

Coordonnées homogènes

- ✓ Outil géométrique très puissant :
 - Rend les transformations projectives linéaires
- ✓ Un point 2D devient un vecteur à 3 coordonnées :

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Coordonnées homogènes

- ✓ Deux points sont égaux si et seulement si :
 - $x'/w' = x/w$ et $y'/w' = y/w$
- ✓ $w=0$: points « à l'infini »
 - Très utile pour les projections, et pour certains types de modélisations (courbes splines)

Translations en c. homogènes

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \begin{cases} \frac{x'}{w'} = \frac{x}{w} + t_x \\ \frac{y'}{w'} = \frac{y}{w} + t_y \end{cases}$$

$$\begin{cases} x' = x + wt_x \\ y' = y + wt_y \\ w' = w \end{cases}$$

Changement d'échelle

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

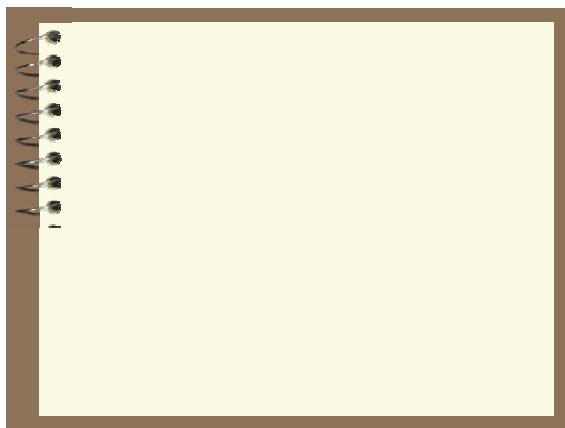
$$\begin{cases} \frac{x'}{w'} = S_x \frac{x}{w} \\ \frac{y'}{w'} = S_y \frac{y}{w} \end{cases}$$

$$\begin{cases} x' = s_x x \\ y' = s_y y \\ w' = w \end{cases}$$

Rotation

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

$$\begin{cases} \frac{x'}{w'} = \cos \theta \frac{x}{w} - \sin \theta \frac{y}{w} \\ \frac{y'}{w'} = \sin \theta \frac{x}{w} + \cos \theta \frac{y}{w} \end{cases}$$

$$\begin{cases} x' = \cos \theta x - \sin \theta y \\ y' = \sin \theta x + \cos \theta y \\ w' = w \end{cases}$$


Et en 3 dimensions ?

- ✓ C'est pareil
- ✓ On introduit une quatrième coordonnée, w
 - Deux vecteurs sont égaux si :
 $x/w = x'/w'$, $y/w = y'/w'$ et $z/w = z'/w'$
- ✓ Toutes les transformations sont des matrices 4x4

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Rotations en 3D

- ✓ Rotation : un axe et un angle
- ✓ La matrice dépend de l'axe et de l'angle
- ✓ Les rotations autour d'un axe de coordonnées ont une expression simple
 - Les autres rotations s'expriment comme combinaison de ces rotations simples

Rotation around x -axis

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

x-axis is unmodified

Sanity check: a rotation of $\pi/2$ should change y in z , and z in $-y$

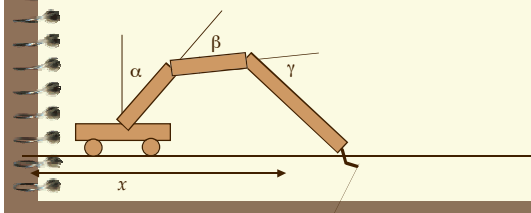
$$R_x\left(\frac{\pi}{2}\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

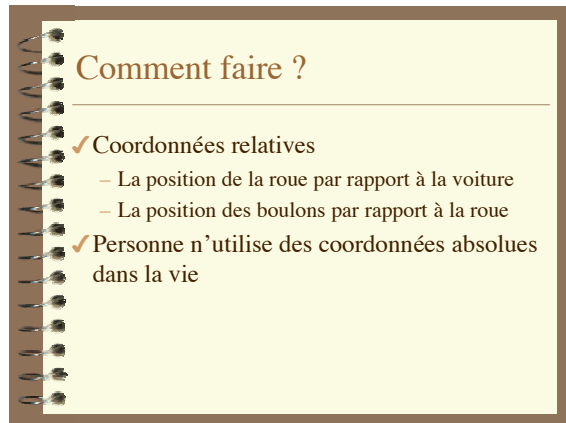
Attention !



- ✓ La multiplication de matrice n'est pas commutative
- ✓ L'ordre des transformations est important
 - Rotation puis translation aura un effet très différent de translation puis rotation
 - L'ordre des angles d'Euler est un grand classique
 - Une source de bugs **très** courante

Definition d'objets complexes





Comment faire ?

- ✓ Coordonnées relatives
 - La position de la roue par rapport à la voiture
 - La position des boulons par rapport à la roue
- ✓ Personne n'utilise des coordonnées absolues dans la vie



Coordonnées relatives

- ✓ Utiliser les coordonnées relatives :

Exemple d'implémentation

- Set transformation as projection matrix
- translate by x (concatenate translation matrix with transformation matrix)
- draw car body
- save transformation matrix
 - translate+rotate
 - draw first wheel
- restore transformation matrix
- save transformation matrix
 - translate+rotate
 - draw second wheel
- restore transformation matrix

Définition hiérarchique

- ✓ Comment savoir quelle est la bonne transformation ?
- ✓ Comment savoir qu'il est temps de revenir à la transformation précédente ?
- ✓ Définition hiérarchique des objets
- ✓ Dessiner l'objet = traversée de la hiérarchie

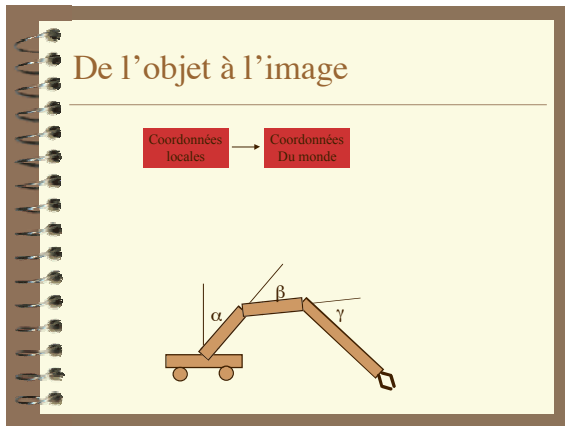
Objet défini hiérarchiquement

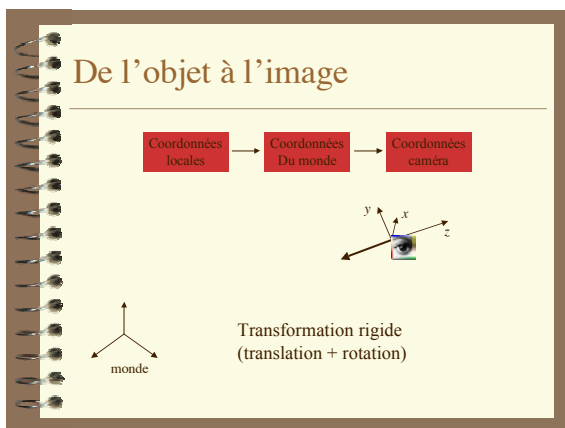
```

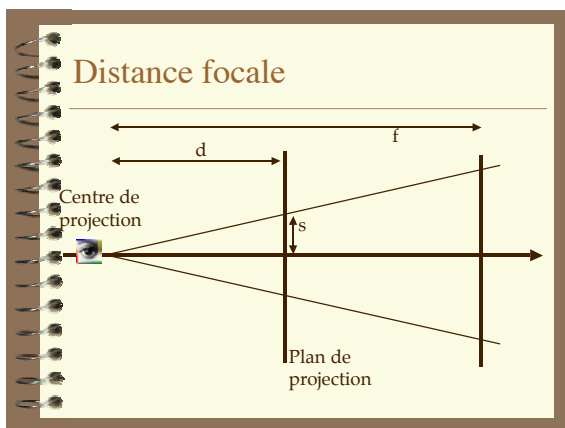
    graph TD
      A[transl. x] --> B[body]
      A --> C[wheel]
      B --> D[transl.]
      B --> E[transl.]
      B --> F[transl.]
      D --> G[1st wh.]
      E --> H[2nd wh.]
      F --> I[Rot. a.]
      I --> J[2nd arm]
      J --> K[...]
    
```

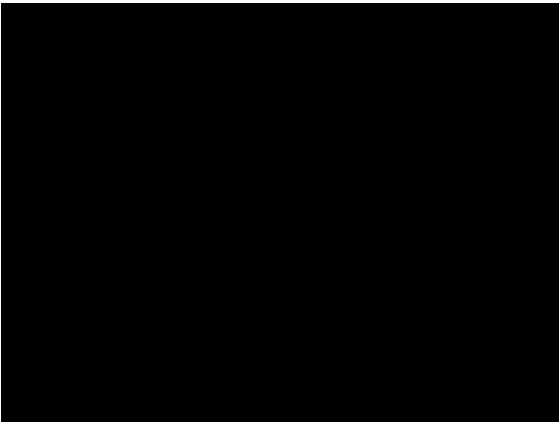
Objets ré-utilisables

- Une seule procédure pour toutes les roues









Transformation de projection

- ✓ Volume « normalisé »
- ✓ Conserve les équations de plans

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-s}{d(1-d/f)} & \frac{s}{(1-d/f)} \\ 0 & 0 & s/d & 0 \end{bmatrix}$$

De l'objet à l'image

```

    graph LR
      A[Coordonnées locales] --> B[Coordonnées Du monde]
      B --> C[Coordonnées caméra]
      C --> D[Coordonnées Ecran]
      D --> E[Coordonnées Fenêtre]
  
```

- purement 2D
- Translation et mise à l'échelle
- Coordonnées en « pixels »

« pipeline graphique »

```

    graph LR
      A[Coordonnées locales] --> B[Coordonnées Du monde]
      B --> C[Coordonnées caméra]
      C --> D[Coordonnées Ecran]
      D --> E[Coordonnées Fenêtre]
      F[Objet 3D] --> A
  
```

- ✓ Composition des transformations
- ✓ Positionnement des opérations
 - Clipping
 - Vecteur normal, calculs d'éclairage

Autres positions de la caméra

- ✓ Position de la caméra ultra-simple
 - Plan image sur $z=0$, vision suivant z
- ✓ Comment passer à un modèle général ?
 - Point de vue quelconque, direction quelconque...
- ✓ Translations, rotations
 - On est ramené au cas précédent

Élimination des parties cachées

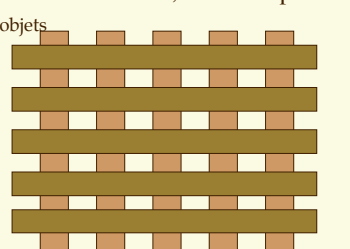
- ✓ Espace image contre espace objet
- ✓ Algorithmes « objet » :
 - Backface culling
 - Tri par la profondeur
 - BSP-trees
- ✓ Algorithmes « image » :
 - Scan-line algorithm
 - Z-buffer

Élimination des parties cachées

- ✓ Détermination des surfaces visibles
- ✓ Complexité : équivalente au tri
 - $O(n \log n)$
- ✓ Résolution dans l'espace image ou dans l'espace objet
 - Espace image : précision limitée, $O(np)$
(p = nombre de pixels, 10^6)
 - Espace objet : précision infinie, $O(n^2)$

Résolution dans l'espace objet

- ✓ Parfois nécessaire, mais complexité la pire :
 - n objets



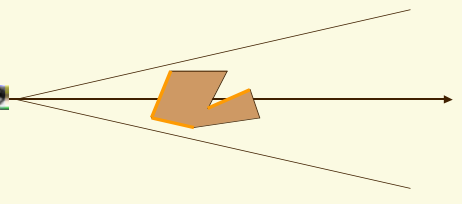
n^2 après la visibilité

Résolution dans l'espace image

- ✓ Permet d'utiliser la cohérence:
 - L'objet visible à un pixel sera sans doute visible sur les pixels voisins
- ✓ Plus rapide en moyenne
- ✓ Cas le pire : $O(np)$

Back Face Culling

- ✓ Éliminer tous les polygones qui ne sont pas tournés vers la caméra :



Back Face Culling : algorithme

- ✓ Si le point de vue n'est pas devant le polygone, on n'affiche pas le polygone
- ✓ Produit scalaire :
 - $(\text{Sommet-PointDeVue}) \cdot \text{normale}$
 - > 0 : on garde le polygone
 - < 0 : on l'élimine

Back Face Culling

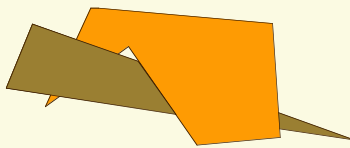
- ✓ Économise 50 % du temps de calcul
 - En moyenne
- ✓ Faible coût par polygone
- ✓ Étape préliminaire pour les autres algorithmes
- ✓ Suffisant pour un seul objet convexe

Tri par la profondeur

- ✓ Dit aussi « algorithme du peintre »
- ✓ On dessine d'abord les polygones les plus lointains, puis les polygones proches de l'œil
- ✓ Les polygones plus proches cachent les polygones lointains
- ✓ Comme un peintre dessine d'abord l'horizon, puis l'arrière-plan, puis le premier plan
- ✓ Problème : définir un ordre sur les polygones
 - Complet, non ambigu

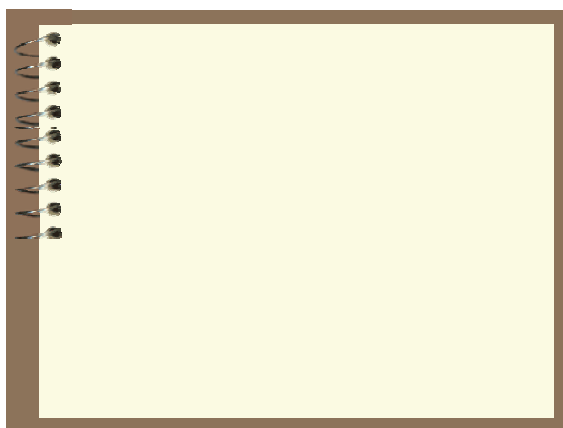
Algorithme simplifié

- ✓ Trier les polygones en fonction de la distance à l'œil
- ✓ Ordre incomplet
- ✓ Ambiguïtés à résoudre :



Algorithme complet

- ✓ Trier les polygones en fonction de leur plus grande coordonnée en z
 - z distance à la caméra suivant la direction de visée
- ✓ Si deux polygones ont des étendues en z qui se recouvrent :
 - On teste :
 - Si les boîtes englobantes de leurs projections sont séparées
 - Si l'un est complètement derrière l'autre
 - Si leurs projections sont séparées
 - Si rien ne marche, on coupe l'un des polygones par le plan de l'autre

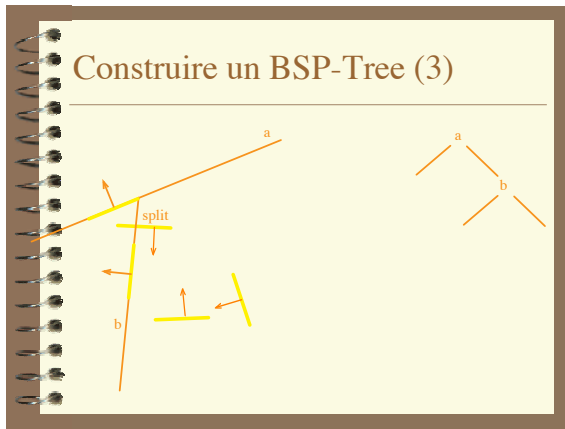


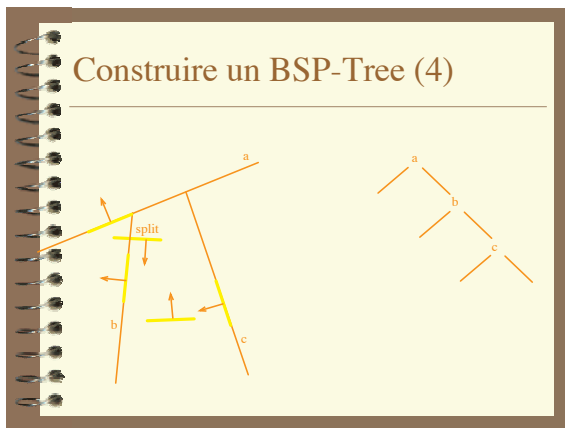
BSP-Trees

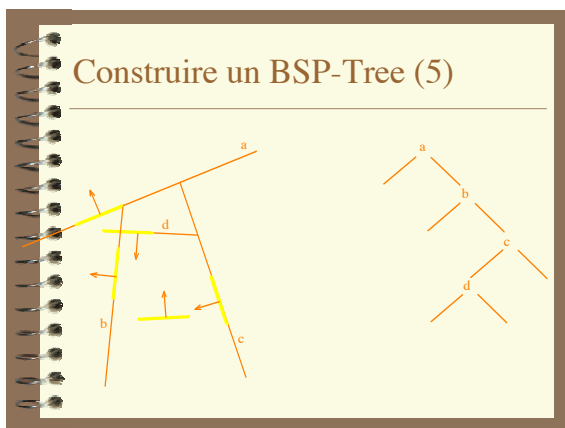
- ✓ On construit un BSP-Tree 3D pour toute la scène
 - En subdivisant les polygones qui sont intersectés par le plan du nœud
- ✓ Affichage des polygones par un parcours de l'arbre:
 - En premier les polygones qui sont derrière le nœud courant (par rapport à la caméra)
 - Puis le nœud courant
 - Puis, les polygones qui sont devant le nœud courant
- ✓ Sorte d'algorithme du peintre
 - Avec un ordre partiel
 - Mais suffisant

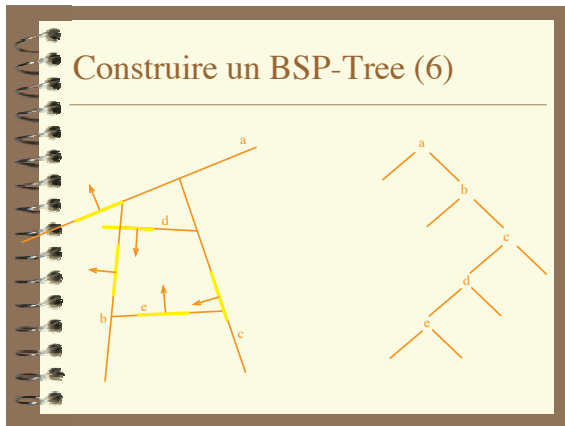
Construire un BSP-Tree (1)

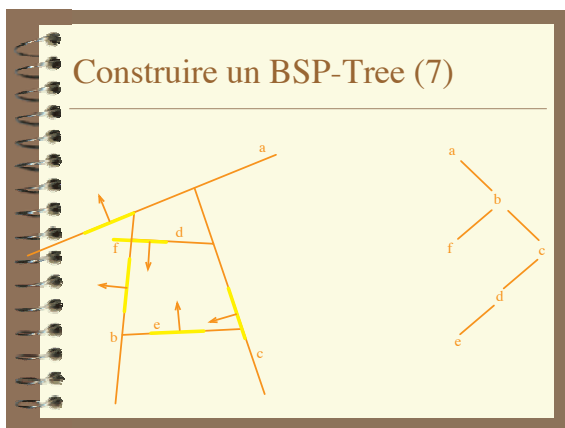
Construire un BSP-Tree (2)

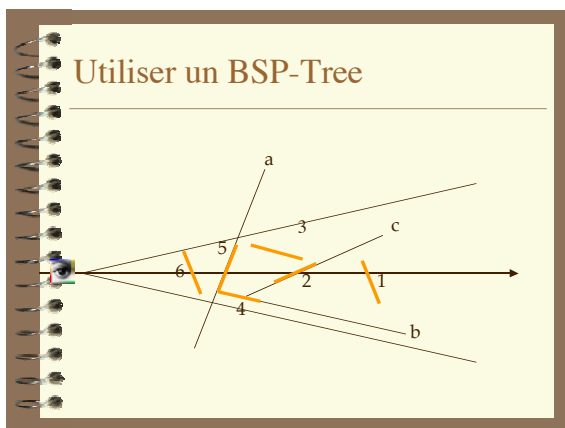












BSP Tree contre algorithme du peintre

- ✓ Le BSP-Tree fait plus de divisions de polygones
- ✓ Mais l'affichage est plus direct
- ✓ BSP-Tree:
 - Pré-traitement plus long
 - Coût mémoire plus élevé
 - Temps par requête (position de la caméra) plus petit
- ✓ Algorithme du peintre:
 - Pas de pré-traitement
 - Temps par requête plus long
- ✓ Dans les deux cas : on affiche toute la scène

Z-Buffer

- ✓ Un tableau, de la taille de l'écran
- ✓ On stocke la valeur maximale de z pour chaque pixel
 - z = direction de visée, exprime la distance à l'oeil
- ✓ Initialisation : tous les pixels à $-\infty$
- ✓ Projection de tous les polygones
 - On met à jour les pixels de la projection du polygone

Z-buffer : algorithme

- ✓ Pour chaque polygone :
 - Projeter le polygone sur le plan image
 - Pour chaque pixel dans la projection du polygone
 - Calculer la valeur de z pour ce pixel
 - Si z est supérieur à la valeur courant de z max
 - Changer z maximal
 - Afficher le pixel à l'écran, de la couleur du polygone

Z-buffer (1)

-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞

Z-buffer (2)

-∞	1	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	1	1	1	-∞	-∞	-∞	-∞	-∞	-∞
-∞	2	2	2	2	-∞	-∞	-∞	-∞	-∞
-∞	2	2	2	2	2	-∞	-∞	-∞	-∞
-∞	3	3	3	3	3	-∞	-∞	-∞	-∞
-∞	3	3	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞

Z-buffer (3)

-∞	1	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	1	1	2	2	2	2	2	2	4
-∞	2	2	2	2	2	2	2	2	2
-∞	2	2	2	2	2	2	2	2	2
-∞	3	3	3	3	3	2	2	2	2
-∞	3	3	∞	-∞	-∞	-∞	-∞	-∞	-∞
-∞	-∞	-∞	∞	-∞	-∞	-∞	-∞	-∞	-∞




Z-Buffer : pour ou contre

✓ Pour :

- Facile à implémenter
- Travaille dans l'espace image
 - Rapide

✓ Contre :

- Coût en mémoire
- Travaille dans l'espace image
 - aliasing
 - artefacts



Z-Buffer

✓ Combien d'information ?

- Combien de bits pour z max ?
- Limité par la mémoire :
 - 8 bits, 1024x1280: 1.25 Mb
 - 16 bits, 1024x1280: 2.5 Mb
- Nécessaire pour la séparation des objets proches :
 - 8 bits, distance minimale entre objets de 0.4 % (4mm pour 1m)
 - 16 bits, distance minimale de 0.001 % (1mm pour 1km)
 - Que se passe t-il en dessous de cette limite ?

