Elimination des surfaces cachées

Michel Buffa, UNSA, UFR Sciences, 1996

1 Introduction

On considère que *les objets 3D sont représentés par des facettes planes triangulaires*. En effet, contrairement aux facettes triangulaires, qui définissent un plan unique à partir de leurs trois sommets, les facettes polygonales non triangulaires ne sont pas forcément planes et peuvent conduire à des problèmes lors du calcul des éclairements ou lors du traitement des collisions et des pénétrations entre objets. Il est par ailleurs plus facile de positionner un point par rapport à une surface plane que par rapport à une surface non plane.

Un autre point en faveur des facettes triangulaires est que les traitements rapides nécessaires en animation interactive s'appuient sur des circuits électroniques spécialement dédiés à leur manipulation (Sillicon Graphics, etc...)

On étudiera dans cette partie du cours 3 méthode de suppression des surfaces cachées : le test de visibilité, le Z-buffer et la liste de priorité.

2 Représentation interne des objets

La représentation interne des objets devra s'appuyer sur trois tableaux. Par exemple (mais rien ne vous oblige à suivre à la lettre ce modèle) :

- `float object[nombre de sommets][3] = {liste des coordonnées x, y, z des sommets}`
- `int facette[nombre de facettes][3] = {liste des indices des sommets appartenant à chacune des facettes}`

  Remarque : les indices sont donnés de façon que la normale à la facette soit dirigée vers l'extérieur de l'objet (voir Figure 1). On utilisera la règle du tire-bouchon : on va du sommet 0 vers le sommet 1 en tournant *dans le sens inverse des aiguilles d'une montre*, le tire-bouchon (la normale) sort de la bouteille (la face).
- `float couleur[nombre de facettes][3] = {liste des couleurs codée en RGB pour chaque facette}`

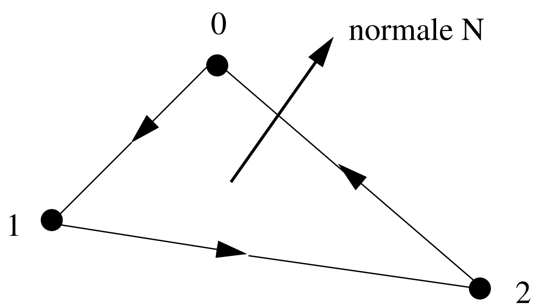


Figure 1 : Définition d'une facette orientée

3 Le test de visibilité

3.1. Description de la méthode

C'est la méthode la plus simple pour résoudre le problème des parties cachées. Elle consiste à vérifier la position de l'observateur par rapport à la normale à la facette.

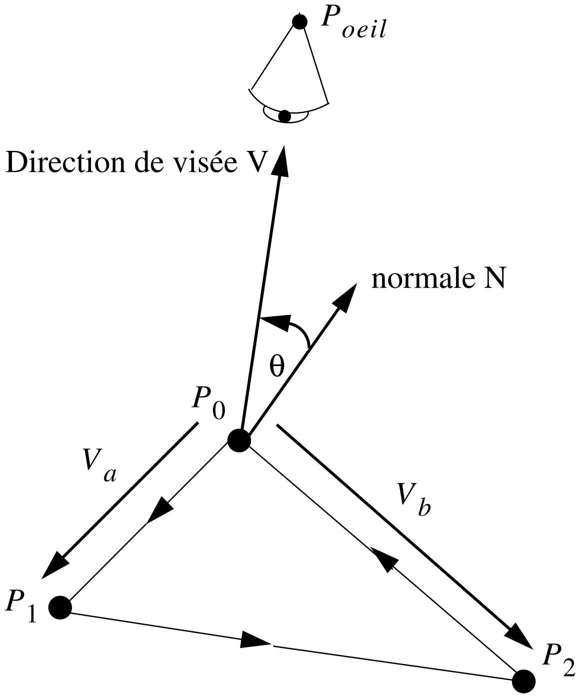


Figure 2 : Calcul de la normale à une facette

Si la normale N est dirigée vers l'observateur, l'angle θ formé entre cette normale et le vecteur V est inférieur à 90 degrés, et la facette est vue. Si la normale N est dirigée à l'opposé, θ est supérieur à 90 degrés, et la facette est non vue.

Calcul de la normale (voir Figure 2) :

- Chaque facette est représentée par trois sommets ordonnés dont on peut déduire les vecteurs $V_a = P_1 - P_0 = (x_a, y_a, z_a)$ et $V_b = P_2 - P_0 = (x_b, y_b, z_b)$. Le produit vectoriel de V_a et V_b permet de calculer les composantes de la normale N, suivant la règle du tire-bouchon :

$$N = V_a \wedge V_b = \begin{bmatrix} z_b \cdot y_a - y_b \cdot z_a \\ x_b \cdot z_a - z_b \cdot x_a \\ y_b \cdot x_a - x_b \cdot y_a \end{bmatrix}. \qquad \text{(EQ 1)}$$

- Connaissant la normale au sommet P_0, on calcule le vecteur $V = P_{oeil} - P_0$ puis on effectue le produit scalaire :

$$V \cdot N = x_v \cdot x_n + y_v \cdot y_n + z_v \cdot z_n \qquad \text{(EQ 2)}$$

Ce produit scalaire peut s'écrire également de la manière suivante :

$$V \cdot N = \|V\| \cdot \|N\| \cdot \cos\theta \qquad \text{(EQ 3)}$$

Par conséquent, lorsque le produit scalaire est positif, la facette est vue puisque θ est inférieur à 90 degrés ($\cos\theta > 0$) et lorsque le produit scalaire est négatif, la facette est non vue puisque θ est supérieur à 90 degrés ($\cos\theta < 0$).

Remarque : ce test est indépendant de la position du sommet P_0 puisqu'on s'intéresse uniquement au signe du cosinus et non pas à la valeur de l'angle θ.

3.2. Inconvénients du test de visibilité

- La méthode élimine les faces cachées d'un objet convexe, mais ne fonctionne pas sur des objets concaves.
- La méthode ne fonctionne pas pour des scènes formées de plusieurs objets même s'ils sont convexes.

4 Le Z-buffer

4.1. Algorithme général

Le Z-buffer constitue une remède aux inconvénients du test de visibilité. Le principe de cette méthode consiste à colorer le pixel avec la couleur de la face située le plus près de l'œil.

Il fonctionne dans le repère écran et travaille dans l'image facette par facette en s'appuyant sur l'algorithme suivant :

```
- Initialiser couleur(X, Y) à la couleur de fond, pour tous
  les pixels de l'écran
- Initialiser Z-buffer(X, Y) à Zmax pour tous les pixels
- Pour chaque facette projetée sur l'écran
  - Pour chaque pixel (X, Y) situé dans la facette
     - Calculer Z en fonction des profondeurs aux sommets de
       la facette
     - Si (Z < Z-buffer(X, Y))
     - Alors
        - Z-buffer(X, Y) = Z
        - couleur(X, Y) = couleur de la facette
     - FinSi
  - FinPour
- FinPour
```

Figure 3 : L'algorithme du Z-buffer

Dans cet algorithme, couleur(X, Y) est la couleur du pixel (X, Y) dans l'image, couleur correspondant à la couleur du point correspondant sur l'objet 3D. Z-buffer(X, Y) représente la valeur de la coordonnée Z dans le repère représenté Figure 4 :

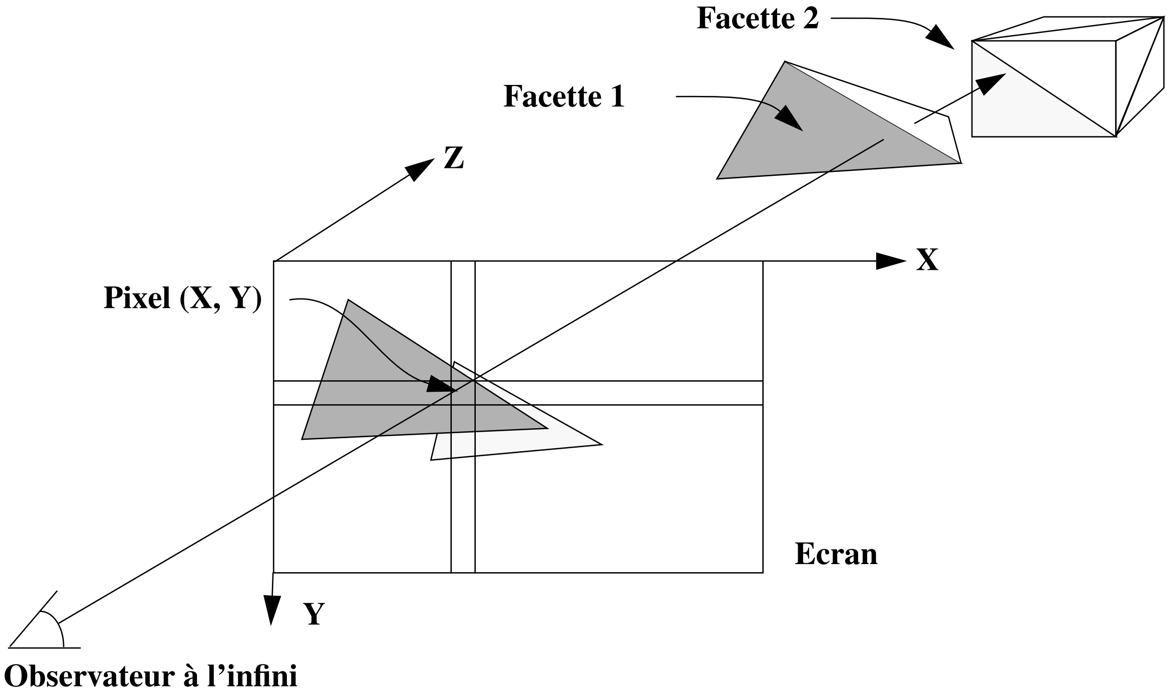


Figure 4 : Principe du Z-buffer

Remarque : la valeur d'initialisation Z_{max} doit représenter une facette située très loin de l'observateur. Avec le repère de la Figure 4 on prendra Z_{max} égal à un nombre positif très grand.

Z-buffer(X, Y) est un tableau d'entiers ou de réels pour stocker la profondeur Z en chaque pixel de l'image. Avec un terminal de 1084 x 1084 pixels de résolution et des réels codés en simple précision sur 32 bits, il nécessite donc une mémoire de 1084 x 1084 x 4 octets = 5 Méga-octets. Pour diminuer la taille mémoire nécessaire, on exprime souvent Z avec des entiers codés sur 16 bits, mais cette solution ne permet pas une bonne précision. Les stations graphiques possèdent fréquemment une zone mémoire de 24 bits dédiée au Z-buffer (Sillicon Graphics par exemple, carte turbo GX pour Suns, etc...)

Malgré sa lourdeur et sa gourmandise en mémoire, le Z-buffer est l'algorithme le plus utilisé car il permet de traiter systématiquement tous les cas qui peuvent se présenter, y compris le cas des facettes interpénétrées présenté sur la Figure 5.

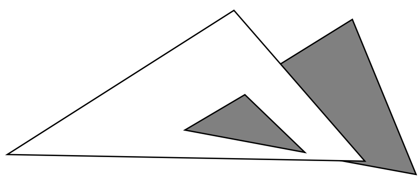


Figure 5 : Interpénétration de facettes

4.2. Le Z-buffer par ligne de balayage

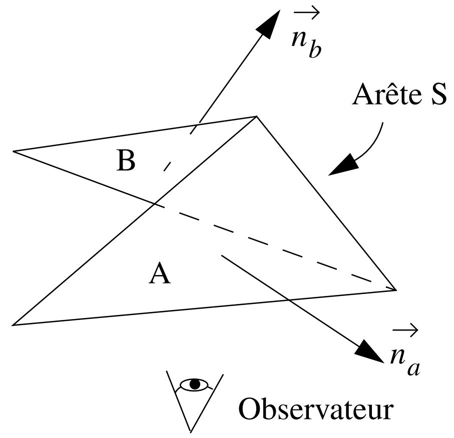
Lorsqu'on ne dispose pas d'une mémoire suffisante pour stocker le Z-buffer en chaque pixel de l'image, on peut utiliser l'algorithme du *Z-buffer par ligne de balayage* qui consiste à traiter l'image ligne par ligne (Figure 6).

```
- Initialiser couleur(X, Y) à la couleur de fond, pour tous
  les pixels de l'écran
- Pour chaque ligne de balayage Y
  - Initialiser Z-buffer(X, Y) à Zmax pour tous les pixels
    de la ligne
  - Pour chaque facette projetée sur l'écran
    - Pour chaque pixel (X, Y) situé dans la facette et sur
      la ligne Y
      - Calculer Z en fonction des profondeurs aux sommets
        de la facette
      - Si (Z < Z-buffer(X, Y))
      - Alors
        - Z-buffer(X, Y) = Z
        - couleur(X, Y) = couleur de la facette
      - FinSi
    - FinPour
  - FinPour
- FinPour
```

Figure 6 : Algorithme du Z-buffer par ligne de balayage

4.3. Problème avec les facettes adjacentes

Problème : le Z-Buffer ne prend pas en compte l'orientation de la normale vis-à-vis de l'observateur et trace systématiquement toutes les facettes proches de celui-ci. Ainsi, lorsque on présente deux facettes A et B qui partagent une arête S, les pixels appartenant à l'arête prendront la couleur de la facette B si celle-ci est traitée avant l'arête A et inversement, ils prendront la couleur de A si celle-ci est traitée avant B (Figure 7).



.

Figure 7 : Problème des faces adjacentes non résolu par le Z-buffer

Une méthode pour éviter ce problème consiste à inclure le test de visibilité dans l'algorithme du Z-buffer. De cette façon, seule les facettes dont la normale est dirigée vers l'observateur seront traitées et les pixels de l'arête S prendront toujours la couleur de la face A.

5 Algorithmes de tri

Ces algorithmes consistent à trier les facettes dans un ordre prédéterminé avant de les tracer sur l'écran. On distingue deux types d'algorithmes : les algorithmes par l*iste de priorité* et les algorithmes de *partitionnement de l'espace*. Les algorithmes par liste de priorité permettent de trier les faces dans l'ordre arrière-avant de manière à tracer d'abord celles qui sont les plus éloignées de l'observateur et terminer par les plus proches. Les algorithmes de partitionnement de l'espace rangent les facettes dans un arbre binaire et utilisent des règles de parcours de l'arbre pour dessiner les faces de la plus éloignée vers la plus proche.

5.1. Algorithmes par liste de priorité

L'algorithme qui range les faces dans l'ordre arrière-avant est appelé *algorithme du peintre*.

Le principal intérêt de cet algorithme est qu'il permet de traiter des objets transparents alors que le Z-buffer ne le permet pas. En effet, lorsque les facettes sont tracées dans l'ordre arrière-avant on peut calculer la couleur du pixel (X,Y) de la face courante F en fonction de la couleur précédemment calculée en (X,Y), correspondant à un point appartenant à la face juste derrière F, et d'un cœfficient de transparence.

5.1.1. Transparence

Si les faces sont triées dans l'ordre arrière-avant, on peut modifier le corps de l'algorithme du Z-buffer :

```
- Si (Z < Z-buffer(X, Y))
- Alors
   - Z-buffer(X, Y) = Z
   - couleur(X, Y) = couleur de la facette
- FinSi
```

par :

```
- Si (Z < Z-buffer(X, Y))
- Alors
   - Z-buffer(X, Y) = Z
   - couleur(X, Y) = t * couleur(X,Y) + (1-t) *
     couleur de la facette
- FinSi
```

Dans le membre de droite de la dernière instruction, t représente le cœfficient de transparence de la face courante ($0 \leq t \leq 1$), couleur(X, Y) représente la couleur actuelle du pixel (X,Y) et “couleur de la facette” représente la couleur de la facette courante. Si $t = 0$ la facette est opaque, si $t = 1$ la facette est complètement transparente.

5.1.2. Principe

Le tri est effectué après la transformation perspective et suppose que les N facettes sont rangées dans un ordre quelconque dans un tableau $T[i], i = 1 \ldots N$. Les algorithmes consistent à comparer les profondeurs des facettes et à éventuellement les inverser dans le tableau T.

Les facettes seront par exemples triées en testant la coordonnée Z maximale d'un de leurs sommets.

Voici deux algorithmes parmi les plus courants. Il en existe d'autres plus performants (quicksort, etc...)

Algorithme 1 :

```
- Pour la facette courante T[j], j = 1...N-1
   - Pour toutes les facettes en cours de test T[i] telles
     que j < i ≤ N-1
      - Si Z de T[j] > Z de T[i]
         - inverser les facettes de T[i] et T[j]
      - FinSi
   - FinPour
- FinPour
```

Figure 8 : Comparaison de la facette courante à toutes celles qui suivent

Algorithme 2 :

```
- TantQue il y a au moins une inversion à faire
   - Pour toutes les facettes T[j] telles que j = 1...N − 1
      - Si Z de T[j] > Z de T[j+1]
         - inverser les facettes de T[j] et T[j+1]
      - FinSi
   - FinPour
- FinTantQue
```

Figure 9 : Tri à bulle (comparaison des facettes 2 à 2)

5.1.3. Inconvénients de ces algorithmes

Dans certains cas, le tri suivant la profondeur maximale d'un des sommets des faces ne suffit plus. Selon le positionnement des facettes on peut rencontrer des ambiguïtés!

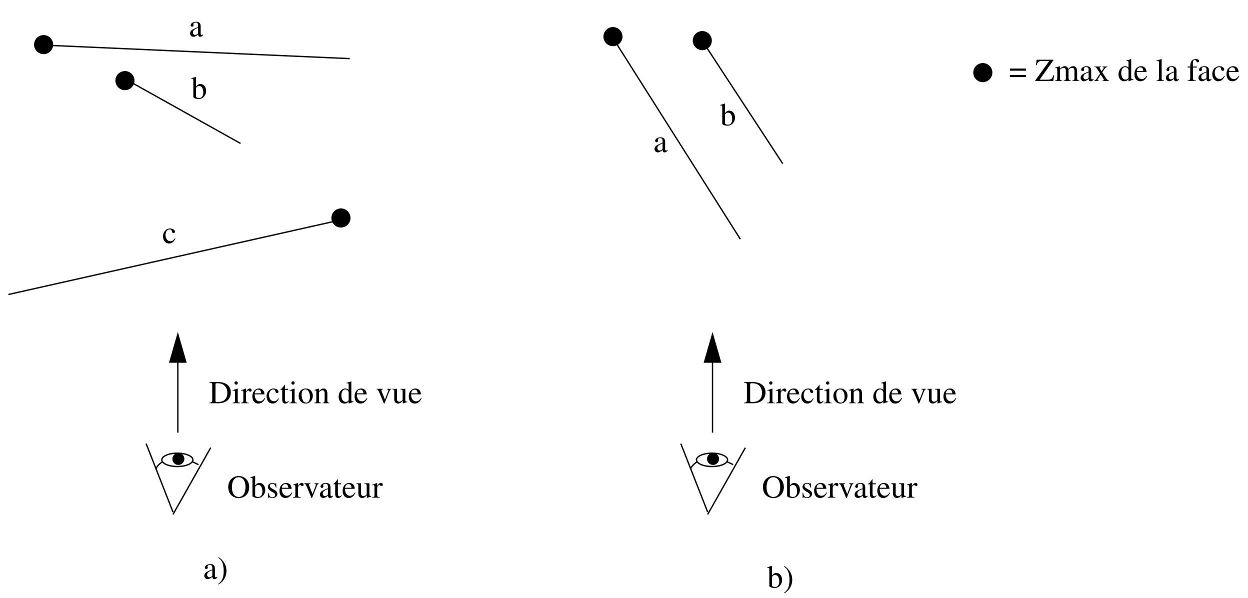


Figure 10 a) fonctionnement correct de la liste de priorité et b) ambiguïté sur la profondeur des facettes

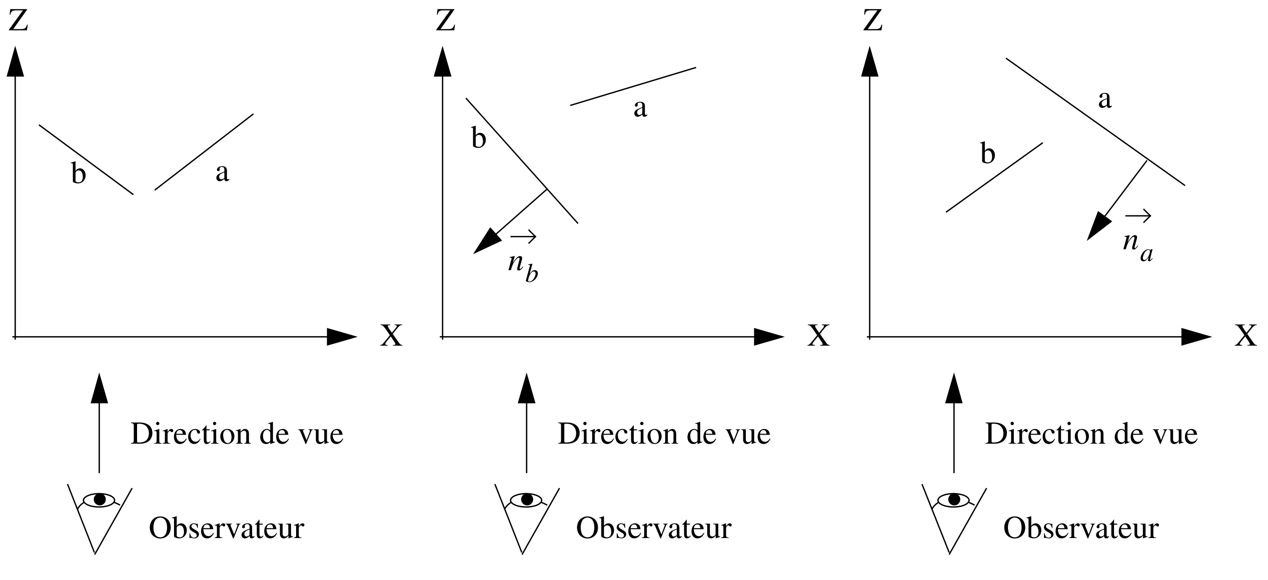
Le tri suivant la profondeur minimale fonctionne correctement avec les faces de la Figure 10a : les facettes sont rangées dans l'ordre arrière-avant sans ambiguïté puisque :

$$Z_{max}(a) > Z_{max}(b) > Z_{max}(c)$$

En revanche il y a ambiguïté lorsque les facettes sont placées comme sur l'exemple de la Figure 10b puisque dans ce cas $Z_{max}(a) > Z_{max}(b)$ alors que la facette a est placée devant la facette b.

Il existe un algorithme pour résoudre ce problème de manière systématique. Soir (a) la facette courante et (b) la facette en cours de test, si la réponse aux questions suivantes est "oui", il ne faut pas inverser les facettes car (a) ne peux pas cacher (b) :

1. Figure 11a : les projections de (a) et de (b) sont-elles disjointes suivant x *et* suivant y ?
2. Figure 11b : (a) est-elle entièrement située du côté du plan de (b) le plus *éloigné* de l'observateur ?
3. Figure 11c : (b) est-elle entièrement située du côté du plan de (a) le plus *proche* de l'observateur ?



a) b) c)

Figure 11 : Cas à envisager pour simplifier les ambiguïtés

En cas de recouvrement cyclique des facettes comme indiqué dans la Figure 12, il est nécessaire de couper des facettes pour éviter des ambiguïtés. On doit par exemple couper la facette (c) suivant le plan de la facette (b) et remplacer dans la liste, la facette (c) par les nouvelles facettes c_1 et $c_2 + c_3$. La coupure nécessite souvent une retriangularisation. Dans l'exemple, on a dû rajouter une arête dans la partie droite de la facette (c), ce qui a crée les nouvelles faces c_2 et c_3.

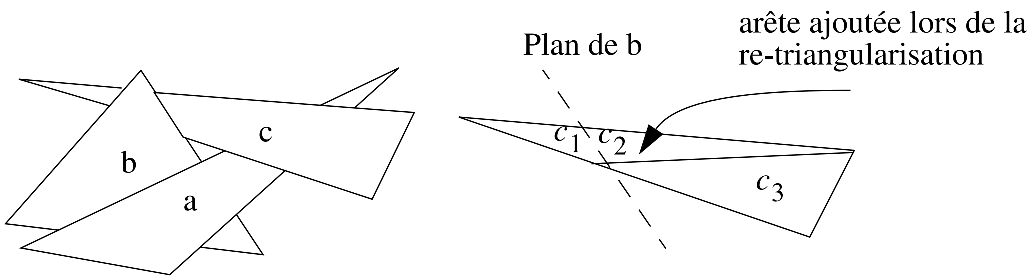


Figure 12 : Traitement du recouvrement cyclique

6 Partitionnement binaire de l'espace

L'algorithme de partitionnement binaire de l'espace développé par Fuchs, Kedem et Naylor est une méthode particulièrement efficace pour précalculer des relations de visibilité d'un ensemble *statique* de polygones observés d'un point de vue arbitraire.

On choisit un polygone initial quelconque qui deviendra la racine de l'arbre que l'on va construire. Ce polygone est utilisé pour partitionner l'espace en deux demi-espaces. L'un de ces demi-espace contient tous les polygones qui se trouvent devant le polygone racine, relativement à la normale orientée de ce dernier. L'autre demi-espace contient tous les polygones qui se trouvent derrière le polygone racine. Tout polygone traversé par le plan du polygone racine est coupé en deux polygones, chacun affecté au demi-espace approprié.

On choisit alors dans chaque demi-espace un polygone arbitraire qui devient un fils du polygone racine. Il est a son tour racine d'un sous-arbre. Il partage l'espace en deux demi-espaces et y range les polygones de son sous-arbre selon qu'ils se trouvent devant ou derrière lui.

Cet algorithme est appliqué récursivement jusqu'à ce que le polygone choisi comme nouveau nœud soit seul dans son demi-espace.

L'arbre binaire ainsi construit ("BSP tree" en anglais pour Binary Space Partitionning Tree) permet, quel que soit le point d'observation, de déterminer rapidement l'ordre dans lequel les polygones doivent être tracés. C'est en positionnant le point d'observation dans la partition de l'espace définie par l'arbre qu'on pourra calculer l'ordre de tracé des polygones: par exemple, si l'observateur se trouve dans le demi-espace avant la racine, il faudra d'abord dessiner les polygones qui se trouvent derrière la racine, puis le polygone racine, puis il faut se positionner par rapport au fils de la racine se trouvant dans le demi-espace avant et réitérer le processus.

7 Comment coder un Z-Buffer efficace

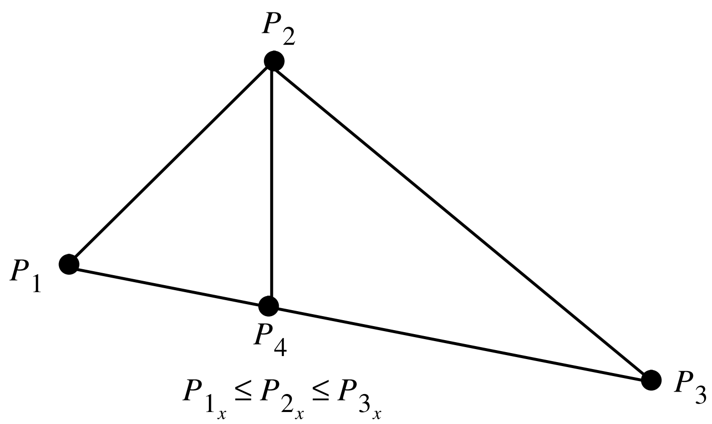
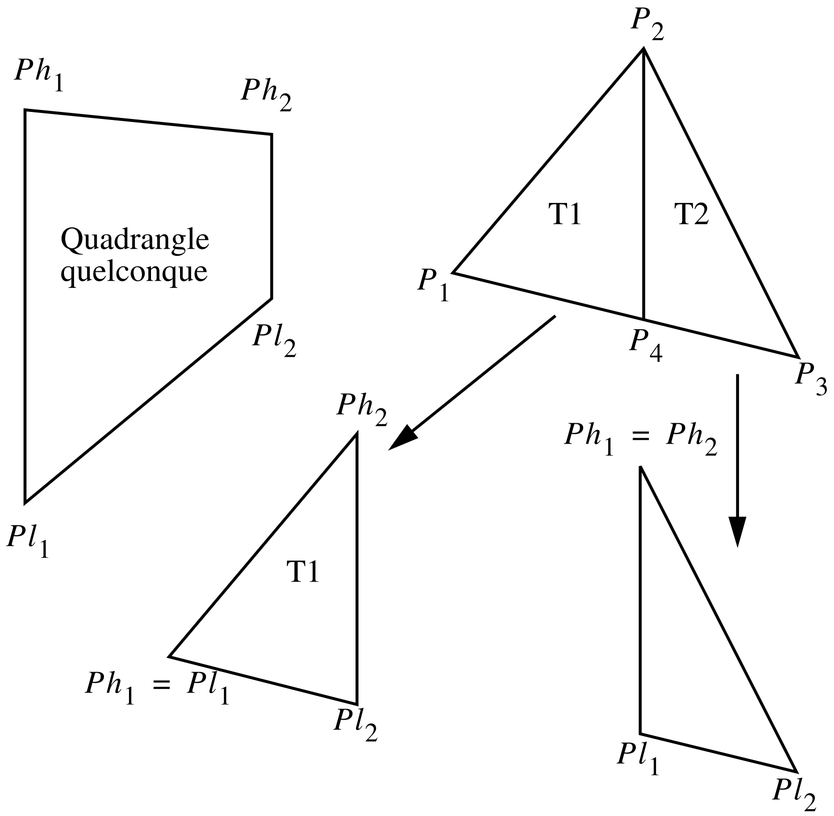


Figure 13 : Découpage du triangle en deux afin de faciliter le balayage vertical



Ph_1
Ph_2
Quadrangle quelconque
Pl_2
Pl_1
P_2
T1
T2
P_1
P_4
P_3
Ph_2
T1
$Ph_1 = Pl_1$
Pl_2
$Ph_1 = Ph_2$
Pl_1
Pl_2

- Classer les sommets selon la coordonnée X $P_{1_x} \le P_{2_x} \le P_{3_x}$
- Calculer $dx = P_{3_x} - P_{1_x}$ le nombre de colonnes entre P_1 et P_3
- **Si** dx différent de 0
- **Alors**
  - Calculer $dbx = P_{2_x} - P_{1_x}$ le nombre de colonnes entre P_2 et P_1
  - Calculer la position écran de P_4 :
    - $q = (P_{3_y} - P_{1_y})/dx$ pente en y entre P_1 et P_3
    - $P_{4_y} = P_{1_y} + dbx \cdot q$
    - $P_{4_x} = P_{2_x}$
  - **Si** ($P_2 = P_4$ **alors** return; (triangle aplati)
  - Calculer P_{4_z} la profondeur de P_4 :
    - $q = (P_{3_z} - P_{1_z})/dx$
    - $P_{4_z} = P_{1_z} + dbx \cdot q$
  - Calculer la couleur de P_4 par interpolation des couleurs de P_1 et P_3 (faire pour el R,le G et le B):
    - $q = (P_{3_{color}} - P_{1_{color}})/dx$
    - $P_{4_{color}} = P_{1_{color}} + dbx \cdot q$
  - **Si** ($P_{4_y} < P_{2_y}$) (P_4 au-dessus de P_2)
  - **Alors**
    - FillQuadrangle(BufferImage, Z-Buffer, P_1, P_1, P_4, P_2)
    - FillQuadrangle(BufferImage, Z-Buffer, P_4, P_2, P_3, P_3)
  - **Sinon**
    - FillQuadrangle(BufferImage, Z-Buffer, P_1, P_1, P_2, P_4)
    - FillQuadrangle(BufferImage, Z-Buffer, P_2, P_4, P_3, P_3)
  - **FinSi**

Figure 14 : Algorithme de découpage de triangle (appelle la fonction de balayage vertical)