur la ligne de balayage courante.

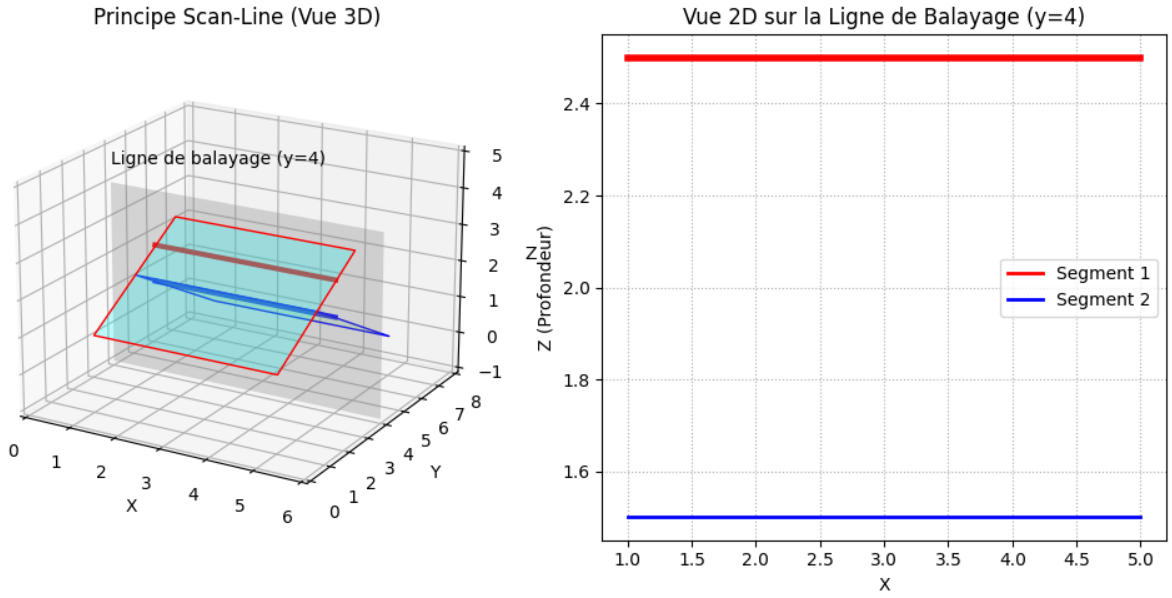

Figure 10: Principe de l'algorithme de balayage (Scan-Line). La scène 3D est coupée par un plan correspondant à la ligne de balayage courante (gauche). Les intersections forment des segments dans le plan X-Z (droite), où la visibilité est déterminée par la profondeur Z.

Pour une ligne de balayage donnée, plusieurs cas peuvent se présenter lors de la comparaison des segments issus de différents polygones. La gestion des intervalles de segments et de leur profondeur est cruciale.

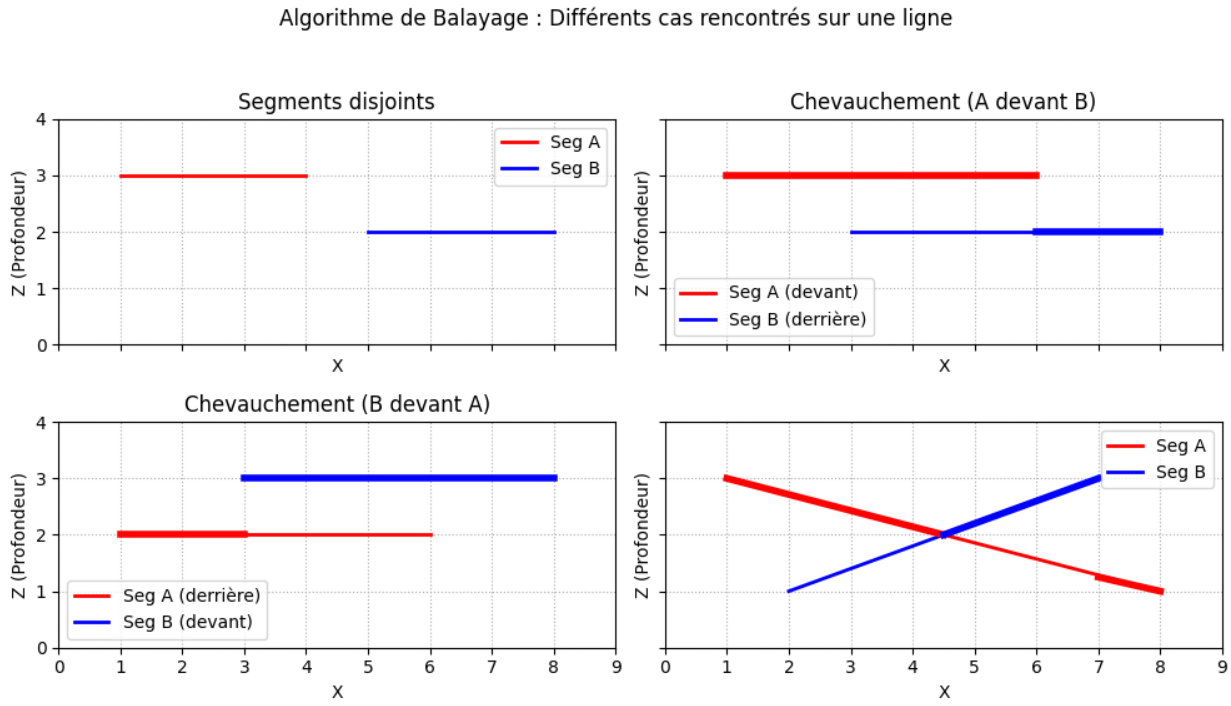

Figure 11: Différents cas de segments rencontrés sur une ligne de balayage et détermination de la visibilité basée sur la profondeur Z.

Des algorithmes spécifiques comme celui de Watkins gèrent ces cas en maintenant une liste active d'arêtes et en calculant les intersections et la visibilité le long de chaque ligne. Une variante utilise un tampon de profondeur (Z-buffer) unidimensionnel pour chaque ligne de balayage.

4 Tampon de Profondeur (Z-buffer)

L'algorithme du Z-buffer, développé par Edwin Catmull en 1974, est l'un des algorithmes d'élimination des parties cachées les plus simples et les plus utilisés, notamment car il est souvent implémenté directement dans le matériel graphique.

4.1 Concept

L'idée principale est de maintenir une mémoire tampon bidimensionnelle, appelée tampon de profondeur ou Z-buffer, de même taille que l'image à générer. Chaque élément de ce tampon stocke la profondeur (coordonnée Z) de l'objet le plus proche trouvé jusqu'à présent pour le pixel correspondant.

- Le calcul de l'image se fait en traitant les objets (polygones/facettes) séquentiellement, les uns après les autres, sans ordre particulier (pas de tri préalable nécessaire).
- Pour chaque polygone, on détermine les pixels qu'il recouvre sur l'écran.
- Pour chaque pixel (x, y) recouvert par le polygone, on calcule sa profondeur Z.
- On compare cette profondeur Z avec la valeur Z déjà stockée dans le Z-buffer à la position (x, y).
- Si la nouvelle profondeur Z est inférieure (plus proche de l'observateur, selon la convention utilisée) à la valeur stockée, cela signifie que ce polygone est devant ce qui avait été dessiné précédemment à ce pixel. On met alors à jour le Z-buffer avec la nouvelle profondeur Z et on écrit la couleur du polygone dans le pixel (x, y) de l'image (tampon couleur).

- Sinon (si Z est supérieur ou égal), le polygone est derrière ou au même niveau que ce qui est déjà visible, donc on ne modifie ni le Z-buffer ni le tampon couleur pour ce pixel.

Il ne s'agit donc pas d'un tri global des objets, mais d'une série de comparaisons et de mises à jour locales (calcul de maximum ou minimum, selon la convention Z) pour chaque pixel.

4.2 Zones Mémoire

Deux zones de mémoire principales sont utilisées :

- **Tampon de Profondeur (Z-buffer)** : Tableau 2D de la taille de l'écran, stockant la profondeur Z du pixel visible le plus proche. Il est initialisé avec une valeur de profondeur maximale (infinie, ou la valeur la plus éloignée possible) pour chaque pixel.
- **Tampon Couleur (Framebuffer)** : Tableau 2D de la taille de l'écran, stockant la couleur finale de chaque pixel. Il est initialisé avec la couleur de fond souhaitée.

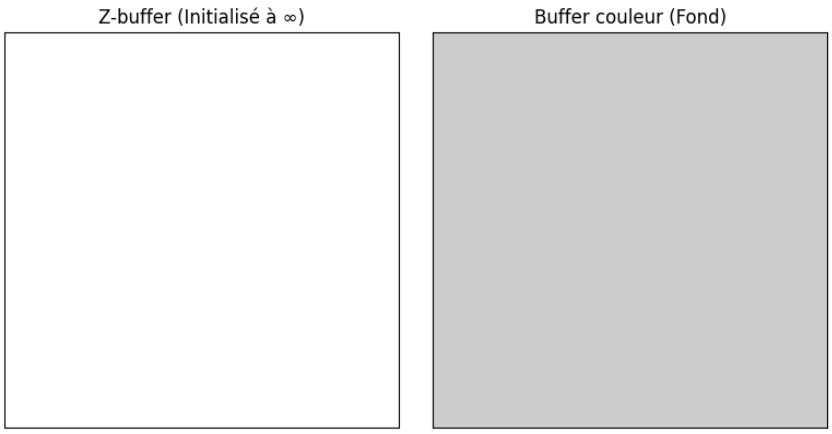

Figure 12: Initialisation des tampons pour l'algorithme Z-buffer.

4.3 Algorithme

Le pseudo-code de l'algorithme Z-buffer est le suivant :

```
// Initialisation
for all pixels (i, j) {
    Depth[i, j] = MAX_DEPTH  // Ou +infini
    Image[i, j] = BACKGROUND_COLOUR
}
// Traitement des polygones
for all polygons P {
    for all pixels (i, j) covered by polygon P {
        Calculate Z_pixel for pixel (i, j) on polygon P // (via interpolation)
        // Convention: Z plus petit est plus proche
        if (Z_pixel < Depth[i, j]) {
            Calculate C_pixel for pixel (i, j) on polygon P // (couleur, ombrage)
            Image[i, j] = C_pixel
            Depth[i, j] = Z_pixel
        }
    }
}
```

Le calcul de la profondeur 'Z$_{p}ixel$'$pourchaquepixelàl'intérieurdupolygoneprojetésefaitgénéralementparinterpolationbiliné$

4.4 Exemple d'Exécution

Considérons le traitement de deux polygones, P1 (violet, Z=5) puis P2 (orange, Z=7), projetés sur l'écran. Le Z-buffer est initialisé à ∞ et le tampon couleur au gris.

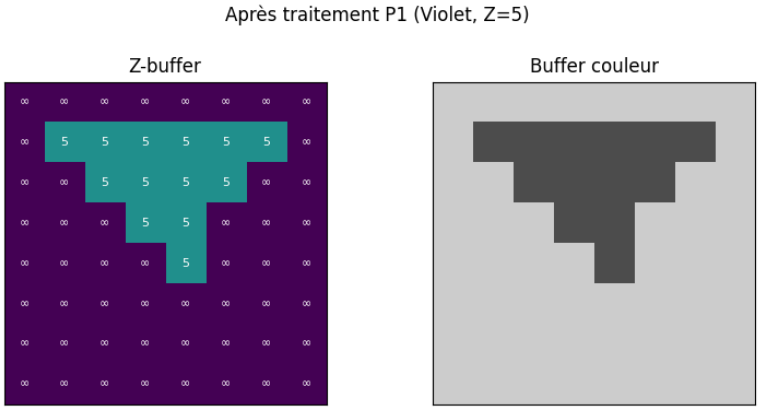

Figure 13: État des tampons après traitement du polygone P1 (Violet, Z=5).

Maintenant, traitons le polygone P2 (orange, Z=7).

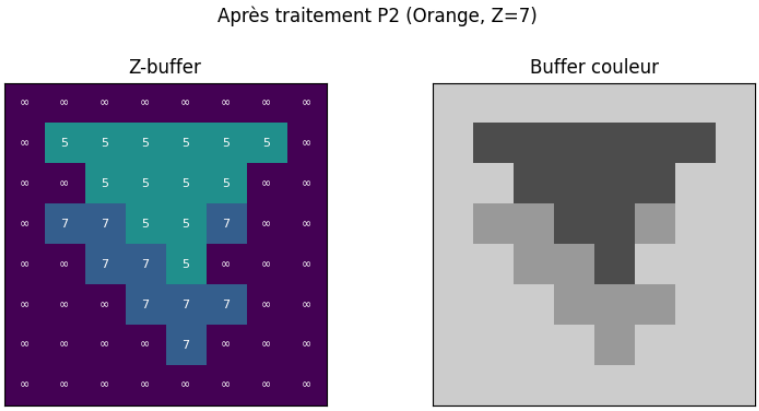

Figure 14: État des tampons après traitement du polygone P2 (Orange, Z=7). Notez que P2 n'écrase P1 que là où P2 est devant (jamais dans cet exemple car Z=7 ¿ Z=5) ou là où P1 n'était pas présent.

Si nous avions traité un troisième polygone P3 (disons, bleu, Z=3) qui recouvre certains pixels de P1 et P2, il aurait écrasé P1 et P2 là où il était présent, car sa profondeur Z=3 est inférieure à 5 et 7.

4.5 Calcul de la Profondeur par Interpolation

Pour déterminer la profondeur Z_p d'un point $P(x_p, y_p)$ à l'intérieur d'un triangle projeté défini par les sommets $P_1(x_1, y_1, z_1)$, $P_2(x_2, y_2, z_2)$, $P_3(x_3, y_3, z_3)$, on utilise généralement l'interpolation bilinéaire. Une méthode courante consiste à interpoler linéairement le long des arêtes, puis à interpoler entre les points interpolés. Par exemple, si P se trouve sur le segment V_sP_3, où V_s est sur le segment P_1P_2, on peut d'abord interpoler Z_s en V_s à partir de Z_1 et Z_2, puis interpoler Z_p en P à partir de Z_s et Z_3. Soit V_s le point d'intersection de la droite horizontale passant par P avec l'arête P_1P_2, et V_t l'intersection avec P_1P_3 (ou

P_2P_3). Les profondeurs Z_s et Z_t peuvent être calculées par interpolation linéaire le long des arêtes :

$$Z_s = Z_1 + (Z_2 - Z_1)\frac{y_s - y_1}{y_2 - y_1} \quad \text{(si } V_s \text{ sur } P_1P_2\text{)}$$
$$Z_t = Z_1 + (Z_3 - Z_1)\frac{y_t - y_1}{y_3 - y_1} \quad \text{(si } V_t \text{ sur } P_1P_3\text{)}$$

(Attention aux cas où les dénominateurs sont nuls - arêtes horizontales). Ensuite, la profondeur Z_p au pixel $P(x_p, y_p)$ est interpolée linéairement entre Z_s et Z_t en fonction de la coordonnée x_p :

$$Z_p = Z_s + (Z_t - Z_s)\frac{x_p - x_s}{x_t - x_s}$$

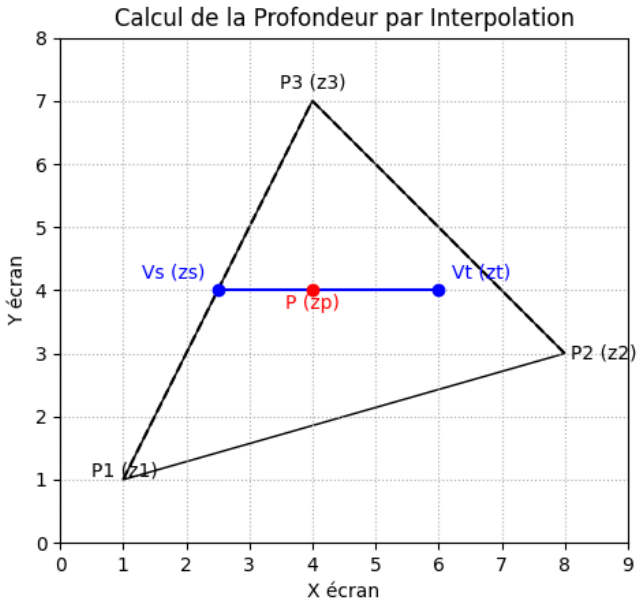

Figure 15: Illustration de l'interpolation bilinéaire pour calculer la profondeur Zp d'un pixel P à l'intérieur d'un triangle projeté.

Note : L'interpolation linéaire de Z dans l'espace écran n'est correcte que pour la projection parallèle. Pour la projection perspective, il faut interpoler 1/Z linéairement, puis inverser le résultat pour obtenir Zp.

4.6 Avantages et Inconvénients

- **Avantages :**
  - Simplicité de mise en œuvre.
  - Pas de tri préalable des polygones nécessaire.
  - Fonctionne pour des scènes complexes avec des polygones entrelacés.
  - Rapidité : Facilement implémentable en matériel, parallélisable au niveau des pixels.
- **Inconvénients :**
  - Traitement de tous les polygones, même ceux qui seront finalement complètement cachés.

- Taille mémoire nécessaire pour le Z-buffer (bien que de moins en moins un problème avec la mémoire graphique actuelle). La précision du Z-buffer (nombre de bits) peut être limitante pour des scènes très profondes.
- Ne traite pas nativement les effets particuliers comme la transparence, les inter-réflexions ou la réfraction (des extensions existent mais complexifient l'algorithme).
- Peut souffrir d'artefacts de "Z-fighting" si la précision du tampon est insuffisante pour distinguer des surfaces très proches.

5 Remplissage de Polygones

Une fois les polygones projetés sur l'écran 2D et leur visibilité déterminée (par ex. via Z-buffer ou Warnock), l'étape suivante est de les "remplir", c'est-à-dire de colorer tous les pixels intérieurs au polygone projeté. Ce processus donne un aspect solide et réaliste aux objets. Le remplissage se fait en utilisant un modèle d'ombrage (shading) pour déterminer la couleur de chaque pixel intérieur, basé sur les propriétés du matériau, les sources de lumière et la géométrie de la surface (par exemple, ombrage plat, Gouraud, Phong, Lambert, etc.). Plusieurs méthodes existent pour déterminer quels pixels sont à l'intérieur d'un polygone projeté et pour les colorer.

5.1 Test d'Appartenance d'un Point à un Polygone

Cette méthode consiste à tester, pour chaque pixel potentiellement couvert par le polygone, s'il se trouve à l'intérieur ou à l'extérieur.

5.1.1 Polygones Convexes : Test des Demi-Plans

Pour un polygone convexe, un point P est à l'intérieur si et seulement s'il se trouve du même côté (par exemple, "à gauche" ou "à l'intérieur") de toutes les droites définies par les arêtes du polygone (orientées de manière cohérente, par ex., dans le sens horaire). On peut déterminer la normale "extérieure" $\vec{N}_{i,i+1}$ pour chaque arête P_iP_{i+1}. Un point P est à l'intérieur si le produit scalaire $\vec{N}_{i,i+1} \cdot \vec{P_iP}$ est négatif ou nul pour toutes les arêtes i. S'il existe une arête pour laquelle ce produit scalaire est positif, le point est à l'extérieur.

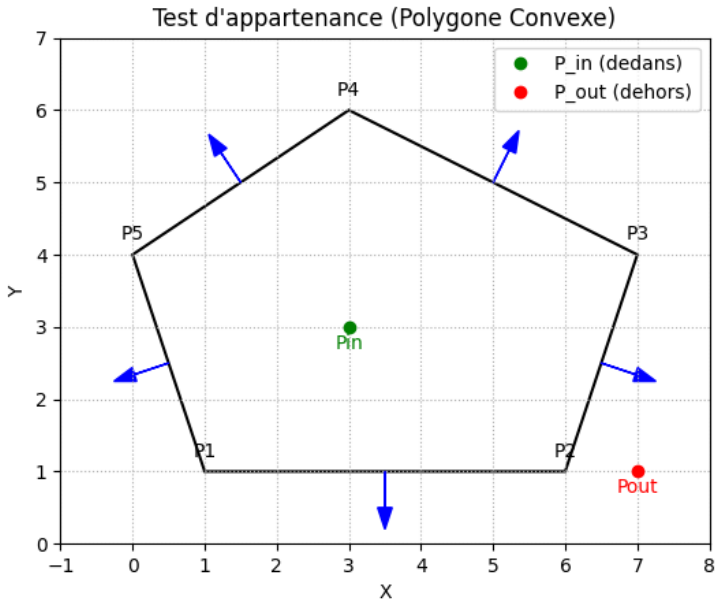

Figure 16: Test d'appartenance pour un polygone convexe. Un point est intérieur s'il est du même côté de toutes les arêtes (par ex., produit scalaire négatif avec les normales extérieures).

5.1.2 Cas Général : Test des Angles

Pour un polygone quelconque (convexe ou concave), on peut calculer la somme des angles orientés formés par les vecteurs reliant le point P à chaque paire de sommets consécutifs (P_i, P_{i+1}).

$$\sum_i \alpha_i = \sum_i \text{angle}(\vec{PP_i}, \vec{PP_{i+1}})$$

Si le point P est à l'intérieur du polygone, la somme des angles est $\pm 2\pi$ (ou 360°). Si le point est à l'extérieur, la somme des angles est 0.

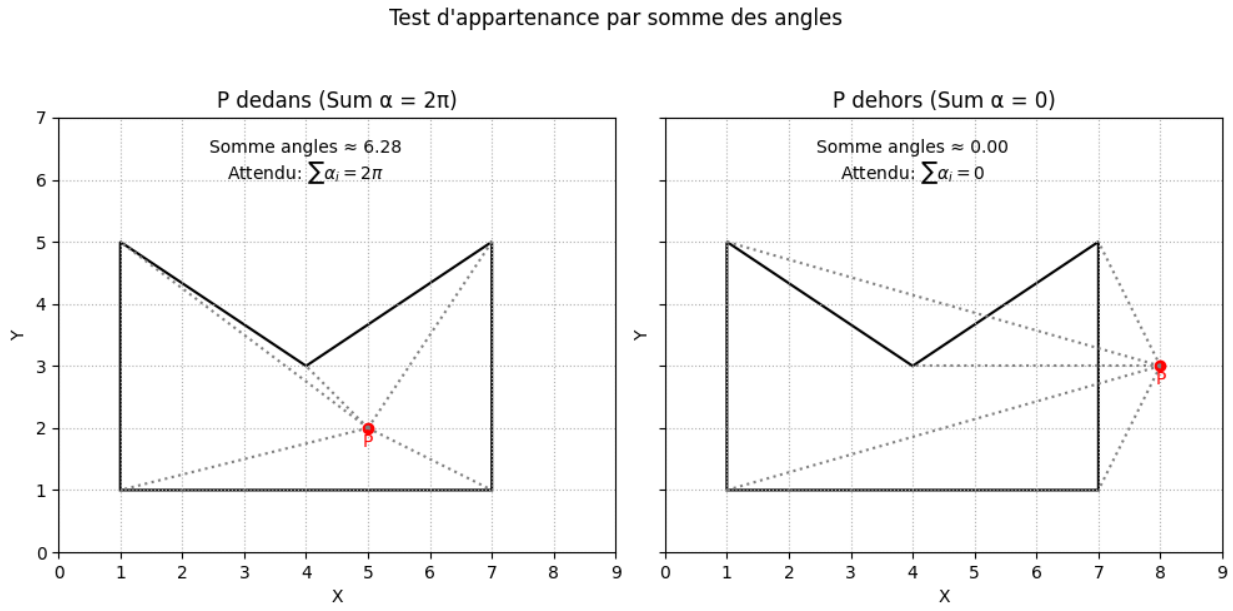

Figure 17: Test d'appartenance par la somme des angles. La somme est 2π si le point est intérieur, 0 s'il est extérieur.

5.1.3 Cas Général : Test du Nombre d'Intersections (Ray Casting)

Une méthode très générale consiste à tracer un rayon (une demi-droite) à partir du point P dans une direction quelconque (par exemple, vers $+X$) et à compter le nombre de fois où ce rayon intersecte les arêtes du polygone.

- Si le nombre d'intersections est **impair**, le point P est à l'intérieur.
- Si le nombre d'intersections est **pair** (y compris zéro), le point P est à l'extérieur.

Il faut traiter attentivement les cas limites :

- Le rayon passe par un sommet : Compter l'intersection uniquement si les deux arêtes adjacentes au sommet sont de part et d'autre du rayon.
- Le rayon chevauche une arête horizontale : Ne pas compter ces intersections ou utiliser une règle cohérente.
- Le point P est sur une arête : Il est considéré comme intérieur ou extérieur selon la convention choisie.

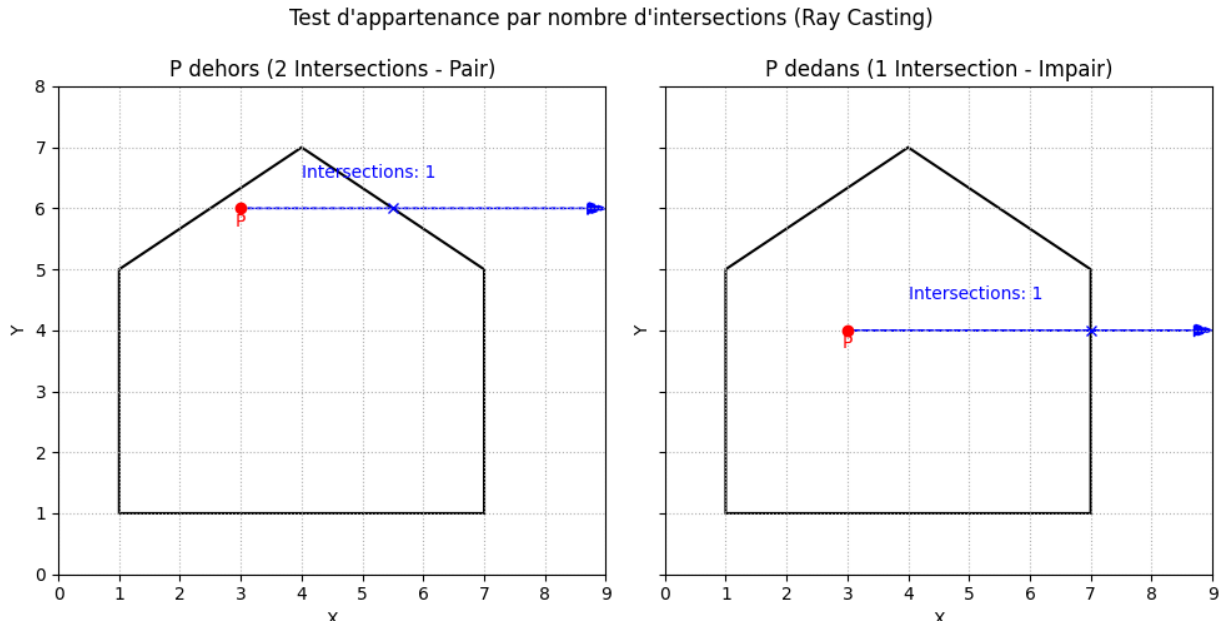

Figure 18: Test d'appartenance par comptage d'intersections. Un nombre impair d'intersections signifie que le point est intérieur, un nombre pair signifie qu'il est extérieur. (Note: les cas limites doivent être gérés avec soin).

5.1.4 Identification d'un Polygone

Pour certains algorithmes, il est utile de savoir si un polygone est convexe ou concave, ou de connaître l'orientation des arêtes. On peut utiliser le produit vectoriel (ou produit mixte en 2D) pour cela. Pour trois sommets consécutifs P_1, P_2, P_3, le signe de $(P_2 - P_1) \times (P_3 - P_2)$ indique le sens de la rotation (gauche ou droite). Si tous les virages sont dans le même sens, le polygone est convexe. Le produit vectoriel entre deux vecteurs $p1 = (p1_x, p1_y)$ et $p2 = (p2_x, p2_y)$ en 2D est $p1_x p2_y - p1_y p2_x$.

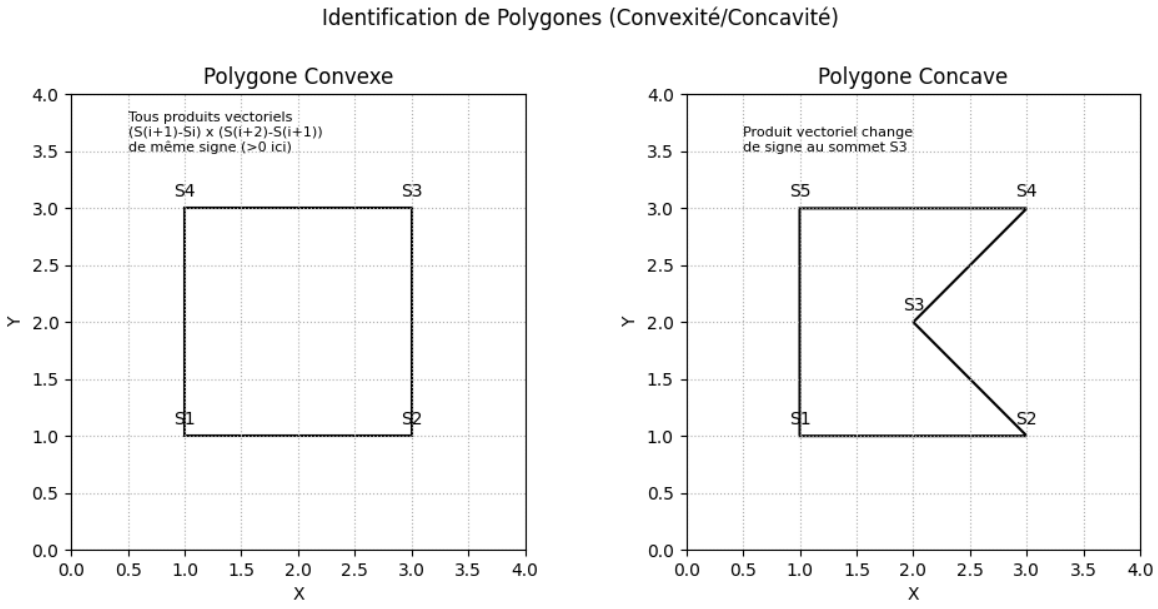

Figure 19: Utilisation du produit vectoriel pour déterminer la convexité (signe constant) ou la concavité (changement de signe) d'un polygone.

5.2 Algorithme de Balayage de Lignes (Scan-Line Filling)

Cette méthode est très efficace pour le remplissage de polygones. Elle combine la détermination des pixels intérieurs et leur coloriage.

- On balaie l'image horizontalement, une ligne de pixels (scan-line) à la fois, typiquement de bas en haut.
- Pour chaque ligne de balayage qui intersecte le polygone :
  1. Trouver toutes les intersections de la ligne de balayage avec les arêtes du polygone.
  2. Trier ces intersections par coordonnée X croissante.
  3. Remplir les pixels entre les paires successives d'intersections (1ère et 2ème, 3ème et 4ème, etc.). C'est la règle de parité : on est à l'intérieur du polygone entre une intersection impaire et une intersection paire.
- Pour optimiser la recherche des intersections, on utilise souvent une structure de données appelée "Table des Arêtes Actives" (Active Edge Table - AET). L'AET contient les arêtes qui sont intersectées par la ligne de balayage courante. Pour chaque arête dans l'AET, on stocke des informations utiles comme Y_{max} (coordonnée Y maximale de l'arête), X_{min} (coordonnée X de l'intersection avec la ligne courante), et dx/dy (l'inverse de la pente, pour mettre à jour X_{min} efficacement lorsque l'on passe à la ligne suivante $y + 1$).
- La liste des intersections pour la ligne courante est obtenue à partir des X_{min} des arêtes dans l'AET. Après avoir rempli les segments de la ligne, on met à jour l'AET : on supprime les arêtes dont Y_{max} est atteint, on ajoute les nouvelles arêtes commençant à $y + 1$, et on met à jour X_{min} pour les arêtes restantes ($X_{min} = X_{min} + dx/dy$).

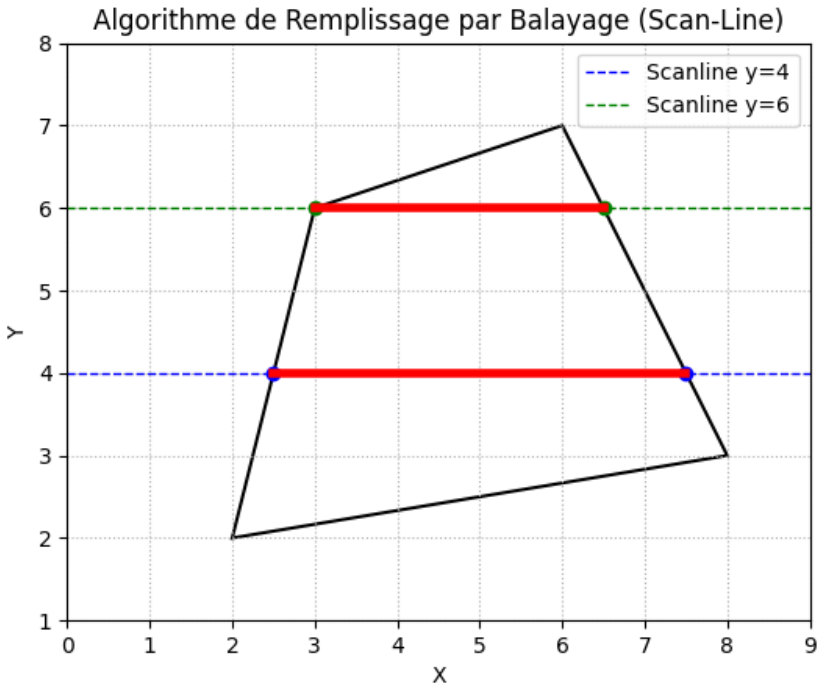

Figure 20: Principe du remplissage par balayage. Pour chaque ligne (ex: y=4, y=6), les intersections avec les arêtes sont trouvées, triées, puis les segments entre paires d'intersections sont remplis.

La gestion des détails, comme les sommets horizontaux, les sommets qui sont des extrema locaux en Y, et la mise à jour de l'AET, est essentielle pour une implémentation correcte.

5.2.1 Gestion des Conflits Frontaliers

Lors de l'utilisation d'algorithmes de traçage de segments (comme Bresenham) pour approximer les arêtes et trouver les intersections X_{min}, des erreurs d'arrondi peuvent survenir. Cela peut conduire à des points d'intersection calculés légèrement à l'extérieur du polygone réel ou à des incohérences entre arêtes partageant un sommet. Pour éviter les chevauchements ou les trous entre polygones adjacents, on peut adopter des conventions strictes : par exemple, toujours prendre les points intérieurs au polygone, ou utiliser les informations de l'équation de la droite (Y_{max}, X_{min}, dx/dy) pour une détermination plus précise des appartenances.

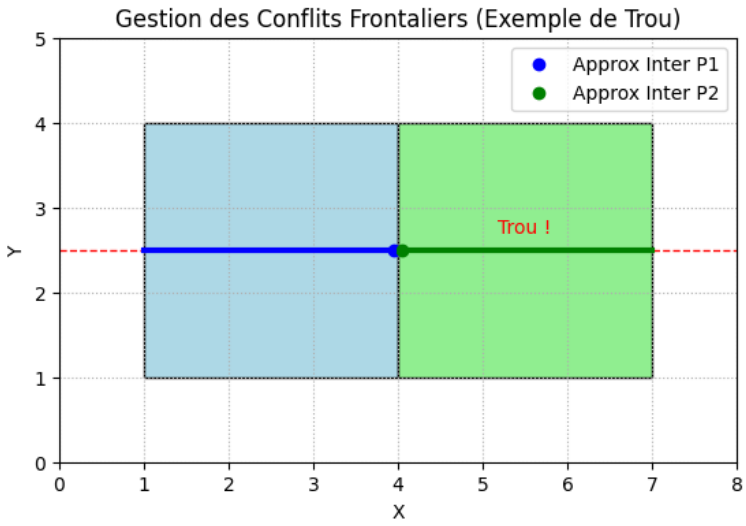

Figure 21: Exemple de conflit frontalier dû à l'approximation. Les intersections calculées pour la même arête partagée peuvent différer légèrement, créant des trous ou des chevauchements si non gérées correctement.

5.3 Remplissage de Régions (Méthode du Germe - Seed Fill)

Cette approche est différente du balayage. Elle fonctionne sur des régions définies par une couleur de frontière explicite dans le tampon d'image (framebuffer).

- On commence avec un pixel "germe" (seed) dont on sait qu'il est à l'intérieur de la région à remplir.
- L'algorithme propage récursivement (ou itérativement avec une pile) la couleur de remplissage aux pixels voisins, tant qu'ils ne sont pas de la couleur de la frontière (ou déjà remplis avec la couleur de remplissage).
- Une version efficace (Scanline Seed Fill) optimise en remplissant des segments horizontaux (spans) de pixels contigus de même couleur initiale, puis en cherchant des pixels germes pour les lignes adjacentes (au-dessus et au-dessous) uniquement au début et à la fin de ces segments.

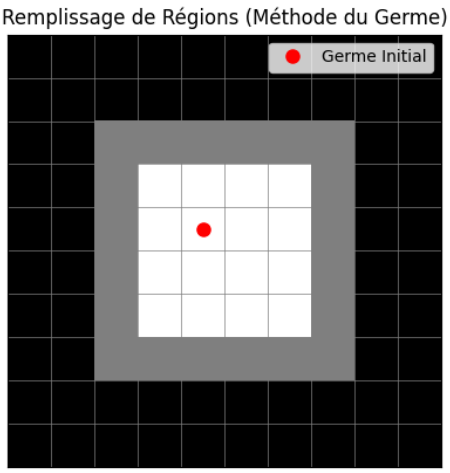

Figure 22: Principe du remplissage par germe. À partir d'un pixel intérieur (germe), la couleur se propage aux voisins jusqu'à rencontrer la frontière.

Cette méthode est utile lorsque les régions sont définies par des frontières de pixels plutôt que par des définitions géométriques de polygones.

6 Coûts Comparés des Algorithmes

Les performances des différents algorithmes d'élimination des parties cachées dépendent fortement de la complexité de la scène (nombre de polygones, profondeur, etc.). Le graphique suivant donne une idée qualitative des coûts relatifs.

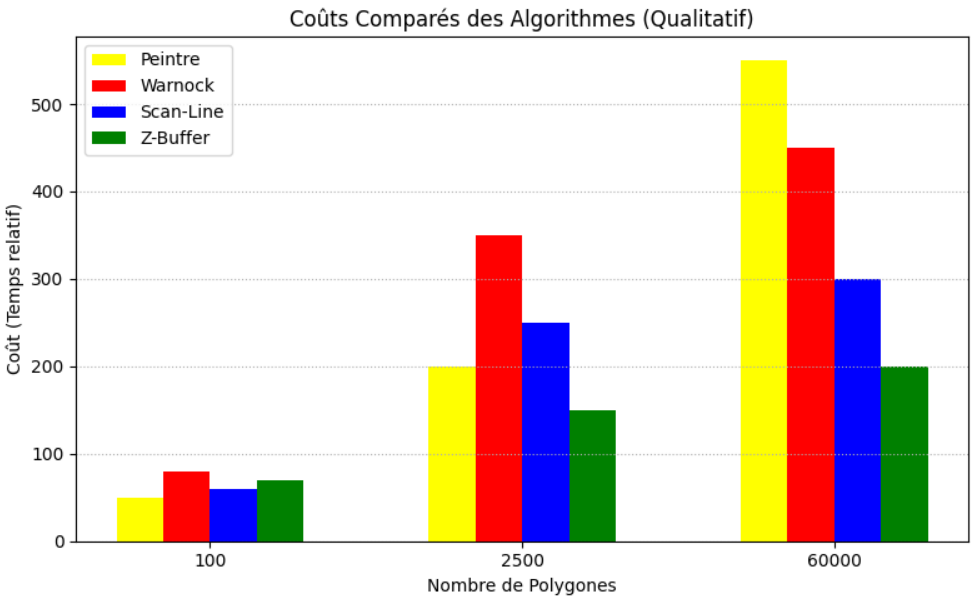

Figure 23: Comparaison qualitative des coûts (temps d'exécution) des algorithmes en fonction du nombre de polygones dans la scène. (Adapté de l'image 95).

On observe que le Z-buffer a tendance à avoir un coût qui augmente moins rapidement avec le nombre de polygones que d'autres méthodes comme l'algorithme du peintre ou Warnock, surtout pour des scènes complexes. Scan-Line se situe souvent entre les deux. Cependant, ces coûts dépendent aussi fortement de l'implémentation (matérielle vs logicielle) et des caractéristiques spécifiques de la scène.

7 Choix de l'Algorithme

Le choix de l'algorithme idéal pour l'élimination des parties cachées et le remplissage dépend de plusieurs facteurs :

- **Complexité de la scène :** Nombre de polygones, profondeur moyenne, distribution spatiale.
- **Matériel disponible :** Présence d'accélération matérielle (GPU) pour certains algorithmes (notamment Z-buffer).
- **Effets souhaités :** Besoin de gérer la transparence, les réflexions, etc., qui peuvent être plus faciles à intégrer dans certains algorithmes.
- **Pré-traitements :** Certains algorithmes nécessitent des tris préalables (Peintre) ou des structures de données complexes (Warnock, Scan-Line AET).
- **Coût mémoire :** Compromis entre la mémoire nécessaire (ex: Z-buffer) et le temps de calcul.

7.1 Scénarios d'Utilisation

- **Z-Buffer :**
  - **Usage général :** C'est l'algorithme le plus couramment utilisé aujourd'hui, surtout grâce à son implémentation matérielle généralisée dans les GPU.
  - **Avantages :** Simple à implémenter (si logiciellement), pas de pré-traitement complexe, performance relativement indépendante de l'ordre des polygones, fourni "gratuitement" par la librairie graphique / le matériel.
  - **Inconvénients :** Peut nécessiter beaucoup de mémoire (moins un problème aujourd'hui), moins élégant pour gérer certains effets (transparence), peut avoir des problèmes de précision (Z-fighting).
- **Scan-Line :**
  - **Usage spécifique ("pour les hackers") :** Peut être très efficace si implémenté soigneusement en logiciel, surtout lorsque l'accès direct au matériel graphique n'est pas possible ou souhaité.
  - **Avantages :** Faible coût mémoire (ne stocke que les informations de la ligne courante), cohérence spatiale exploitée (les informations d'une ligne aident pour la suivante), peut être très efficace car lié directement à la fonction d'affichage ligne par ligne.
  - **Inconvénients :** Plus complexe à implémenter correctement (gestion de l'AET, cas particuliers), moins facilement parallélisable que le Z-buffer au niveau pixel.
- **Warnock :**
  - **Usage historique/pédagogique :** Intéressant pour son approche "diviser pour régner".
  - **Avantages :** Peut être rapide pour des scènes simples ou avec de grandes zones uniformes.
  - **Inconvénients :** La subdivision peut devenir coûteuse pour des scènes complexes, la gestion des polygones aux frontières des quadrants ajoute de la complexité. Moins utilisé dans les systèmes modernes comparé au Z-buffer.

En pratique, l'algorithme Z-buffer domine largement dans le rendu en temps réel grâce à son efficacité et son intégration matérielle. Les algorithmes de type Scan-Line peuvent encore trouver leur place dans des contextes spécifiques ou pour des rendus logiciels optimisés.