Le Z-Buffer

Introduction

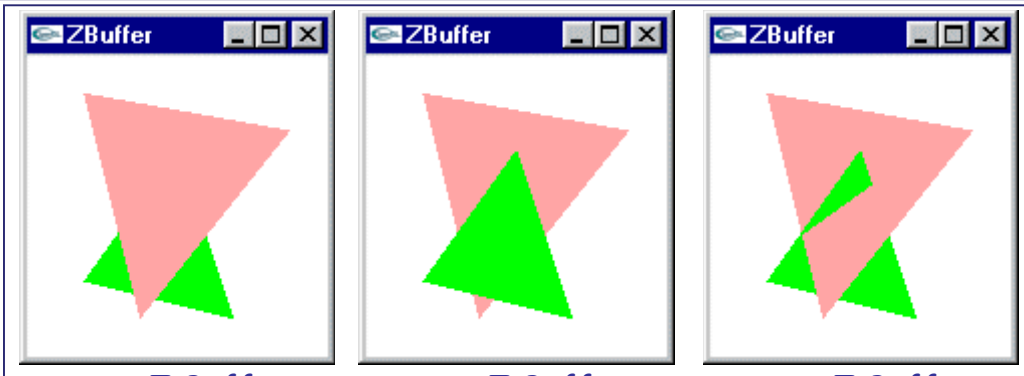
L'algorithme du Z-Buffer est un algorithme d'élimination des parties cachées.
-> Lors de la visualisation d'une scène, cet algorithme calcule l'image de manière que l'affichage de tout objet ou morceau d'objet masqué par un autre objet ou par lui même soit supprimé.

Algorithme conçu au milieu des années 70.

Autres algorithmes d'élimination des parties cachées:

- Algorithme du peintre
- Algorithmes de la ligne de balayage (scanline)
- Algorithmes par subdivision de l'image (Warnock, ...)
- ...

RETOUR



Sans Z-Buffer facette 1 puis 2 | Sans Z-Buffer facette 2 puis 1 | Avec Z-Buffer ordre indifférent

Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Principe du Z-Buffer

Le calcul de l'image est effectué séquentiellement en traitant les objets extraits de la scène (classiquement des facettes triangulaires) les uns à la suite des autres pour en extraire l'intégralité des pixels qui les représentent.
Au cours du processus de traitement d'un objet, la "profondeur" écran (de -∞, le plus profond possible, à +∞, le

moins profond possible) de chaque pixel calculé est comparée à celle déjà calculée en cette position image.
En fonction du résultat de cette comparaison, la couleur du pixel est:
- soit inchangée si la profondeur calculée établit que l'objet en cours de traitement est derrière,
- soit affectée avec la couleur de l'objet en cours de traitement, s'il est devant.

On gère ainsi, pour chaque pixel de l'écran, une recherche de maximum dans le but d'afficher finalement l'objet le moins profond présent en cette position (vis à vis de l'observateur).
-> Il n'y a pas de tri des objets et donc pas de problème de non-linéarité intrinséque du temps d'exécution.

Ne pas réaliser la séquentialisation principale sur les pixels mais sur les objets a beaucoup d'intérêts:

- optimisation des calculs,
- algorithme en temps linéaire du nombre de facettes,
- possibilité de commencer le calcul de l'image sans connaître l'intégralité de la scène,
- ...

Implantation

On définit deux zones mémoire de résolutions identiques à la résolution de l'image que l'on souhaite calculer:

- Un tampon couleur destiné à contenir la couleur de chacun des pixels de l'image (initialisé à la couleur de fond souhaitée)

- Un tampon profondeur destiné à contenir la profondeur écran maximum de chacun des pixels de l'image (initialisée à $-\infty$, utilisée comme valeur courante de comparaison en cours d'exécution du Z-Buffer)

La scène a été pré-traitée et est donc disponible en coordonnées écran.

On décompose (rasterise) chaque objet o à afficher en l'ensemble des pixels de coordonnées (x,y) qui le composeraient s'il était intégralement affiché seul.

Au cours de la décomposition, on calcule la profondeur écran z de chaque pixel de o de coordonnées image (x,y).

Si cette coordonnée z est plus grande que celle déjà présente dans le tampon profondeur aux coordonnées (x,y), la nouvelle profondeur en (x,y) devient z et la couleur du pixel (x,y) du tampon couleur est réaffectée avec une valeur calculée en fonction de x, de y, des caractéristiques de l'objet o, de sources lumineuses, ...

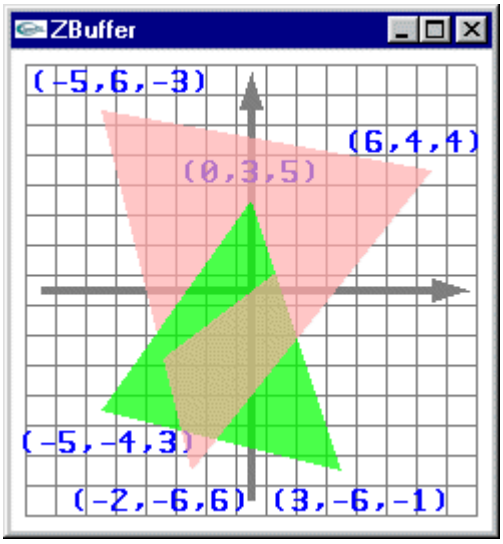
Quand tous les objets ont été traités, l'image est disponible dans le tampon couleur pour être affichée à l'écran.

Algorithme

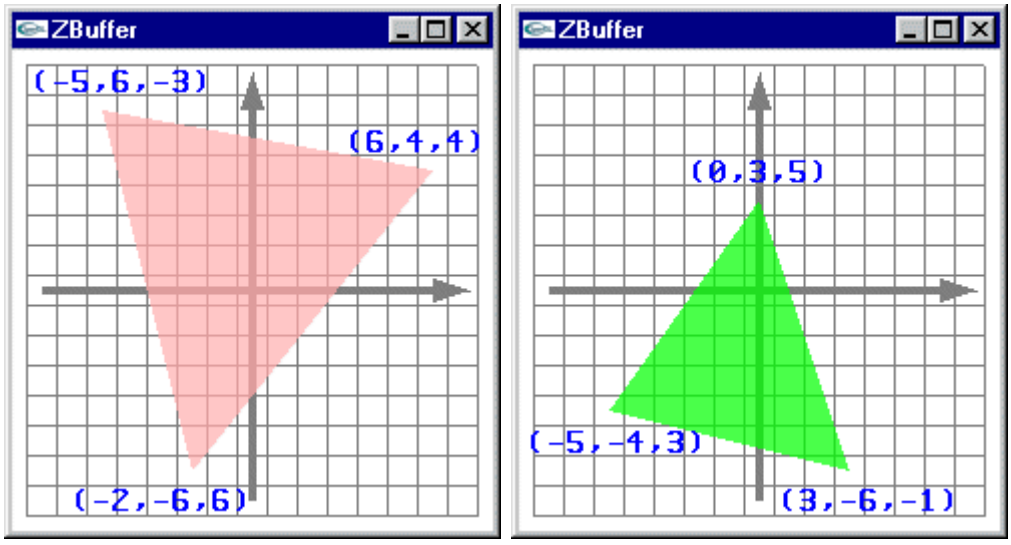
```
Initialiser le tableau PROF à -∞
Initialiser le tableau COUL
             à la couleur de fond
Pour chaque objet o de la scène à représenter
  Pour chaque pixel p=(x,y) de o
    Calculer la profondeur z de p=(x,y) ;
    Si z > PROF(x,y) alors
      PROF(x,y) = z ;
      COUL(x,y) = couleur(o,x,y) ;
    Finsi
  Finpour
Finpour
```

Les objets sont très fréquemment des facettes planes triangulaires car ce sont des surfaces simples à rastériser au moyen de variantes de l'algorithme de Bresenham.
Pour ce type d'objet, le calcul de la profondeur écran nécessaire à la rastérisation se marie bien avec l'opération de remplissage et entraîne donc un surcoût de calcul peu important.

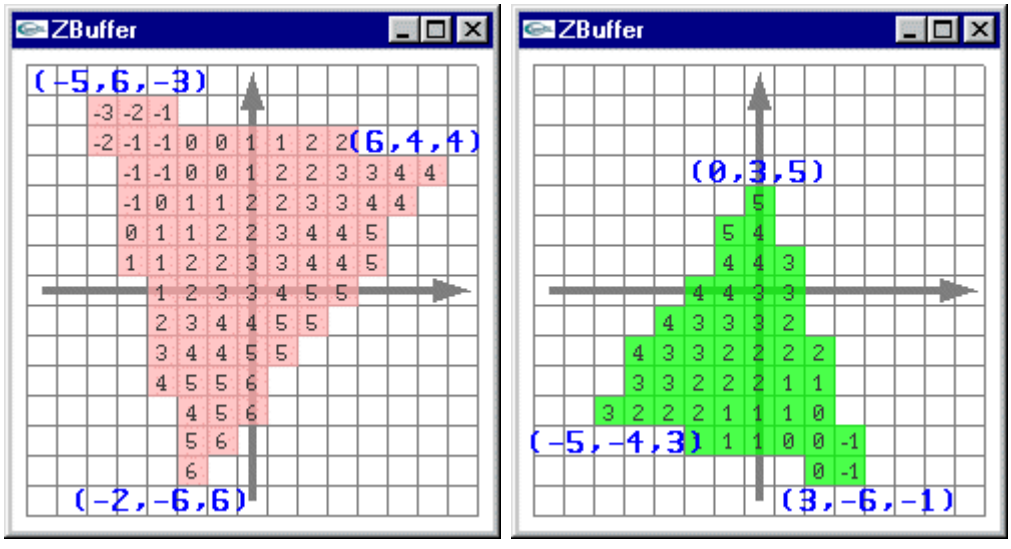
Exemple



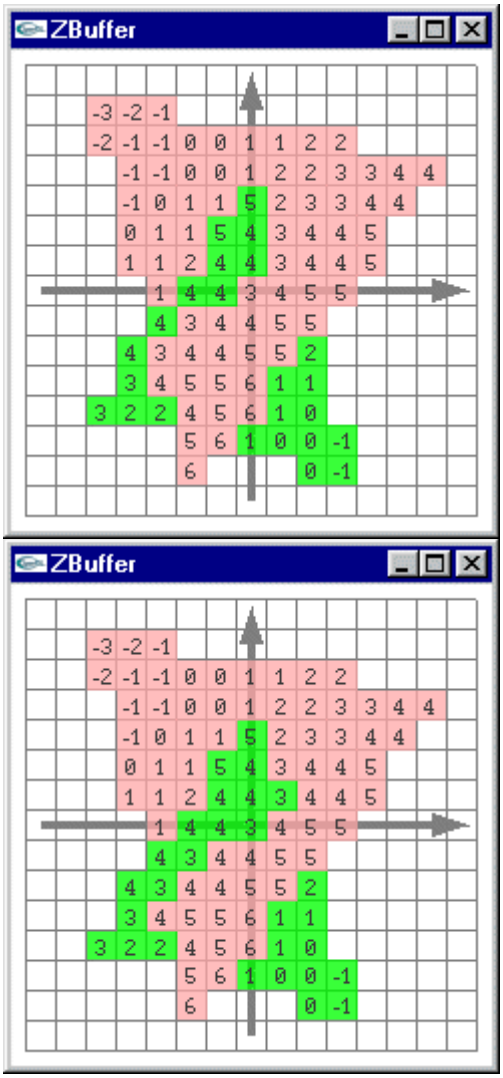
2 facettes à afficher
au moyen d'un Z-Buffer



Rasterisation de chacune de ces facettes



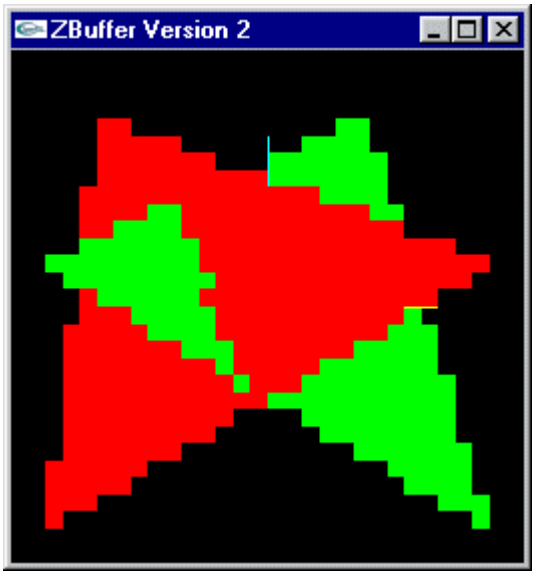
Calcul d'altitude pour chacun des pixels
de chacune des deux facettes



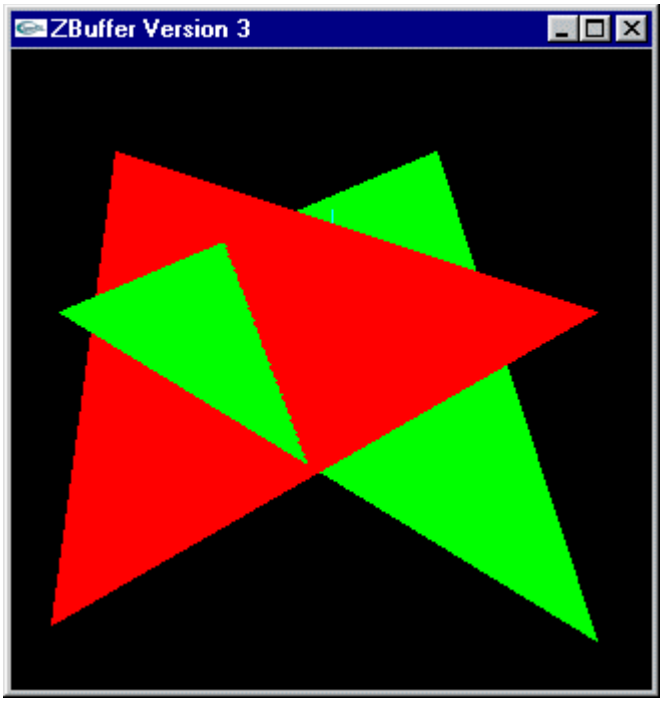
Affichage des pixels visibles
par comparaison des altitudes
(deux affichages possibles suivant
l'ordre de parcours des facettes)

Le programme GLUt

Exécutable GLUt



Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt



Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Implantation pratique

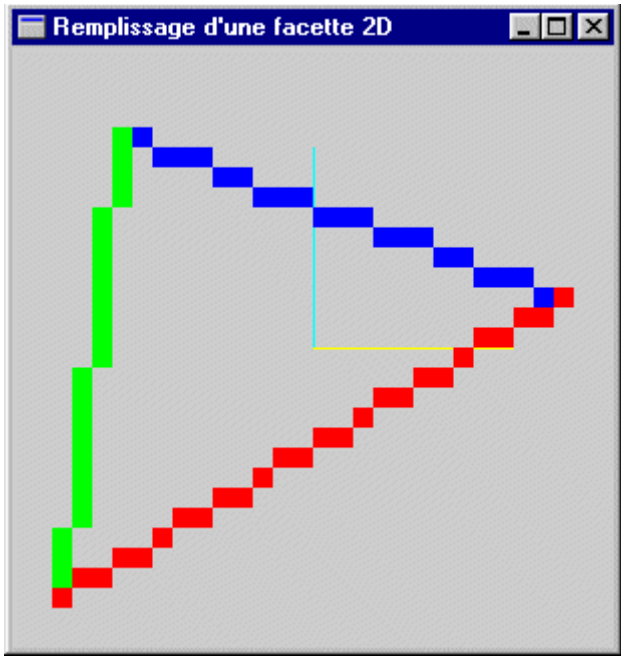
On dispose de facettes triangulaires dont les sommets sont connus en coordonnées écran.

Problèmes:

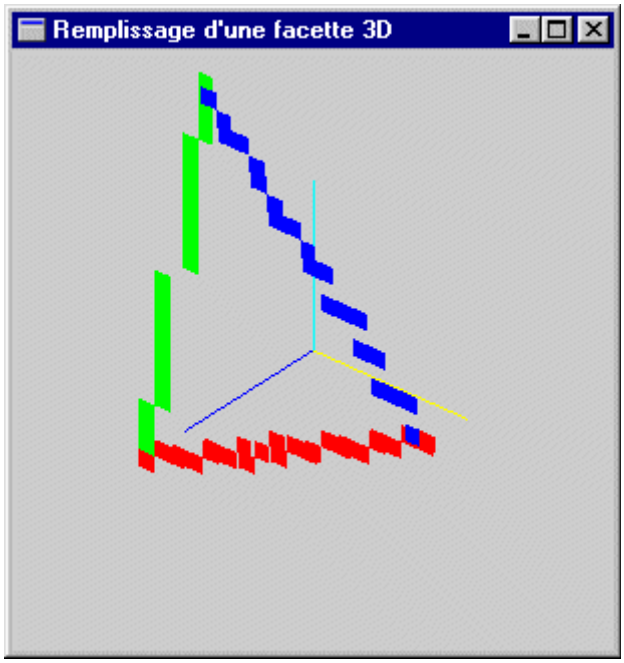
- Établir quels sont les pixels recouvrant chacune des facettes : Remplissage 2D.
- Calculer l'altitude de chacun de ces pixels.

Solution au premier problème: Trouver les pixels sur les cotés droit et gauche de la facette traitée. Tracer la trame horizontale entre le couple de pixels de chaque ligne de pixels sur laquelle la facette apparaît.

Solution au second problème: Travailler le remplissage 2D en gérant z en paramètre supplémentaire (pour optimiser les calculs, mariage de la gestion de z avec l'algorithme de remplissage 2D).

Problème 1 : Remplissage d'une facette 2D	
	
Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Problème 2 : Remplissage d'une facette 3D



Le programme Aux	Le programme GLUt
Exécutable Aux	Exécutable GLUt

Performances

Différentes unités de mesure de performance avec élimination des parties cachées:

- Taux de génération de pixels (pixels/seconde)
- Segments/seconde
- Polygones/seconde
- ...

Sur PC, de quelques dizaines de millions à plusieurs centaines de milliards de pixels par seconde, de quelques dizaines de milliers à plusieurs dizaines de millions de polygones par seconde (de quelques dizaines d'€ à plusieurs milliers d'€).

En constante évolution.

Conclusion

Intérêt

- Algorithme facilement implantable.

- Algorithme pouvant être optimisé (utilisation exclusive d'entiers, utilisation de variantes de l'algorithme de Bresenham pour le tracé de segments).
- Algorithme facilement implantable hard.
- Algorithme facilement pipelinable et parallélisable.
- Compatible de manière simple avec le calcul d'illumination de Gouraud et le plaçage de texture.
- Gestion implicite des recoupements entre objets.
- Exécution en un temps en o(n) du nombre de facettes -> bonne scalabilité.

Inconvénients

- Les deux zones tampon peuvent avoir des tailles très importantes (plusieurs Mo dans le cas de grands écrans). Actuellement, ce n'est plus véritablement un problème.
- La décomposition de chaque surface élémentaire en pixels nécessite une puissance de calcul importante (de l'ordre de 50 Mips pour 100000 facettes/s) et une vitesse d'accès mémoire très élevée (mémoire spécifique -> plus chère).
  Des techniques incrémentales (Bresenham) permettent toutefois d'optimiser les performances par l'utilisation de variables entières et d'opérations simples (additions et comparaisons).
- Pas de gestion intrinsèque simple des phénomènes de réflexion spéculaire et de transmission.
  Pas de gestion implicite des ombres portées.
  -> Difficultés pour implanter la modélisation de phénomènes optiques complexes pour des rendus plus réalistes.

-> Algorithme standard pour l'obtention en temps réel d'images de qualité moyenne.

-> Application à la conception et fabrication assistée par ordinateur (station de travail graphique), à la réalité virtuelle, à la réalité augmentée et au jeu.

-> Pas d'application à la génération d'images photo-réalistes car méthode de rendu trop limitée fonctionnellement.