1 Rastérisation

La rastérisation, aussi appelée pixélisation, est un processus fondamental en infographie qui consiste à convertir des descriptions géométriques de primitives (comme les points, les lignes et les triangles) en une représentation basée sur des pixels sur un écran ou dans une image bitmap. C'est une étape clé du pipeline graphique qui permet d'afficher des scènes 3D sur des écrans 2D.

1.1 Pipeline Graphique

Le pipeline graphique est une séquence d'étapes qui transforment une description de scène 3D en une image 2D affichable. La rastérisation est l'une des dernières étapes majeures de ce pipeline.

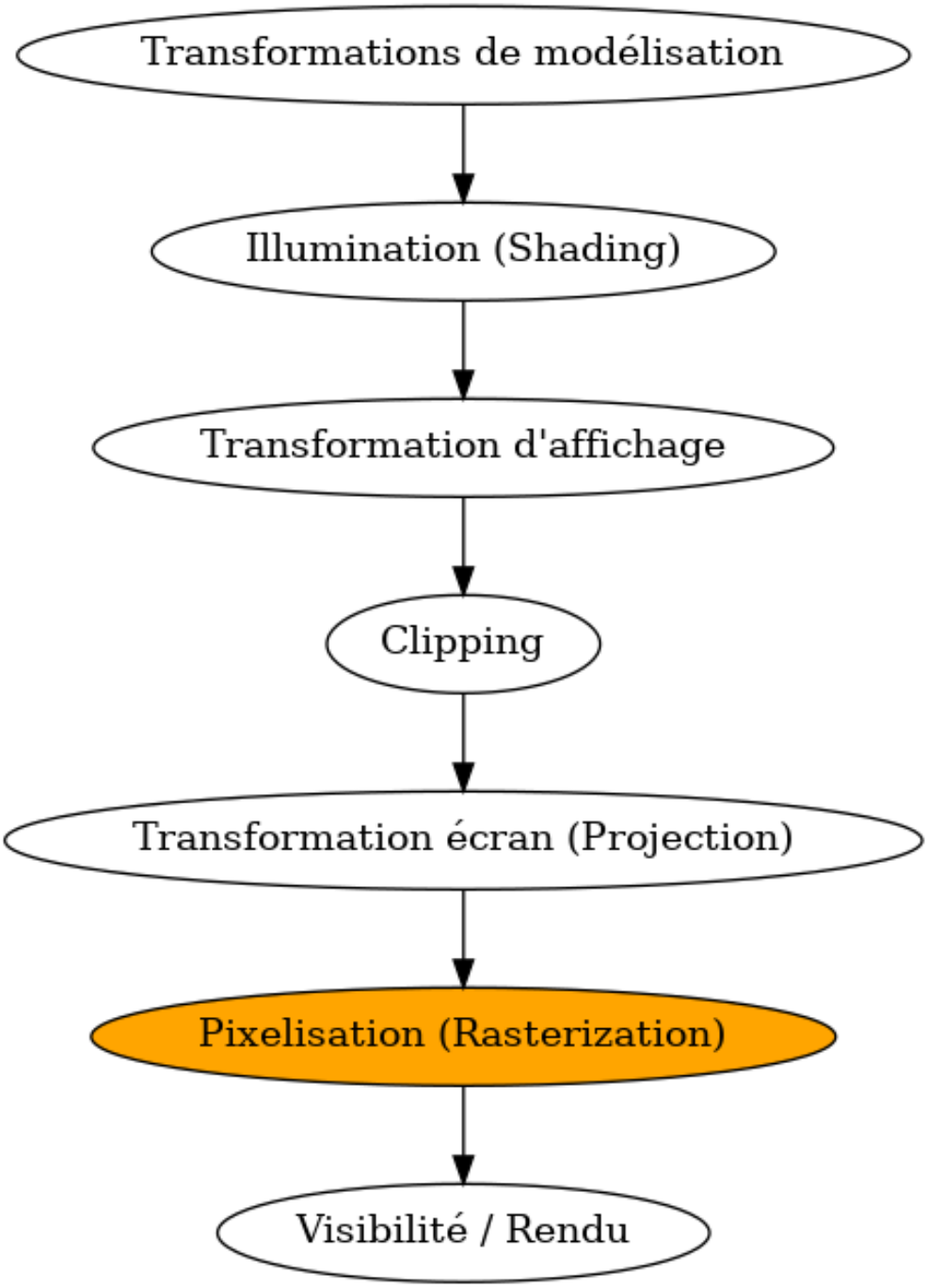

Figure 1: Étapes simplifiées du pipeline graphique mettant en évidence la Rastérisation.

La rastérisation implique deux actions principales :

- **Découpe des primitives 2D en pixels :** Déterminer quels pixels de la grille de l'écran sont couverts par chaque primitive géométrique (après projection).
- **Interpolation des valeurs connues aux sommets :** Calculer les attributs (comme la couleur, la profondeur, les coordonnées de texture) pour chaque pixel couvert (fragment) en interpolant les valeurs définies aux sommets de la primitive.

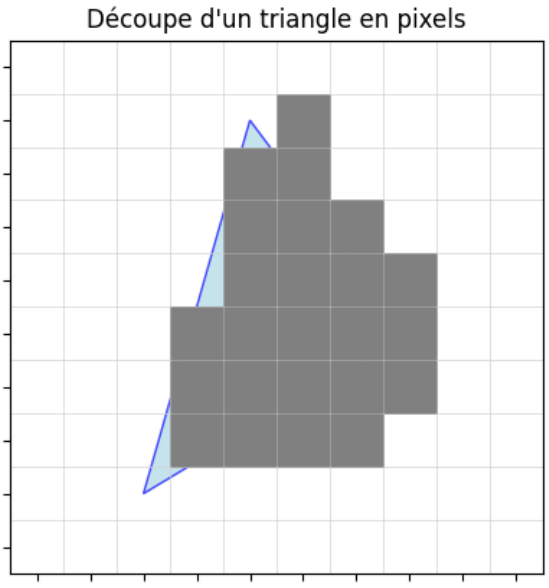

Figure 2: Illustration de la découpe d'une primitive (triangle) en pixels sur une grille.

1.2 Pipeline de Rastérisation

Le pipeline de rastérisation moderne est conçu pour la génération d'images en temps réel.

- **Input:** Primitives 3D, essentiellement des triangles, potentiellement avec des attributs supplémentaires (couleur, normales, coordonnées de texture). Par exemple, un objet peut être défini par une liste de sommets (LS) et une liste de faces (F), chaque face référençant des sommets:
  - LS = {P0, P1, P2, P3}
  - F1 = (LS[0], LS[1], LS[2])
  - F2 = (LS[0], LS[2], LS[3])
  - ...
  - Obj = {F1, F2, F3, F4}
- **Output:** Une image bitmap (un tableau 2D de pixels), potentiellement avec des informations supplémentaires par pixel (profondeur pour le test de visibilité, alpha pour la transparence).
- **Objectif:** Comprendre les étapes intermédiaires qui mènent de la géométrie aux pixels.
- **Exécution:** En pratique, ces étapes sont massivement parallélisées et exécutées par le GPU (Graphics Processing Unit).

1.3 Pourquoi les triangles?

Le pipeline de rastérisation est fortement optimisé pour le traitement des triangles. Toutes les primitives géométriques, y compris les points et les lignes, sont généralement converties en triangles avant ou pendant la rastérisation.

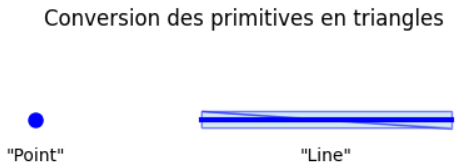

Figure 3: Conversion conceptuelle de points et lignes en triangles pour la rastérisation.

Pourquoi cette focalisation sur les triangles?

- **Approximation universelle:** N'importe quelle forme 3D complexe peut être approximée par un maillage de triangles.
- **Planaire:** Un triangle est toujours planaire, ce qui garantit un vecteur normal bien défini (utile pour l'éclairage).
- **Interpolation facile:** Les attributs (couleur, etc.) peuvent être facilement et efficacement interpolés sur la surface du triangle en utilisant les coordonnées barycentriques.

1.4 Pixels sur l'écran

L'étape finale de la rastérisation consiste à déterminer la couleur de chaque pixel sur l'écran.

- Chaque élément d'image (pixel) est affiché comme un petit carré de lumière avec la couleur appropriée (approximativement).
- Pixel signifie "picture element".

1.5 Rastérisation en bref

La question centrale de la rastérisation est: **Quels pixels le triangle (projeté) chevauche-t-il?**

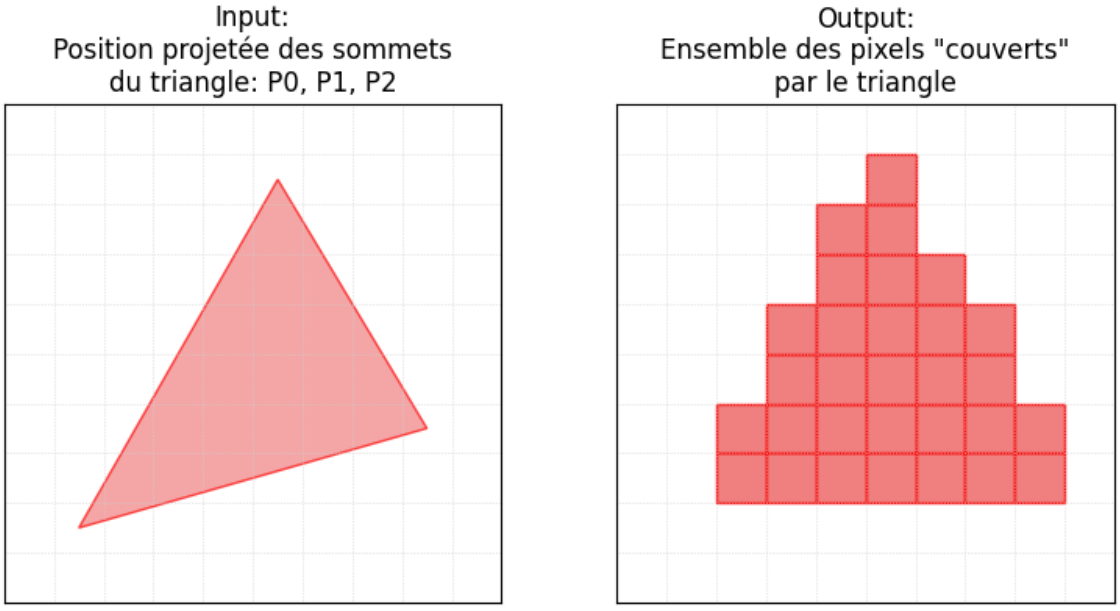

Figure 4: Entrée (sommets du triangle projeté) et sortie (pixels couverts) de la rastérisation.

Une autre question importante gérée pendant ou après la rastérisation est la visibilité : **Quel est le triangle le plus proche de la caméra dans chaque pixel ?** Ceci est généralement résolu à l'aide d'un Z-buffer (tampon de profondeur).

Les principaux aspects abordés dans le contexte de la rastérisation incluent :

- **Traçage :** Algorithmes pour dessiner des lignes et des courbes (segments de droites, cercles).
- **Anticrénelage (Antialiasing) :** Techniques pour réduire les artefacts visuels (jaggies) dus à la discrétisation.
- **Remplissage des primitives projetées :** Algorithmes pour remplir l'intérieur des polygones (triangles).

1.6 Traçage : Segments de Droites

Le traçage de segments de droite est un algorithme de base essentiel pour de nombreux traitements en infographie, tels que le dessin en fil de fer, le remplissage de polygones, et l'élimination des parties cachées. L'objectif est de déterminer quels pixels doivent être allumés pour représenter au mieux un segment de droite continu entre deux points (x_1, y_1) et (x_2, y_2) sur une grille de pixels discrète.

Il y a trois impératifs pour un bon algorithme de traçage de segment discret :

1. Tout point du segment discret est traversé par le segment continu.
2. Le segment discret doit être connexe. Tout point du segment discret touche au moins un autre point, soit par l'un de ses côtés (4-connexité), soit par un de ses sommets (8-connexité). La 4-connexité est souvent préférée pour éviter des lignes trop épaisses en diagonale.
3. On trace le moins de points possibles (idéalement, un seul pixel par colonne ou par ligne de la grille, selon la pente).

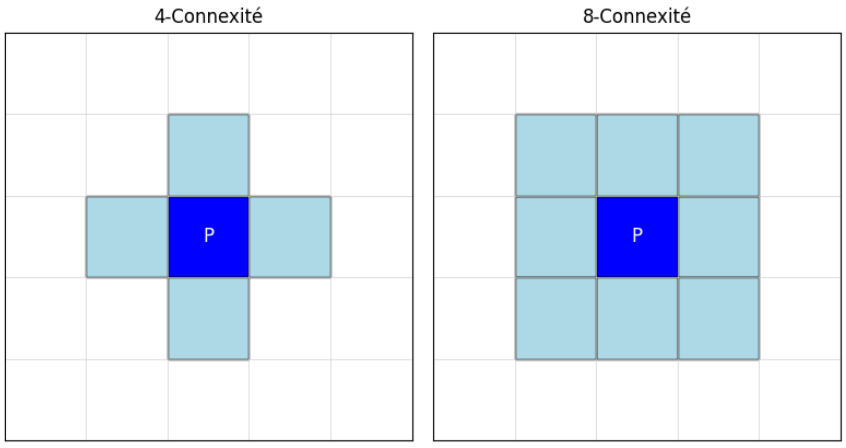

Figure 5: Voisins d'un pixel (P) en 4-connexité (côtés) et 8-connexité (côtés et sommets).

1.6.1 Tracé de segments par l'équation cartésienne

Une approche simple consiste à utiliser l'équation cartésienne de la droite : $y = ax + b$. Soit un segment entre (x_1, y_1) et (x_2, y_2). La pente a et l'ordonnée à l'origine b sont :

$$a = \frac{y_2 - y_1}{x_2 - x_1}$$
$$b = y_1 - ax_1$$

Pour simplifier, supposons $x_2 > x_1$, $y_2 \geq y_1$ et que la pente est inférieure ou égale à 1, c'est-à-dire $(x_2 - x_1) \geq (y_2 - y_1)$. Dans ce cas, pour chaque valeur entière de x entre x_1 et x_2, on calcule la valeur y correspondante et on arrondit à l'entier le plus proche pour déterminer le pixel à allumer.

Incrémentation suivant l'axe x (pente ≤ 1) Si la valeur absolue de la pente $|a| \leq 1$, on incrémente x de x_1 à x_2 et on calcule $y = \text{round}(ax + b)$ pour chaque x.

Listing 1: Algorithme simple basé sur l'équation cartésienne (incrémentation en x)

```
DroiteSimple (int x1, int x2, int y1, int y2)
{
  a = (y2 - y1) / (x2 - x1);
  b = y1 - a * x1;
  x = x1;
  while (x <= x2) // Inclure x2
  {
    AfficherPixel(x, round(a * x + b)); // Arrondi nécessaire
    x = x + 1;
  }
}
```

Figure 6: Exemple de tracé par incrémentation suivant l'axe x (pour une pente ≤ 1).

Incrémentation suivant l'axe y (pente > 1) Si la valeur absolue de la pente $|a| > 1$, on doit incrémenter y de y_1 à y_2 et calculer $x = \text{round}((y - b)/a)$ pour chaque y.

Listing 2: Algorithme simple basé sur l'équation cartésienne (incrémentation en y)

```
DroiteSimpleY (int x1, int x2, int y1, int y2)
{
  a = (y2 - y1) / (x2 - x1);
  b = y1 - a * x1;
  y = y1;
  while (y <= y2) // Inclure y2
  {
    AfficherPixel(round((y - b) / a), y); // Arrondi nécessaire
    y = y + 1;
  }
}
```

Figure 7: Exemple de tracé par incrémentation suivant l'axe y (pour une pente > 1). Les pixels sont ceux indiqués sur la diapositive.

Il faut donc "switcher" entre ces deux versions (incrémentation en x ou en y) en fonction de la pente de la droite. Les autres cas (pentes négatives, $x_1 > x_2$, etc.) peuvent être traités par symétrie.

Caractéristiques

- **Simplicité algorithmique :** L'idée de base est facile à comprendre.
- **Lenteur :** Cette méthode nécessite des calculs en virgule flottante (division pour a, multiplication ax, addition $ax + b$), ainsi qu'une opération d'arrondi ('round' ou 'cast' implicite après ajout de 0.5), ce qui est coûteux en termes de performance par rapport aux opérations sur entiers.

1.6.2 Algorithmes de Bresenham

L'algorithme de Bresenham (développé par Jack Bresenham en 1962) est une méthode beaucoup plus efficace pour tracer des lignes car il utilise uniquement des opérations sur entiers (additions, soustractions, comparaisons et décalages binaires implicites par multiplication par 2).

L'idée clé est de maintenir une variable d'erreur e qui représente la distance (ou une valeur proportionnelle à la distance) entre la position y du pixel courant et la position y exacte sur la ligne idéale, pour le x courant. À chaque étape (incrémentation de x), on met à jour e. Si e dépasse un certain seuil (0.5 pour l'erreur normalisée, ou dx pour l'erreur entière), cela signifie que la ligne est passée plus près du pixel supérieur $(x_k + 1, y_k + 1)$ que du pixel est $(x_k + 1, y_k)$. Dans ce cas, on incrémente y et on ajuste e en conséquence.

Considérons le cas simple $0 \leq a \leq 1$. À l'étape k, on a tracé le pixel (x_k, y_k). Pour $x_{k+1} = x_k + 1$, on doit choisir entre le pixel Est $E = (x_k + 1, y_k)$ et le pixel Nord-Est $NE = (x_k + 1, y_k + 1)$. On choisit le pixel le plus proche de la ligne idéale.

Soit y la coordonnée y exacte sur la ligne pour x_{k+1}. La décision est basée sur la distance verticale entre y et le point milieu $M = (x_k + 1, y_k + 0.5)$.

- Si $y < y_k + 0.5$, on choisit E.
- Si $y \geq y_k + 0.5$, on choisit NE.

Cette condition est équivalente à tester le signe d'un paramètre de décision $d = f(x_k + 1, y_k + 0.5)$, où $f(x, y) = (y_2 - y_1)x - (x_2 - x_1)y + c$ est dérivé de l'équation implicite de la droite. L'astuce de Bresenham est de calculer ce paramètre de décision de manière incrémentale en utilisant uniquement des entiers.

Algorithme de base (avec flottants pour l'erreur) Une première version conceptuelle utilise une erreur e initialisée à 0. À chaque pas en x, on ajoute la pente $a = dy/dx$ à e. Si e dépasse 0.5, on incrémente y et on soustrait 1 à e (pour ramener l'erreur par rapport au nouveau y).

Listing 3: Algorithme de Bresenham (conceptuel, avec erreur flottante)

```
DroiteBresenhamFloat (int x1, int x2, int y1, int y2)
{
  dx = x2 - x1;
  dy = y2 - y1;
  a = dy / dx; // Pente (flottant)
  x = x1;
```

```
  y = y1;
  e = 0.0;      // Erreur (flottant)
  while (x <= x2)
  {
    AfficherPixel (x, y);
    e = e + a;
    x = x + 1;
    if (e > 0.5)
    {
      y = y + 1;
      e = e - 1.0;
    }
  }
}
```

Optimisation 1 : Éliminer la division initiale On peut éviter la division dy/dx en multipliant toute l'équation de l'erreur par dx. L'erreur e devient $e' = e \times dx$. L'incrément devient $a \times dx = dy$. Le seuil 0.5 devient $0.5 \times dx$. L'ajustement devient $1.0 \times dx = dx$.

Listing 4: Bresenham sans division initiale (seuil flottant)

```
DroiteBresenhamNoDiv (int x1, int x2, int y1, int y2)
{
  dx = x2 - x1;
  dy = y2 - y1;
  x = x1;
  y = y1;
  e = 0; // Erreur * dx (entier)
  while (x <= x2)
  {
    AfficherPixel (x, y);
    e = e + dy;      // Incrémenter l'erreur (entier)
    x = x + 1;
    // Comparaison avec seuil flottant dx/2
    if (e * 2 > dx) // Equivalent à e > dx/2 en entiers si dx > 0
    {
      y = y + 1;
      e = e - dx;   // Ajuster l'erreur (entier)
    }
  }
}
```

Optimisation 2 : Éliminer la comparaison flottante (ou la multiplication par 2) On peut éliminer la comparaison $e > dx/2$ (ou $2e > dx$) en initialisant l'erreur différemment. L'algorithme classique de Bresenham utilise un paramètre de décision $p_k = 2dx \times e_k = 2dy \cdot (x_k - x_1) - 2dx \cdot (y_k - y_1)$. La valeur initiale est $p_0 = 2dy - dx$. La mise à jour se fait comme suit :

- Si $p_k < 0$, le prochain pixel est $E = (x_k + 1, y_k)$, et $p_{k+1} = p_k + 2dy$.
- Si $p_k \geq 0$, le prochain pixel est $NE = (x_k + 1, y_k + 1)$, et $p_{k+1} = p_k + 2dy - 2dx$.

Listing 5: Algorithme de Bresenham classique (entiers seulement)

```
DroiteBresenhamInt (int x1, int x2, int y1, int y2)
{
```

```
  dx = x2 - x1;
  dy = y2 - y1;
  x = x1;
  y = y1;
  p = 2 * dy - dx; // Parametre de decision initial
  incE = 2 * dy;       // Increment si E choisi
  incNE = 2 * (dy - dx); // Increment si NE choisi

  AfficherPixel (x, y);
  while (x < x2) // Boucle dx fois
  {
    x = x + 1;
    if (p < 0)
    {
      p = p + incE;
      // y ne change pas
    }
    else
    {
      y = y + 1;
      p = p + incNE;
    }
    AfficherPixel (x, y);
  }
}
```

Cet algorithme final n'utilise que des additions, soustractions et comparaisons sur des entiers, le rendant extrêmement rapide. Des adaptations similaires existent pour les autres pentes et directions.

Rapidité due à :

- Utilisation exclusive d'entiers courts (les valeurs de dx, dy, p restent généralement petites, de l'ordre de la résolution de l'écran).
- Opérations arithmétiques simples sur ces entiers (additions, soustractions, comparaisons).

2 Anticrénelage (Antialiasing)

Les algorithmes de tracé comme celui de Bresenham produisent des lignes qui peuvent apparaître comme des "escaliers" (appelés "jaggies"), en particulier sur les lignes à faible pente. Ce phénomène est appelé **crénelage** (aliasing). Il est dû à la nature discrète de la grille de pixels qui sous-échantillonne le signal continu de la ligne idéale. D'autres artefacts, comme les motifs de Moiré, peuvent aussi apparaître.

L'**anticrénelage** (antialiasing) regroupe les techniques visant à réduire ces artefacts pour améliorer la qualité visuelle des images. L'idée générale est de simuler une couverture partielle des pixels par la primitive, en utilisant des nuances de couleur ou d'intensité.

Image: Comparaison Crénelage vs Anticrénelage

Figure 8: Effet du crénelage (gauche) et résultat après anticrénelage (droite).

Les deux principales approches sont :

- Le sur-échantillonnage de l'image.
- Les algorithmes de tracé de segments corrigés.

2.1 Antialiasing: Sur-échantillonnage de l'Image

Le principe est de calculer l'image à une résolution plus élevée que la résolution finale désirée (par exemple, 3x3 ou 5x5 sous-pixels pour chaque pixel final), puis de combiner les valeurs des sous-pixels pour obtenir la valeur du pixel final. Cette combinaison se fait par **filtrage**.

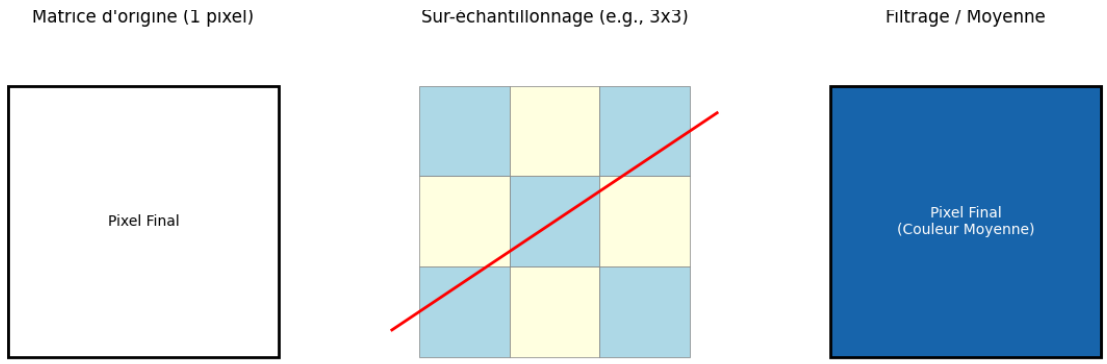

Figure 9: Concept du sur-échantillonnage : calculer sur une grille plus fine, puis filtrer.

Plusieurs paramètres influencent le sur-échantillonnage :

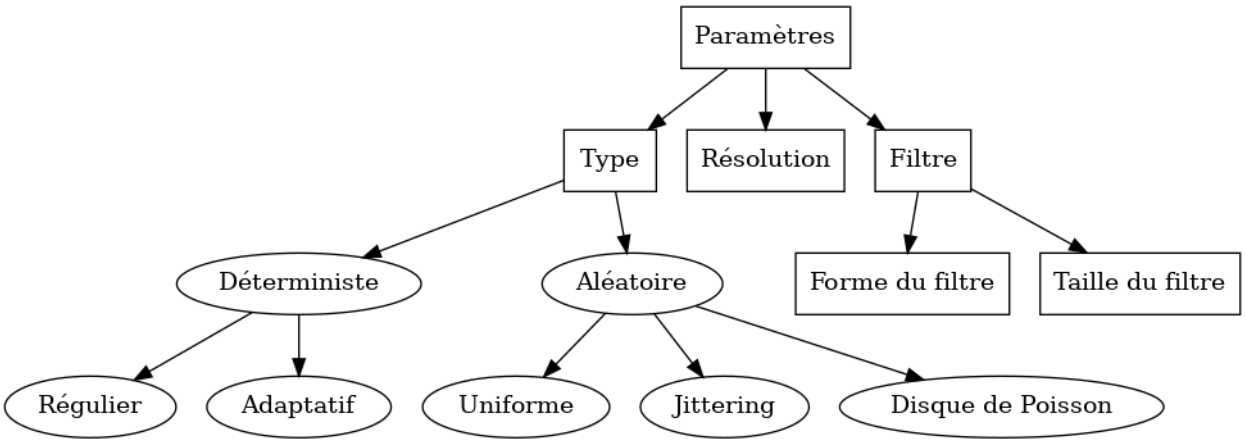

Figure 10: Paramètres clés des techniques d'anticrénelage par sur-échantillonnage.

- **Type d'échantillonnage:**
  - **Déterministe:** Les positions des sous-pixels sont fixes.
    - *Régulier:* Grille régulière de sous-pixels (e.g., 2x2, 4x4). Simple mais peut réintroduire des motifs réguliers. Augmente le temps de calcul par n^2 (si $n \times n$ sous-pixels).
    - *Adaptatif:* Raffinement progressif. On échantillonne plus finement seulement dans les zones où c'est nécessaire (e.g., près des contours, zones à haute fréquence de texture), basé sur certains critères (différence de couleur/profondeur entre échantillons voisins). Plus efficace que le régulier mais plus complexe.
  - **Aléatoire:** Les positions des sous-pixels sont choisies aléatoirement ou pseudo-aléatoirement à l'intérieur du pixel. Transforme l'aliasing cohérent en bruit, ce qui est souvent moins gênant pour l'œil humain.

    - *Uniforme:* Chaque position est tirée indépendamment et uniformément. Simple, mais peut créer des amas et des vides.
    - *Jittering (Grille perturbée):* On part d'une grille régulière et on perturbe aléatoirement la position de chaque échantillon à l'intérieur de sa cellule. Bon compromis entre régularité et aléatoire.
    - *Disque de Poisson:* Garantit une distance minimale entre les échantillons, imitant la distribution des photorécepteurs dans la rétine. Donne des résultats de haute qualité mais plus coûteux à générer.

- **Résolution:** Le nombre de sous-pixels utilisés (e.g., 4x, 8x, 16x). Plus la résolution est élevée, meilleur est l'anticrénelage, mais plus le coût de calcul est important.

- **Filtre:** La manière dont les valeurs des sous-pixels sont combinées.
  - *Forme du filtre:* Définit comment pondérer les échantillons.
    - Boîte (Box): Moyenne simple (tous les échantillons ont le même poids). Le plus simple, mais peut flouter l'image.
    - Cône/Tente (Tent): Poids décroissant linéairement avec la distance au centre du pixel. Meilleur compromis netteté/flou que la boîte.
    - Gaussienne: Poids décroissant selon une fonction gaussienne. Donne des résultats doux, souvent perçus comme naturels.
  - *Taille du filtre:* La zone sur laquelle le filtre est appliqué (souvent liée à la taille du pixel final).

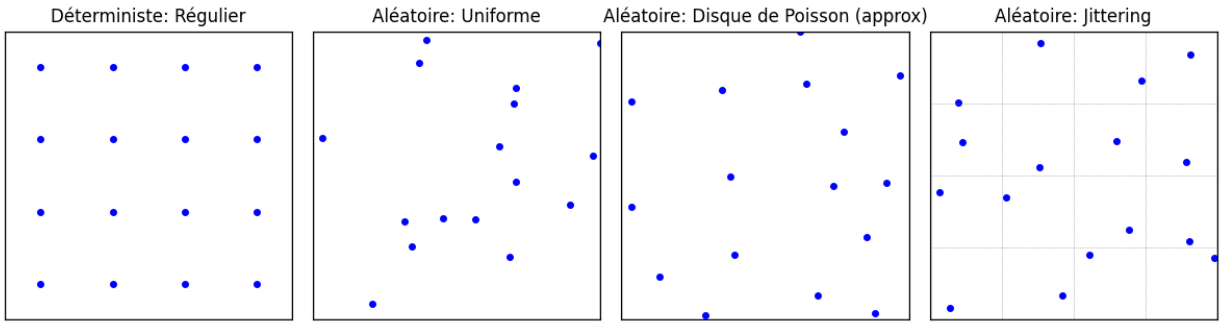

Figure 11: Différents types de motifs d'échantillonnage à l'intérieur d'un pixel.

2.1.1 Forme du Filtre

Le filtre détermine comment les contributions des échantillons sont pondérées pour calculer la couleur finale du pixel.

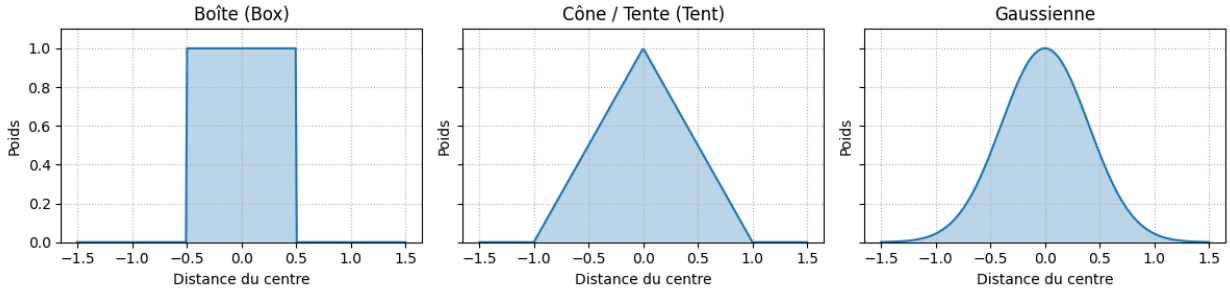

Figure 12: Profils 1D de filtres courants : Boîte, Cône/Tente, Gaussienne.

2.1.2 Taille du Filtre

La taille du filtre (souvent exprimée comme un noyau de convolution) affecte également le résultat. Des filtres plus larges peuvent produire plus de flou mais mieux lisser les hautes fréquences. Les filtres gaussiens sont souvent définis par leur écart-type et peuvent être de différentes tailles (e.g., 3x3, 5x5 pixels).

Exemples de noyaux de filtre Gaussiens (non normalisés):

3x3

1	2	1
2	4	2
1	2	1

5x5

1	2	3	2	1
2	4	6	4	2
3	6	9	6	3
2	4	6	4	2
1	2	3	2	1

2.1.3 Filtrage Aléatoire : Monte-Carlo Pondérée

Pour les échantillons aléatoires, une méthode consiste à utiliser un diagramme de Voronoï basé sur les points d'échantillonnage à l'intérieur du pixel. L'aire de la cellule de Voronoï de chaque échantillon peut être utilisée pour pondérer sa contribution à la couleur finale du pixel.

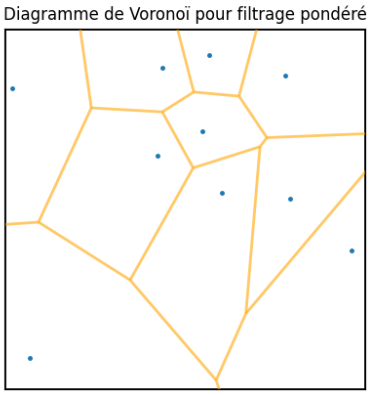

Figure 13: Filtrage Monte-Carlo : pondération par l'aire des cellules de Voronoï.

2.2 Antialiasing: Algorithmes de tracé de segments corrigés

Une alternative (ou un complément) au sur-échantillonnage est de modifier directement les algorithmes de tracé (comme Bresenham) pour qu'ils calculent une intensité ou une couverture pour chaque pixel proche de la ligne, au lieu de simplement l'allumer ou l'éteindre.

Le principe est d'allumer plusieurs pixels à chaque itération, avec des intensités différentes suivant leur proximité par rapport à la primitive (ligne) idéale.

2.2.1 Approche par Aire de Recouvrement

On peut considérer le segment de droite comme ayant une épaisseur non nulle (par exemple, 1 pixel de large). L'intensité d'un pixel est alors proportionnelle à l'aire d'intersection entre ce pixel (carré) et le segment épaissi.

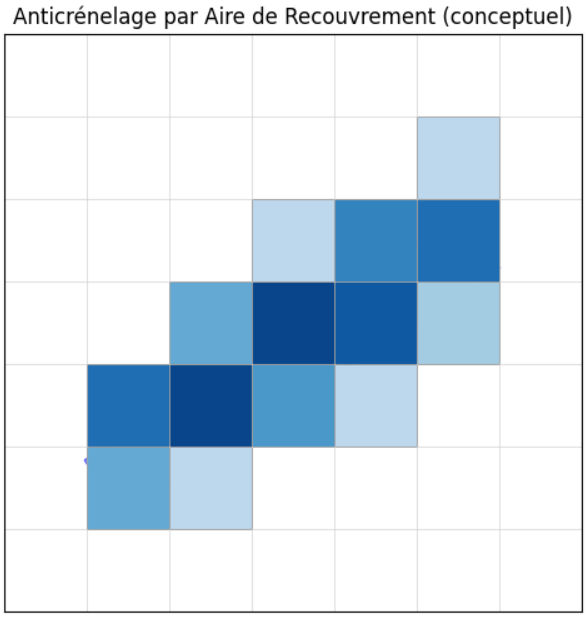

Figure 14: Intensité des pixels proportionnelle à l'aire de recouvrement par une ligne épaisse.

2.2.2 Approche par Distance

Plutôt que de calculer des aires d'intersection (qui peuvent être complexes), on peut calculer l'intensité d'un pixel en fonction de sa distance à la ligne idéale. Plus le pixel est proche, plus son intensité est élevée.

- **Approche non-pondérée (simplifiée) :** L'intensité pourrait être liée à la fraction de la ligne passant dans le pixel.
- **Approche pondérée :** L'intensité est calculée par une fonction (souvent gaussienne) de la distance entre le centre du pixel et la ligne.

2.2.3 Algorithme Gupta-Sproull

Cet algorithme est une modification de Bresenham qui implémente l'anticrénelage basé sur la distance.

- À chaque itération de l'algorithme de Bresenham (qui détermine le pixel principal (x, y)), on active non seulement ce pixel, mais aussi ses voisins (généralement un voisinage 3x3).
- Le niveau de gris (intensité) de chaque pixel activé (le pixel principal et ses voisins) est proportionnel (via une fonction, souvent de type cône ou gaussienne) à sa distance perpendiculaire à la ligne idéale.

L'équation de la droite peut s'écrire $ax + by + c = 0$. La distance D d'un point (x_p, y_p) à cette droite est donnée par :

$$D = \frac{|ax_p + by_p + c|}{\sqrt{a^2 + b^2}}$$

L'intensité I du pixel (x_p, y_p) peut être calculée comme $I = f(D)$, où f est une fonction décroissante (e.g., $f(D) = \max(0, 1 - kD)$ pour un filtre Cône, ou une gaussienne $f(D) = e^{-kD^2}$).

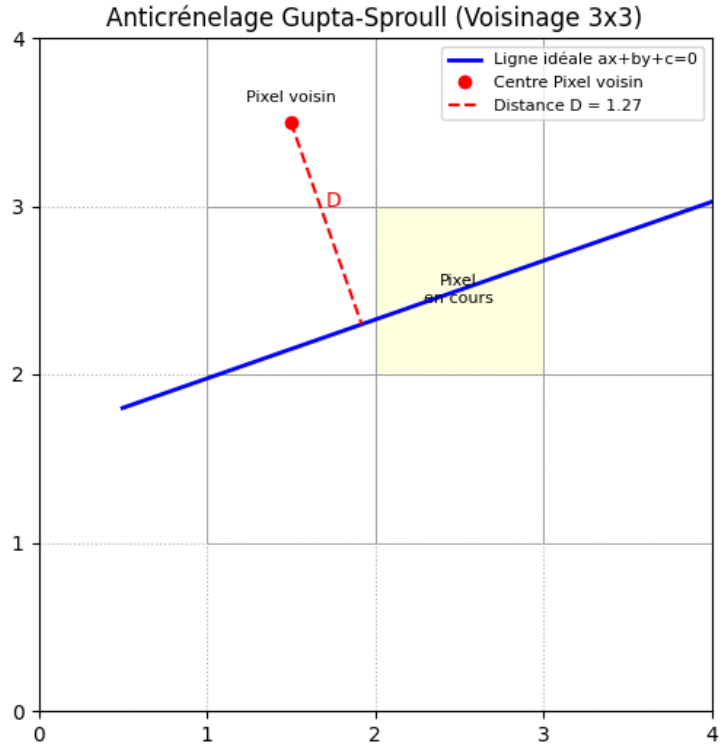

Figure 15: Calcul de l'intensité d'un pixel voisin basée sur sa distance D à la ligne idéale (Algorithme Gupta-Sproull).

Il est facile de modifier l'algorithme de trace de segments (Bresenham) pour calculer et appliquer ces intensités. Classiquement, on applique une fonction de filtrage Gaussienne pour passer de la distance D à l'intensité.

2.3 Antialiasing: Résultats

Les techniques d'anticrénelage améliorent significativement la qualité visuelle des lignes et des contours dans les images rastérisées.

- **Sans antialiasing :** Les lignes présentent des artefacts d'escalier ("jaggies").
- **Avec antialiasing (Sur-échantillonnage / Moyennage) :** Les jaggies sont réduits, les lignes apparaissent plus lisses mais peuvent être légèrement floues, surtout avec un filtre Boîte ou un filtre Gaussien large.
- **Avec antialiasing (Basé sur la distance / Gupta-Sproull) :** Les jaggies sont également réduits, donnant des lignes lisses. La netteté dépend de la fonction de distance utilisée.

Le choix de la méthode dépend du compromis souhaité entre qualité visuelle, performance et complexité de mise en œuvre.

Image: Comparaison des Résultats d'Anticrénelage

Figure 16: Comparaison visuelle : (Gauche) Sans anticrénelage, (Centre) Anticrénelage par sur-échantillonnage, (Droite) Anticrénelage basé sur la distance.

3 Conclusion

La rastérisation est le processus fondamental de conversion de la géométrie vectorielle en une image pixelisée. Les algorithmes de tracé de lignes, comme celui de Bresenham, sont optimisés pour effectuer cette tâche efficacement en utilisant des calculs entiers. Cependant, la nature discrète de la rastérisation introduit des artefacts de crénelage.

L'anticrénelage vise à atténuer ces artefacts. Le sur-échantillonnage calcule une image à haute résolution puis la filtre, offrant diverses options (régulier, adaptatif, aléatoire) et filtres (boîte, cône, gaussien). Les algorithmes de tracé corrigés, comme Gupta-Sproull, modifient directement le tracé pour calculer des intensités de pixels basées sur la proximité de la primitive. Chaque approche présente des avantages et des inconvénients en termes de qualité, de performance et de complexité. Le choix dépend des exigences spécifiques de l'application graphique.