Chapitre 3 : Elimination des parties cachées

Modélisation 3D et Synthèse

Fabrice Aubert
fabrice.aubert@lifl.fr

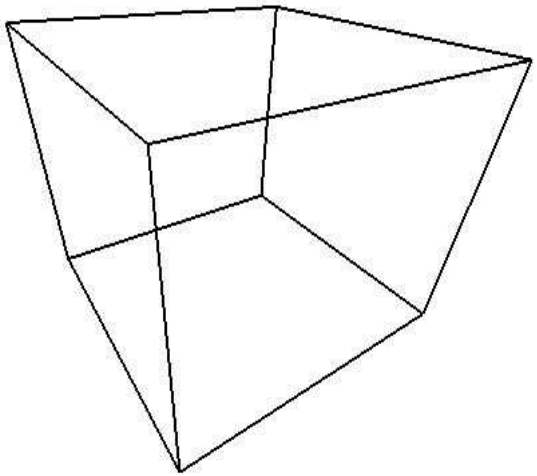
IEEA - Master Info - Parcours IVI

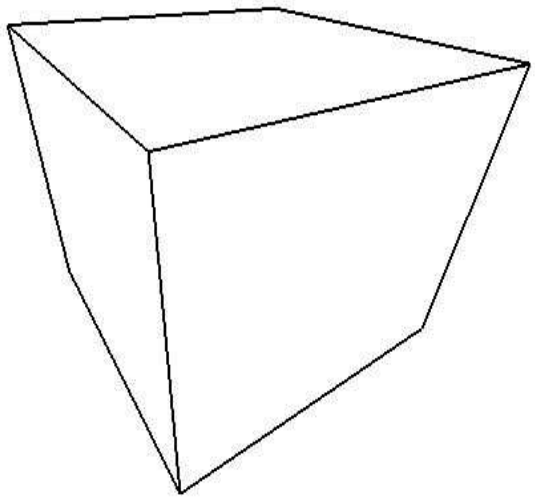
2012-2013

1 Introduction

But

- Voir uniquement ce qui doit être vu...
- Exemple : un cube représenté par Brep :





Abordé dans ce chapitre...

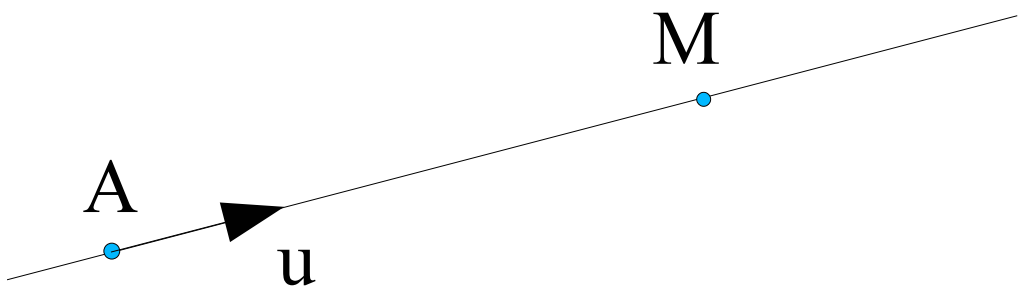
- Technique du « peintre » : tout afficher en affichant les points les plus éloignés d'abord (les points proches recouvrent alors les points éloignés).
  - cas particulier : scènes par facettes et tri par BSP.
- Technique du « depth buffer » : mémoriser en chacun des pixels de l'écran la profondeur du point actuellement affiché.
- Optimisation : élimination des faces arrières.

–

- Remarque : pour le lancer de rayon, l'élimination est intrinsèque à la méthode (trouver le point le plus proche de l'observateur suivant un rayon) $\Rightarrow$ voir chapitre « lancer de rayon » .
- Autres techniques non vues : élimination arêtes, scan-line, ...

2 Eléments fondamentaux

Droite

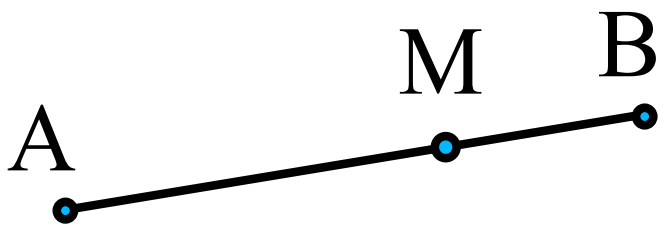


- M décrit la droite $D = (A, u)$ si $\overrightarrow{AM} = \lambda u$ ($\lambda \in \boldsymbol{R}$).
- Comme $\overrightarrow{AM} = M - A$ on a $\boxed{M \in D \Leftrightarrow M = A + \lambda u}$

avec $M = (x, y, z)$, $A = (A_x, A_y, A_z)$ et $u = (u_x, u_y, u_z)$:

$$M = \begin{cases} x = A_x + \lambda u_x \\ y = A_y + \lambda u_y \\ z = A_z + \lambda u_z \end{cases}$$

Segment

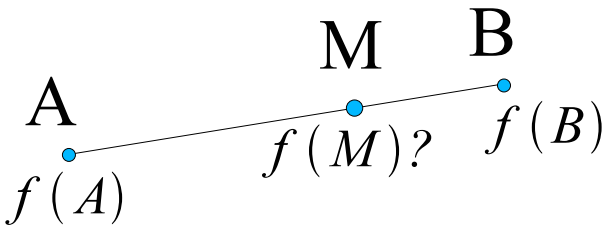


$$M \in [AB] \Leftrightarrow \overrightarrow{AM} = \lambda \overrightarrow{AB} \Leftrightarrow \boxed{M = (1-\lambda)A + \lambda B} \text{ (avec } \lambda \in [0,1])$$

Remarques :

- pour λ donné, M est le barycentre de $(A, 1-\lambda)$ et (B, λ).
- $\lambda = \frac{|AM|}{|AB|}$

Interpolation linéaire

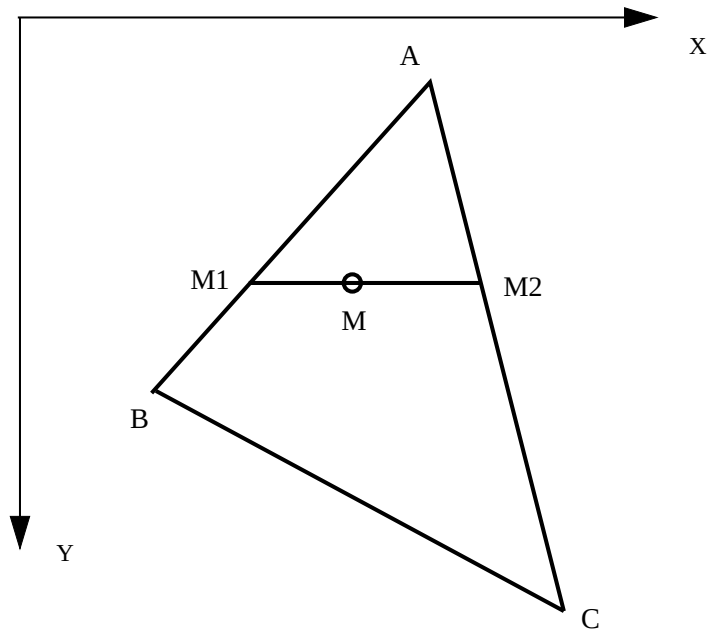


$$\boxed{M = (1-\lambda)A + \lambda B} \text{ (avec } \lambda \in [0,1])$$

Soit f un attribut défini en A et en B. Interpoler linéairement f entre A et B signifie qu'on considère que f varie linéairement entre A et B (i.e. la représentation de f en fonction de λ est une droite).

$$\boxed{\Rightarrow f(M) = (1-\lambda)f(A) + \lambda f(B)}$$

Interpolation bilinéaire



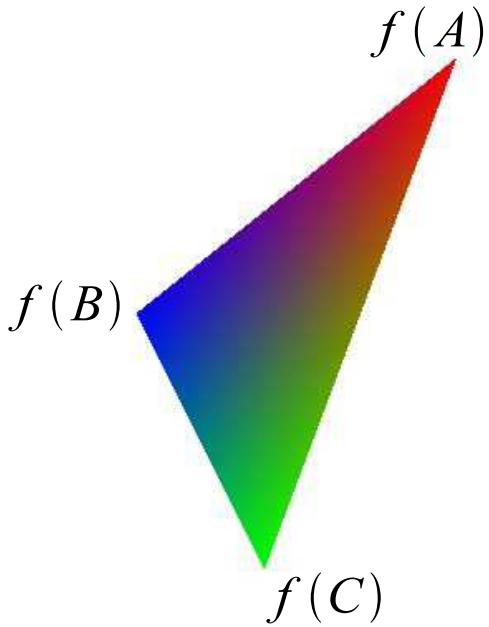
- Interpolation linéaire sur $[AB]$: $M_1 = (1-\lambda_1)A + \lambda_1 B \Rightarrow f(M_1) = (1-\lambda_1)f(A) + \lambda_1 f(B)$
- Interpolation linéaire sur $[AC]$: $M_2 = (1-\lambda_2)A + \lambda_2 C \Rightarrow f(M_2) = (1-\lambda_2)f(A) + \lambda_2 f(C)$
- Interpolation linéaire sur $[M_1, M_2]$:
  $M = (1-\lambda)M_1 + \lambda M_2 \Rightarrow f(M) = (1-\lambda)f(M_1) + \lambda f(M_2)$

Remarques :

- « en dessous »de B, il faut calculer M_1 avec les sommets B et C.
- Remarque : $\lambda_1 = \frac{AM_1}{AB} = \frac{y_{M_1} - y_A}{y_B - y_A}$, ...

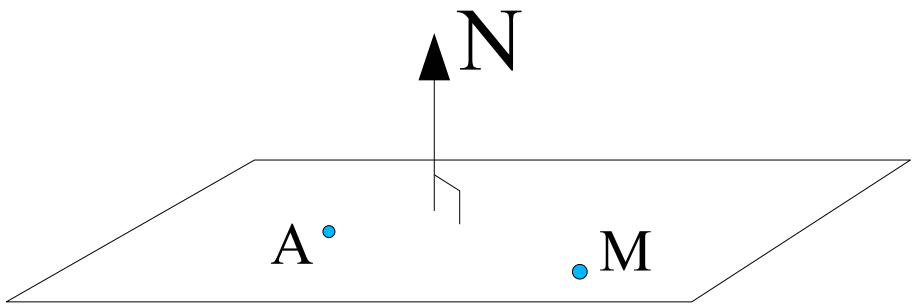
Interpolation bilinéaire : exemple

$f(M)$ est une couleur avec les composantes rouge, vert, bleu $f(M) = (f_r(M), f_v(M), f_b(M))$ et on connait la couleur aux sommets A, B et C. On interpole linéairement f (i.e. on interpole f_r, f_v et f_b) :



$\Rightarrow$ L'interpolation bi-linéaire appliquée aux couleurs est appelée **interpolation de Gouraud**

Plan

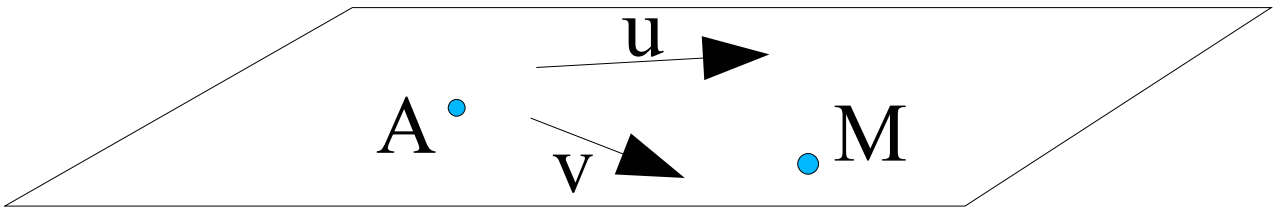


- M décrit le plan $P = (A, n)$ (n=normale donnée au plan) si $\overrightarrow{AM} \cdot n = 0$ ($\overrightarrow{AM}$ est orthogonal à n).
- $\boxed{M \in P \Leftrightarrow M \cdot n - A \cdot n = 0}$

En développant si $M = (x, y, z)$, $n = (a, b, c)$ et $A = (A_x, A_y, A_z)$:

$$ax + by + cz + d = 0 \text{ avec } d = -A \cdot n = -(aA_x + bA_y + cA_z)$$

Plan (2)



- M décrit le plan $P = (A, u, v)$ (u et v appartiennent au plan et sont non colinéaires) si $\overrightarrow{AM} = \alpha u + \beta v$ avec $(\alpha, \beta) \in \boldsymbol{R}^2$
- $\boxed{M \in P \Leftrightarrow M = A + \alpha u + \beta v}$

Si $M = (x, y, z)$, $A = (A_x, A_y, A_z)$, $u = (u_x, u_y, u_z)$ et $v = (v_x, v_y, v_z)$

$$M = \begin{cases} x = A_x + \alpha u_x + \beta v_x \\ y = A_y + \alpha u_y + \beta v_y \\ z = A_z + \alpha u_z + \beta v_z \end{cases}$$

Exercices

- Intersection d'une droite (A, u) avec un plan (P, n) ?
- Distance d'un point M à un plan (P, n) ?
- Distance d'un point M à une droite (A, u) ?
- Distance d'un point M à un segment (A, B) ? (étude de cas).

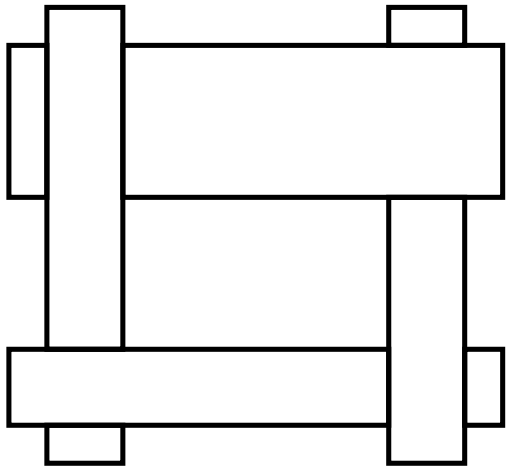
3 Algorithme du « peintre » (newell)

Principe

- Il « suffit » de tracer les points du plus éloigné au plus proche (par rapport à l'observateur).
- Si la scène est constituée de polygones :
  1. Trier les facettes de la plus éloignée vers la plus proche.
  2. Afficher les facettes selon cet ordre (recouvrement des parties cachées par les polygones plus proches).

Principal problème

- Il faut trouver un critère pour comparer 2 facettes entre elles (pour pouvoir les trier).
- Remarque : à priori cela est impossible sans décomposer (i.e. couper) les polygones :



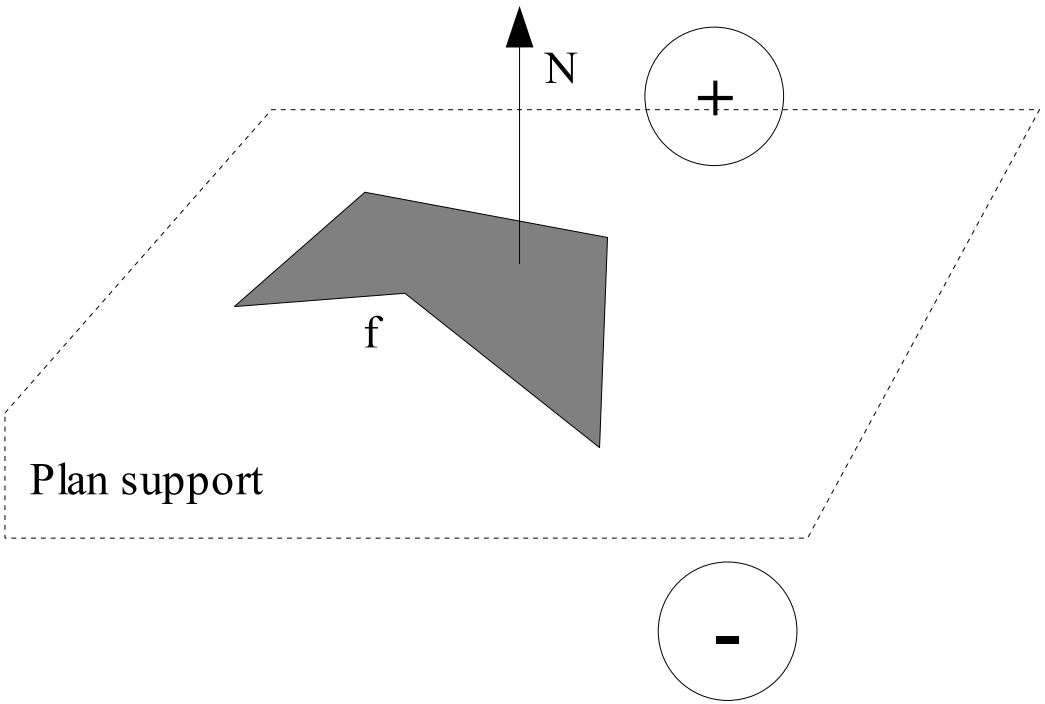
Première approche

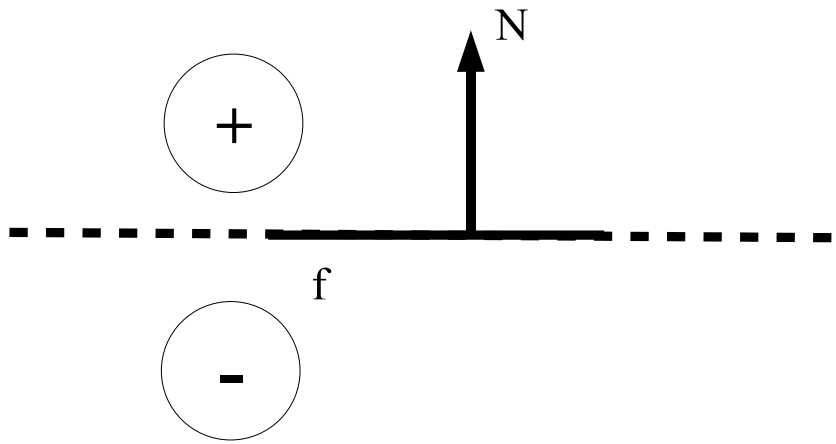
- En oubliant le cas particulier précédent :
  - Considérer le barycentre des facettes : principe faux et facile à mettre en défaut (pourrait être envisagé pour des scènes particulières et en toute première approximation).
  - Considérer tous les sommets (exercice - le faire en 2D pour comparer deux arêtes : principe du scan-line).
- Quelque soit l'approche $\Rightarrow$ tous les cas mènent à la nécessité de couper les faces.
- « Une » bonne approche $\Rightarrow$ tri par BSP.

4 Arbre BSP (Binary Spatial Partition)

Espaces positif et négatif

- Soit un polygone (ou facette) *f* et une normale arbitraire *n*. Le plan porteur du polygone partage l'espace en deux sous espaces : le « coté » de la normale est appelé sous espace positif (l'autre est le sous espace négatif).





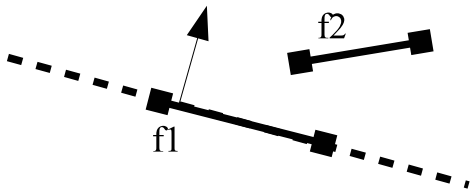
Analogie 2D (ou vue orthogonale à f)

Points positif et négatif

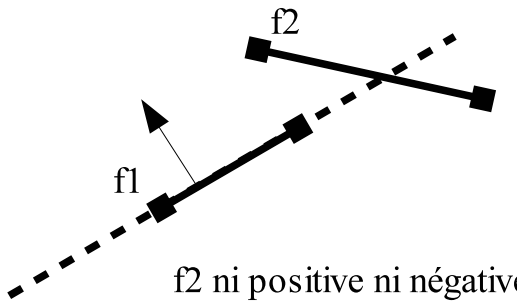
- Un point P est dit positif par rapport à f s'il est dans le demi-espace positif de f.
- remarque : Si f est définie par sa normale n et un point A alors $P = (X, Y, Z)$ est positif ssi $AP \cdot n >= 0$ (négatif sinon).
- remarque : le cas où P appartient au plan est inclus dans le cas `positif`.

Localisation d'une facette

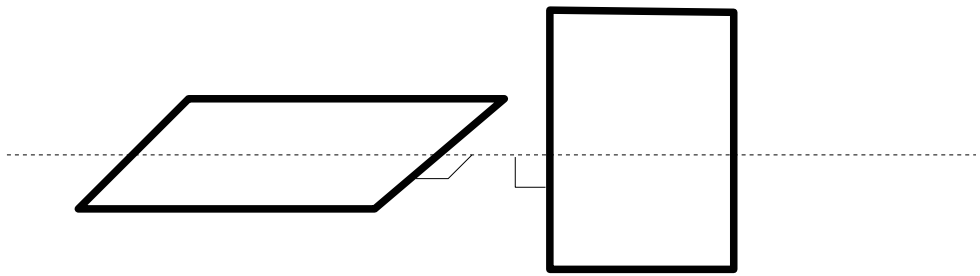
- Soit 2 facettes f_1 et f_2. f_2 est dite positive par rapport à f_1 si tous les points de f_2 sont positifs (si tous les points sont négatifs, elle est dite négative).
- Remarque : il suffit de considérer le signe des sommets de f_2 par rapport à f_1.
- Remarque : f_2 peut être ni positive, ni négative par rapport à f_1.



f2 positive par rapport à f1

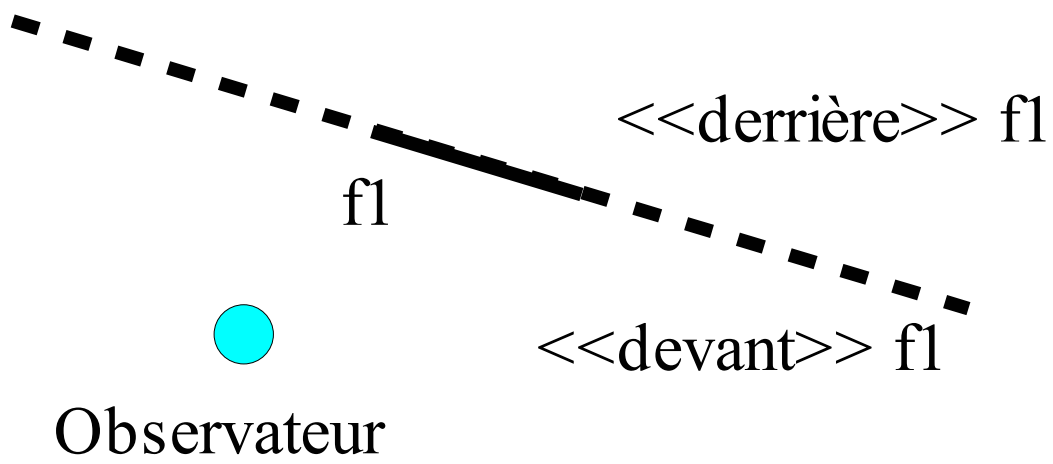


f2 ni positive ni négative



Propriété pour l'élimination

- On suppose que f_2 est soit positive, soit négative par rapport à f_1 (plus tard on traitera le cas contraire en coupant f_2 par le plan de f_1).
- Propriété :
  - Si l'observateur et f_2 sont de mêmes signes (par rapport à f_1) alors f_1 ne peut pas occulter f_2 (i.e. f_2 se trouve « devant » f_1 et donc f_1 doit être tracée avant f_2 pour l'algo du peintre).
  - Si l'observateur et f_2 sont de signes contraires alors f_2 ne peut pas occulter f_1 (i.e. f_2 se trouve « derriere » f_1 et donc f_2 doit être tracée après f_1).



Arbre BSP

- Chaque noeud est identifié à une facette f.
- Chaque noeud f possède au plus deux sous-arbres (arbre binaire) :
  - Un sous arbre positif dont tous les noeuds (i.e. toutes les facettes) sont positifs par rapport à f.
  - Un sous arbre négatif dont tous les noeuds sont négatifs par rapport à f.

Algo du peintre suivant le BSP

- Il suffit de faire un parcours infixe de l'arbre : soit $B = (f, \mathit{negatif}, \mathit{positif})$ un arbre BSP alors :

```
Afficher(B) {
Si B non vide alors
  Si f(Observateur)<0 alors
    Afficher(Positif); Afficher(f); Afficher(Negatif);
  Sinon
    Afficher(Negatif); Afficher(f); Afficher(Positif);
  Fin Si
}
```

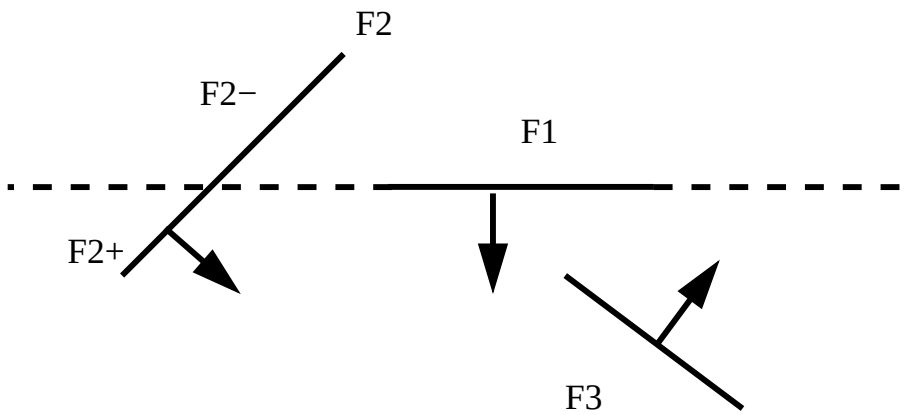
Construction

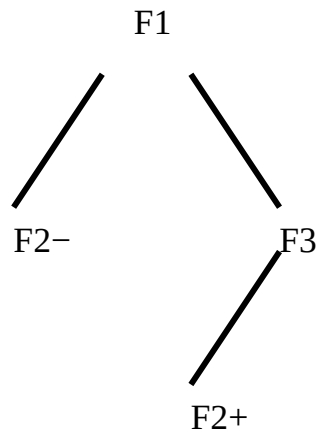
Soit une scène constituée d'une liste L de facettes.

- Prendre une facette f arbitraire de L.
- Construire la liste $L+$ des facettes positives à f et construire la liste $L-$ des facettes négatives.
- Construre le BSP $B+$ de $L+$ et le BSP $B-$ de $L-$ (récursivement).
- Le BSP de L est $(f, B-, B+)$.

Remarques :

- Pour toute facette f_i ni positive, ni négative : il faut couper f_i par le plan porteur de f. On obtient alors des f_i+ (à inclure dans $L+$) et des f_i- (à inclure dans $L-$).
- $\Rightarrow$ Voir pseudo-code complet en TD.





Remarques sur les BSP

- On peut faire l'analogie avec le tri par quick sort (pivot f, couper en deux listes « plus petit » , « plus grand »), mais, ici, on garde explicitement tout l'arbre de construction (la notion d'ordre change à chaque noeud).
- Le choix du pivot peut être plus judicieux pour « tenter » d'obtenir l'arbre équilibré.
- L'arbre est indépendant de la position de l'observateur (pas de reconstruction à faire lorsque l'observateur se déplace dans une scène statique).
- Par contre : si des objets sont en mouvement il faut refaire l'arbre (ou « tenter » de traiter à part les objets mobiles et immobiles).
- Les BSP (ou raisonnement similaires) peuvent être utilisés pour d'autres objectifs que pour le peintre (optimisation pour la collision, optimisation d'occlusion,...).

Avantage-Inconvénient du peintre

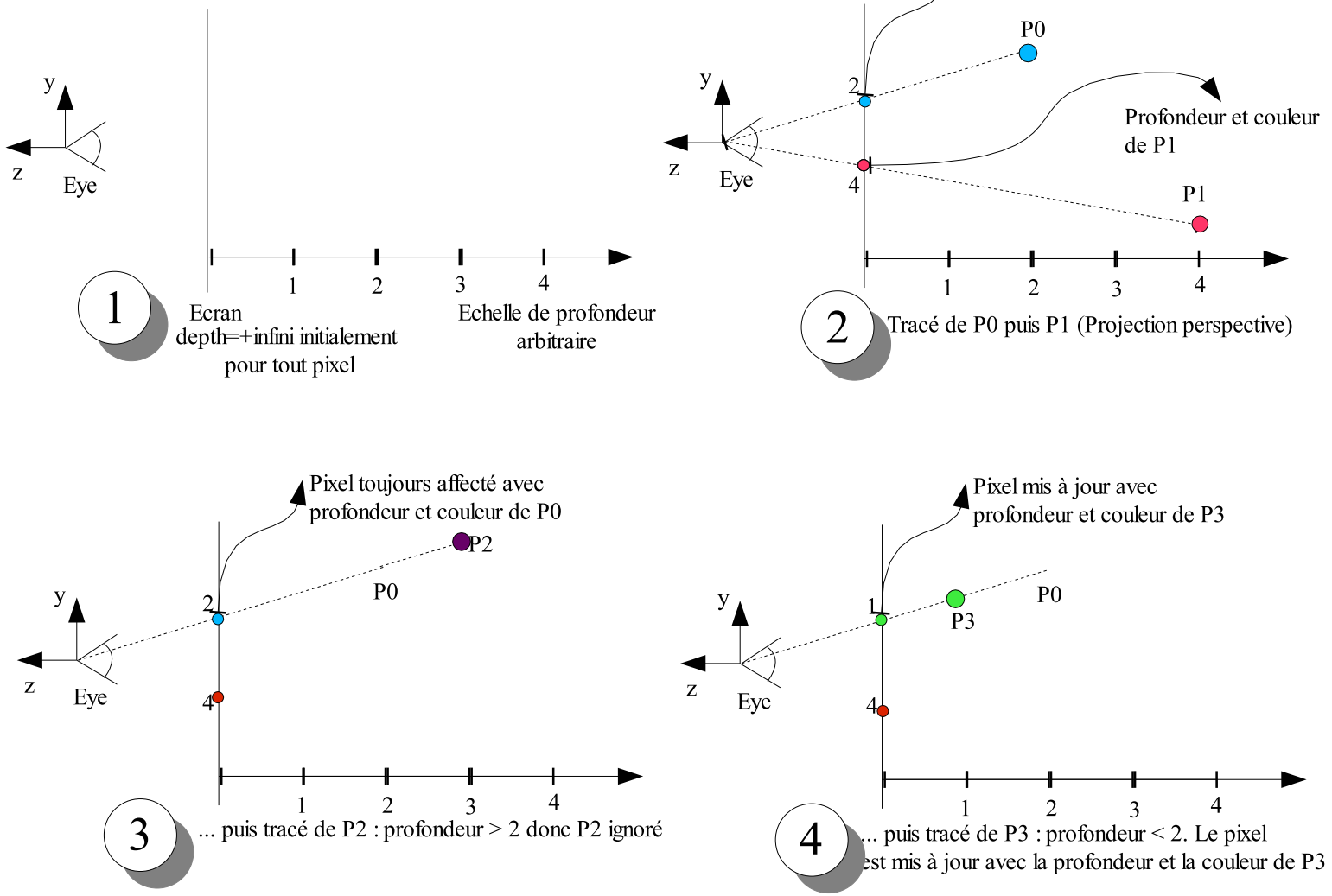
- Il faut tout afficher, même ce qui ne sera pas vu.
- Le tri (par BSP par exemple) peut s'avérer lourd lors d'animations (scène avec objets mobiles).
- Principe du peintre obsolète... mais pas les raisonnements de tris, de localisation basés sur les BSP.

5 Depth Buffer

Introduction - Principe

- Algorithme du Depth-buffer = algorithme du « tampon de profondeur » . Appelé aussi Z-Buffer.
- Le raisonnement se fait sur l'écran en 2D (i.e. dans l'espace des pixels dit « espace image »). A opposer au peintre qui se fait dans l'espace 3D (dit « espace objets »).
- A chaque pixel de l'écran est affecté une valeur de profondeur :
  - représente la profondeur du point qui est projeté actuellement en ce pixel.
  - si un nouveau point se trouve projeté sur le même pixel : on compare sa profondeur avec la profondeur actuelle du pixel $\Rightarrow$ visible ou non visible.
- $\Rightarrow$ Dédié au rendu projectif.
- Appliqué par les cartes 3D actuelles.

Principe sur exemple



Fragments

- Chaque pixel peut être affecté, ou influencé, par des attributs : couleur, valeur de profondeur,...
- L'ensemble de ces attributs est appelé fragment.
- On différencie :
  - Le fragment destination : est celui qui est affecté au pixel (i.e. la valeur courante).
  - Le fragment source : est celui qui est en train d'être tracé (i.e. la valeur qui « arrive »), et qui est donc susceptible de mettre à jour le pixel.

Algorithme du Depth Buffer

```
Effacer Ecran (initialiser couleur de fond et
               valeurs de depth à +infini
               pour chaque pixel)

Pour tout point P à tracer (source) faire
  Déterminer les coordonnées du pixel (xi,yi)
  Calculer le depth du fragment source (depth de P)
  Calculer la couleur du fragment source

  -- pipeline pixel :
  Si depth(source) < depth(destination) Alors
     depth(destination) <-- depth(source)
     couleur(destination) <-- couleur(source)
  Fin Si
  --

Fin Pour
```

Remarques

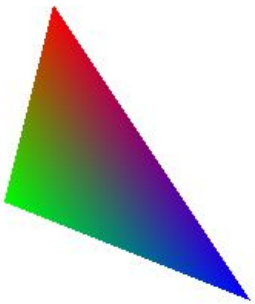
- Inutile de trier : les pixels peuvent être affichés dans n'importe quel ordre.
- Pour la profondeur il suffit de comparer selon les coordonnées $-Z$ des points dans le repère Eye (d'où le nom de Z-Buffer).
- Comme pour le peintre : des points peuvent être affichés inutilement.
- Contrairement au peintre : des points peuvent être éliminés par le test, ce qui évite leur l'affichage.
- Extrêmement simple (cablé sur les cartes graphique) et efficace.
- Il faut savoir traduire le « pour tout point P à tracer... » : les polygones sont bien adaptés (remplissage et calcul incrémental du depth et couleur).
- Historiquement : la technique ne s'est pas imposée immédiatement (pour le rendu projectif) pour des raisons de coût mémoire.

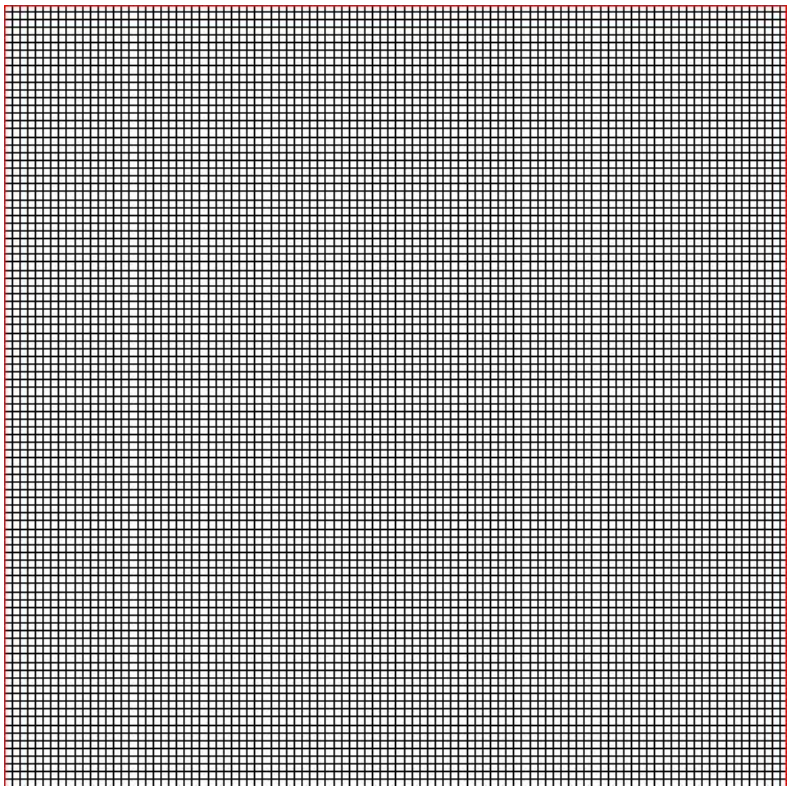
6 Depth buffer en rendu projectif

Contexte

On considère le contexte suivant (contexte d'OpenGL) :

- La scène à visualiser est constituée de triangles 3D.
- Pour un triangle $(A_{eye}, B_{eye}, C_{eye})$ donné son affichage consiste à :
  - Projeter le triangle $(A_{eye}, B_{eye}, C_{eye})$ sur l'écran pour obtenir (A_p, B_p, C_p) (triangle 2D)
  - Puis remplir pixel par pixel le triangle (A_p, B_p, C_p) (balayage du triangle 2D).

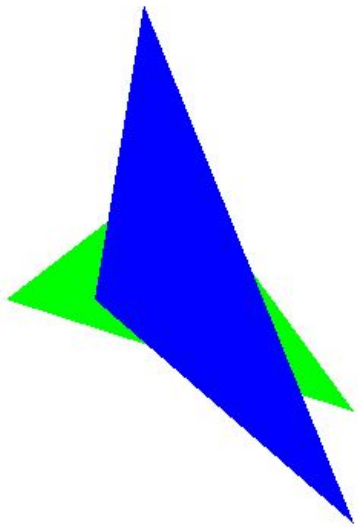


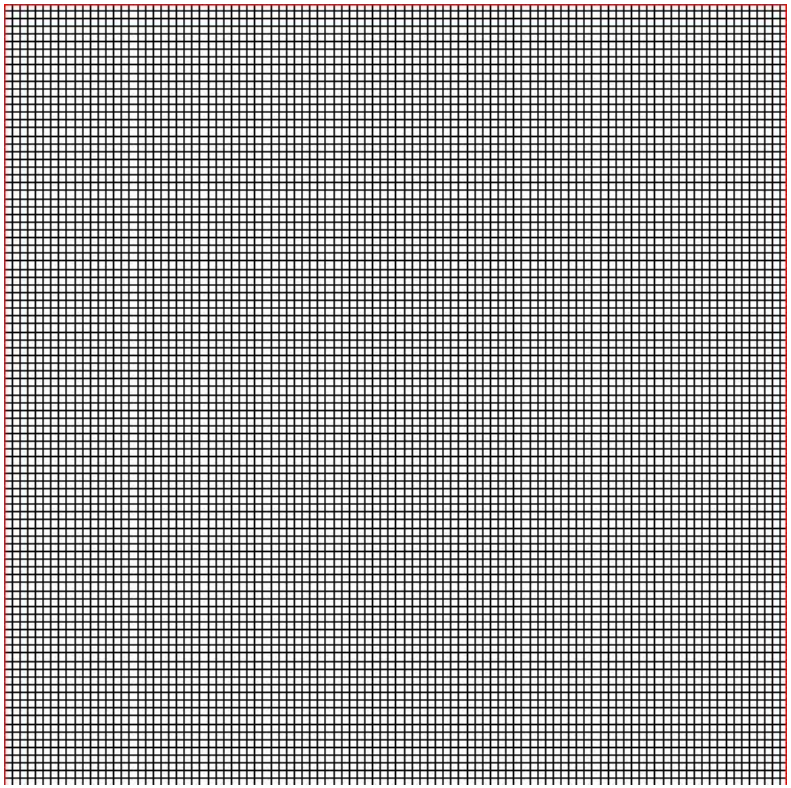


Elimination des parties cachées en OpenGL

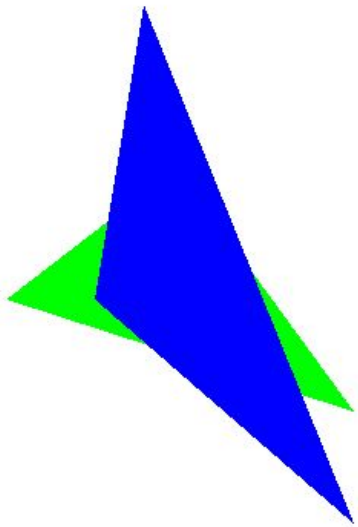
L'élimination des parties cachées est assurée pour chaque pixel tracé par :

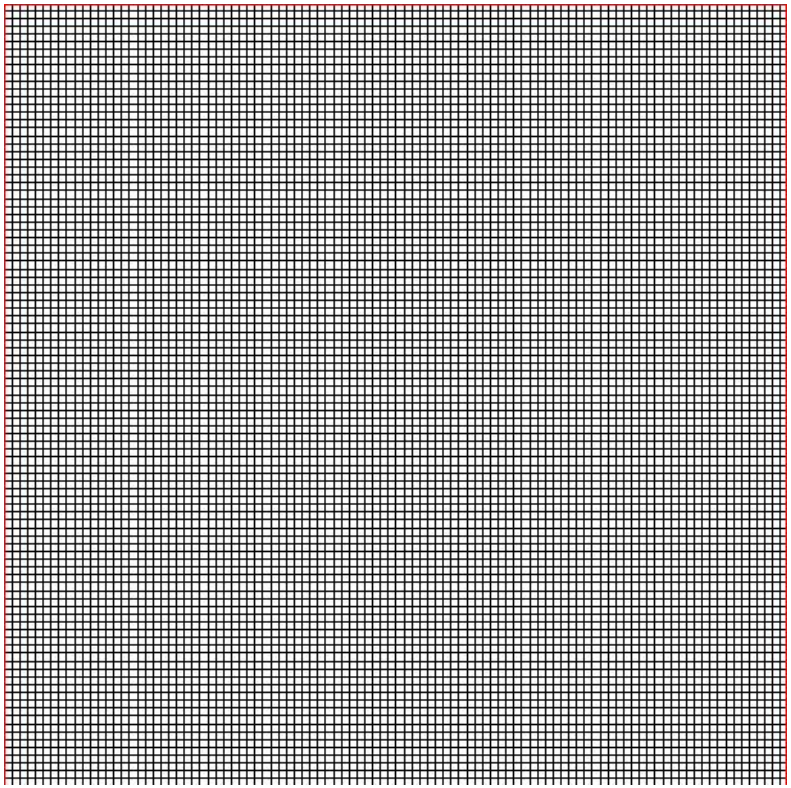
```
Si depth(source)<depth(destination) alors
 depth(destination) <- depth(source) (Mise à jour de la profondeur).
 color(destination) <- color(source) (Mise à jour de la couleur).
Fin Si
```





Autre ordre





Comment est calculée la profondeur de chaque pixel tracé (i.e. `depth(source)`) ?

Projection

- L'élimination par depth se fait en coordonnées écran. Comment passer de P_{eye} à P_{screen} ?
- Spécification en OpenGL du passage en coordonnées écran :

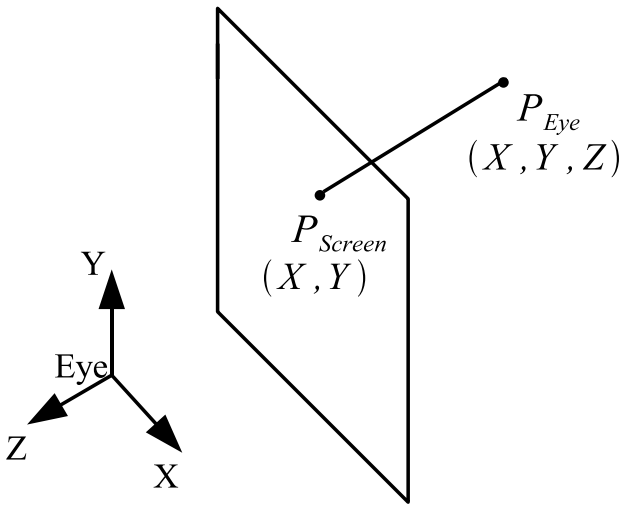
```
void initGL() {
  // définition de la matrice de projection (projection perspective ici)
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glFrustum(-1,1,-1,1,0.1,100);

  // définition de la matrice de transformation (identité = repère courant sur Eye).
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glViewport(0,0,width,height); // fen\^etre graphique d'OpenGL
  ...
}
```

Projection orthogonale et depth

- Principe :



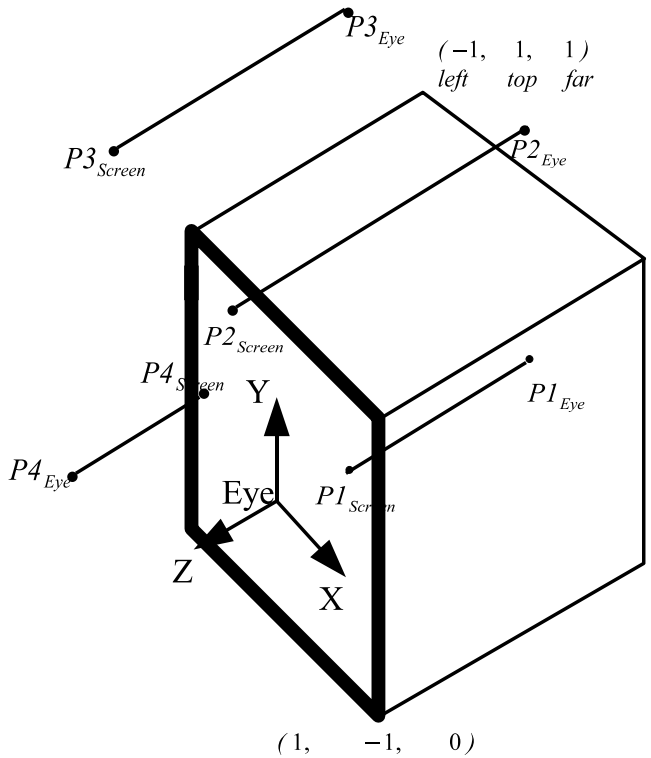
$$\begin{pmatrix} x_{screen} \\ y_{screen} \end{pmatrix} = \begin{pmatrix} x_{eye} \\ y_{eye} \end{pmatrix}$$

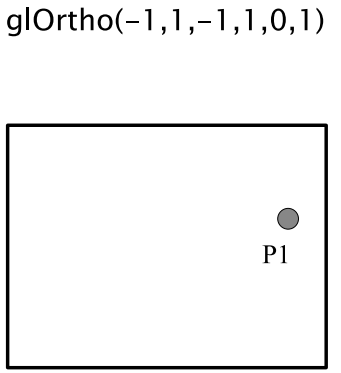
Exercice : quelle est la matrice homogène pour passer de P_{eye} à P_{screen} sachant que l'écran est à une distance `near` de `eye` ?

- Ne suffit pas :
  - Comment en déduire des coordonnées écran (tenir compte de ses dimensions) ?
  - Absence de l'information de profondeur pour les points projetés ($P_{screen} = d$ pour tout point).

Projection orthogonale et depth

- Solution adoptée pour calculer la projection :
  - Rester en coordonnées homogènes pour le calcul du projeté des sommets $\Rightarrow$ conserve l'information de profondeur.
  - Diviser par w pour passer en coordonnées 3D (x et y correspondent aux coordonnées sur le plan de l'écran, et z à la profondeur).
  - S'assurer que toutes les coordonnées (incluant la profondeur) sont dans l'intervalle $[-1, 1]$ (coordonnées dites normalisées).
  - $\Rightarrow$ définir un volume de visualisation :





Ecran obtenu

Projection orthogonale et depth

$$P_p = M_{\text{PROJECTION}} P_{Eye}$$

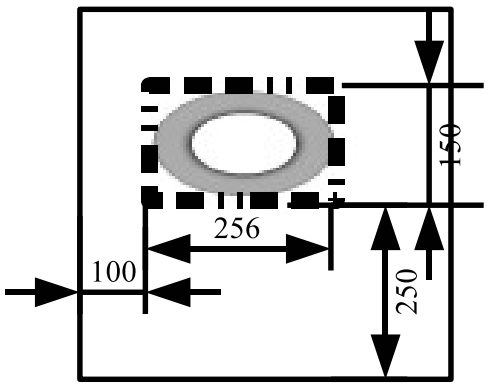
avec

$$M_{\text{PROJECTION}} = \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- c'est la matrice calculée par `glOrtho(left,right,bottom,top,near,far)`.
- résulte de l'application d'une rêgle de 3 : passer de $x_{eye} \in [left, right]$ à $x_p \in [-1, 1]$.

Calcul du depth pour chaque pixel

- Phase géométrique :
  - Tout sommet subit $P_p = M_{\text{PROJECTION}} M_{\text{MODELVIEW}} P_{Local}$
  - Les coordonnées normalisées sont obtenues en divisant P_p par sa coordonnée homogène.
  - On obtient les coordonnées entières à l'écran de P_p en appliquant un `viewport` (définition de la fenêtre graphique) :
  - Exemple :



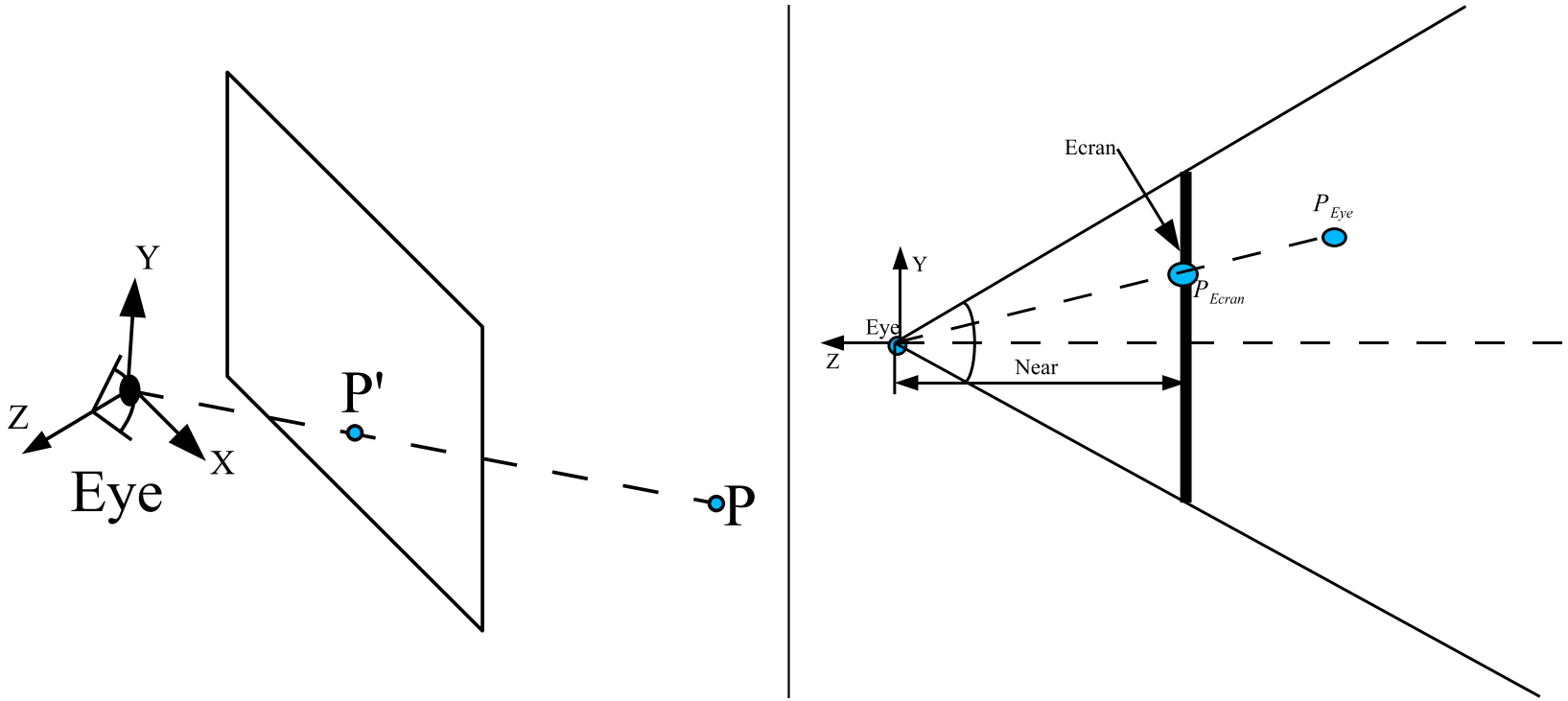
glViewport(100,250,256,150)

`glViewport(x_min,y_min,width,height)`

$$\Rightarrow \begin{cases} x_{ecran} = (x+1)\frac{width}{2} + x_{min} \\ y_{ecran} = (y+1)\frac{height}{2} + y_{min} \end{cases}$$

- Rasterization :
  - remplissage pixel par pixel en interpolant linéairement la profondeur des sommets

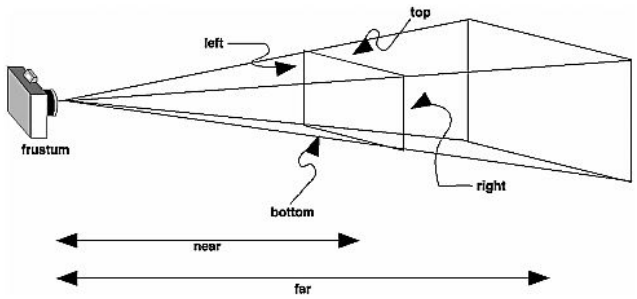
Projection perspective et depth



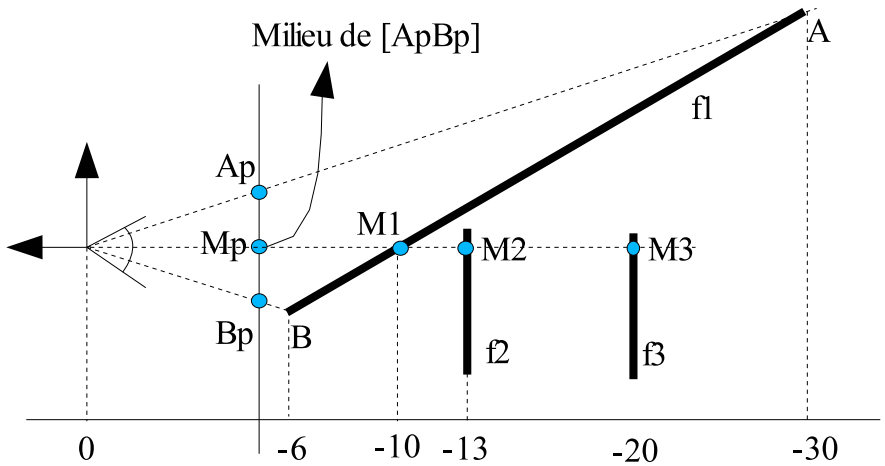
Exercice : quelle est la matrice de passage $M_{Screen \rightarrow Eye}$?

Projection Perspective et depth

- Définition d'un volume de visualisation pour normaliser (en OpenGL : `glFrustum(left,right,bottom,top,near,far)`).



- Tracer selon le même schéma (phase géométrique, rasterization avec interpolation bilinéaire et test du depth).
- $\Rightarrow$ Problème :



- $\Rightarrow$ la profondeur z_p n'est pas linéaire (i.e. interpoler donne une approximation), et cette

Projection Perspective et depth

On peut montrer par contre que $\frac{1}{z}$ est linéaire.
$\Rightarrow$ lorsqu'on projette les sommets, on ne conserve donc pas z pour l'élimination des parties cachées mais $\frac{1}{z}$:
$P_p = M_{\text{PROJECTION}} P_{Eye}$ avec

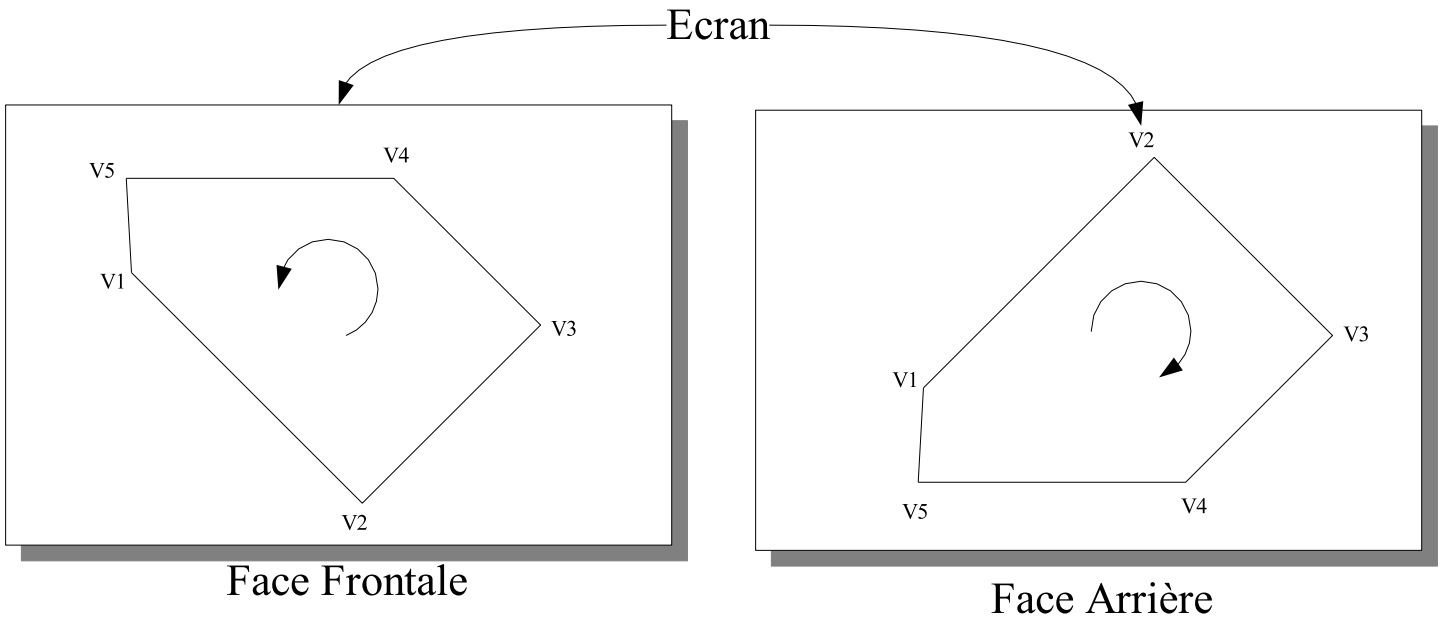
$$M_{\text{PROJECTION}} = \begin{pmatrix} 2\frac{near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & 2\frac{near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -2\frac{far*near}{far-near} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Pour interpoler $\frac{1}{z_{eye}}$ il suffit d'interpoler $\frac{z_p}{w_p}$.
Exercice : retrouver cette matrice (on part de $z \in [near, far]$, donc $\frac{1}{z} \in [\frac{1}{far}, \frac{1}{near}]$ et on reporte dans l'intervalle $[-1, 1]$).

7 Une optimisation : Elimination des parties arrières

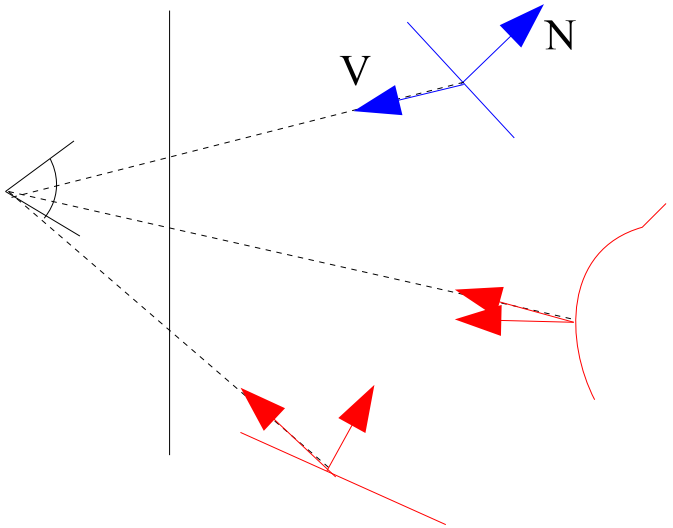
Facettes frontales/arrières

- On raisonne dans le cadre d'une scène polygonale visualisée par projection.
- Une facette est dite frontale si son polygone projecté sur l'écran est orienté direct (elle est dite arrière sinon).
- Autrement dit : la facette est frontale si, à l'écran, on « voit » sa face directe.



Remarques

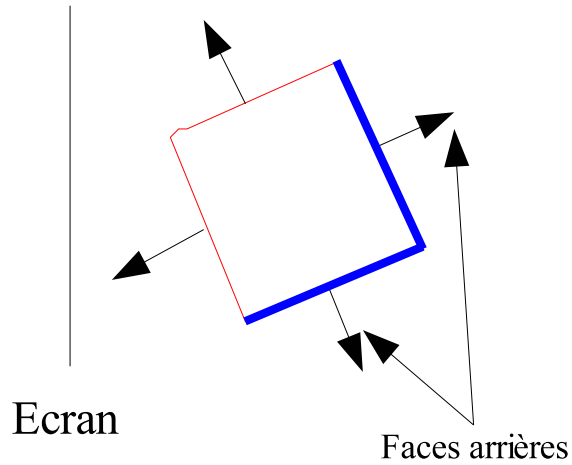
- Pour les facettes convexes $(V_1, V_2, V_3, \ldots)$, le signe de ${V_1}_{\text{proj}}\, {V_2}_{\text{proj}} \wedge {V_2}_{\text{proj}}\, {V_3}_{\text{proj}}$ (= déterminant) suffit pour déterminer si la facette est frontale ou non.
- Un point P est dit frontal s'il est élément d'une facette frontale.
- Si N est la normale directe (appliquée en P), alors P est frontal ssi $V.N > 0$.
- ($\Rightarrow$ on peut appliquer la notion frontal/arrière à des objets non décrits par polygones).

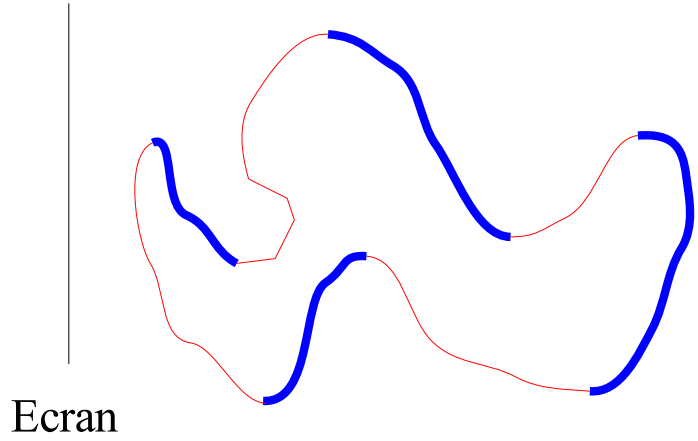


Faces Frontales et
Face Arrière

Propriété

- Pour une surface close (frontière entre l'intérieur et l'extérieur d'un volume) bien orientée (faces directes vers l'extérieur) et pour un observateur placé à l'extérieur du volume :
  - une facette arrière correspond à la face intérieure au volume (i.e. le coté du polygone qui fait face à l'observateur est intérieur au volume).
  - $\Rightarrow$ les facettes arrières sont donc nécessairement occultées (i.e. l'observateur ne voit pas l'intérieur).
  - ... donc inutile de les tracer $\Rightarrow$ élimination des faces arrières ou « back face culling ».





Remarques

- Il s'agit d'une optimisation : l'élimination des faces arrières ne suffit pas pour l'élimination des parties cachées.
- A appliquer uniquement aux volumes bien orientés...
- En OpenGL :
  - `glCullFace(GL_BACK)` ou `glCullFace(GL_FRONT)` pour indiquer les faces à éliminer
  - `glEnable(GL_CULL_FACE)` pour activer l'élimination (les sommets sont toujours projetés, mais les faces éliminées ne subissent pas la phase de rasterization).