# IMN638 - Interactions visuelles numériques

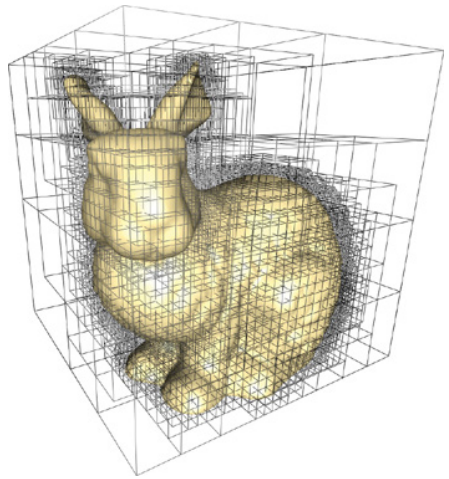
## Chapitre 2.5 - Gestion de la visibilité

Université de sherbrooke

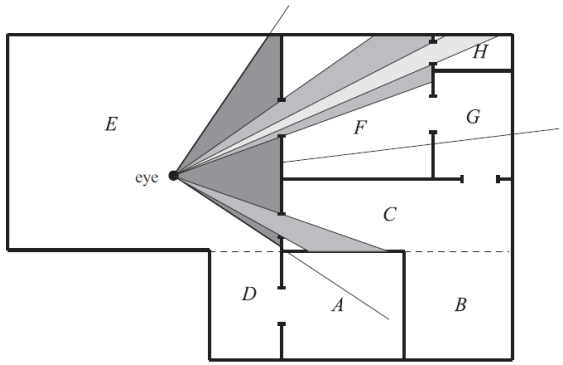
04 novembre 2013

# Sommaire

- Organisation spatiale des scènes
  - Hiérarchie de volumes englobants
  - Octree
- Optimisation du rendu par la visibilité
  - Face culling
  - Tronquage de vision
  - Groupes de visibilité
  - Portails
  - Optimisation selon la contribution
  - Optimisation selon l'occultation



Représentation d'un Octree (Sylvain Lefebre, 2005)




Visibilité dans une scène rendue par portails (RTR3, 669)

# Introduction

Un des objectifs principaux du rendu 3D en temps réel est d'atteindre des performances de rendu suffisamment élevées, pour une qualité d'image suffisamment élevée.

La matière contenue dans ce chapitre propose une série d'algorithmes permettant d'optimiser les performances du rendu. Le principe sous-jacent de tous ces algorithmes est le suivant : **Il ne faut pas envoyer au pipeline ou rendre inutilement ce qui n'est pas visible dans une scène.**

# Introduction

Bien que l'idée soit simple, sa mise en application reste complexe. En effet, pour la mettre en application, **il doit être possible de détecter plus rapidement ce qui n'est pas visible que d'en faire inutilement son rendu**.

Autrement dit, si un objet non-visible utilise un temps $X$ dans le pipeline de rendu avant d'être éliminé, le temps requis pour établir sa non-visibilité doit être inférieur à $X$ pour avoir un gain réel de performances.

Dans ce chapitre, nous étudierons des méthodes d'organisation de scène et des algorithmes de détermination de la visibilité permettant d'optimiser le rendu en fonction de la visibilité des objets contenus dans la scène.

# Organisation spatiale des scènes

# Volumes englobants

La notion de volume englobant est une notion capitale dans l'optimisation du rendu. De manière générale, elle permet de simplifier la représentation de l'espace géométrique occupé par un objet.

Comme son nom l'indique, un volume englobant est **une forme géométrique simple englobant totalement un objet géométriquement plus complexe**.

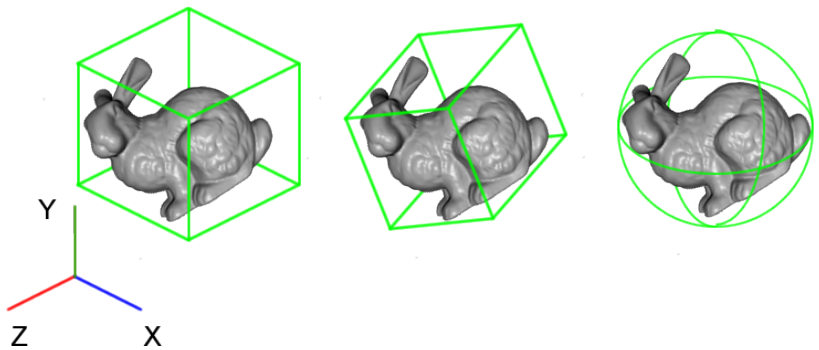
Les propriétés d'un volume englobant sont :

- La représentation du volume doit être plus simple que la représentation de la géométrique qu'il englobe.
- Le volume doit totalement contenir la géométrie qu'il englobe.
- La position du volume englobant dans la scène doit correspondre à celle de l'objet géométrique qu'il englobe.

# Volumes englobants

De manière générale, nous utilisons une des trois représentations suivantes pour les volumes englobants : (*Notons qu'il existe d'autres représentations mais qu'elles sont plus populaires pour la simulation physique et moins souvent employées en 3D en temps réel.*)

- Une sphère englobante. (Bounding sphere)
- Une boîte englobante alignée sur les axes. (Axis-aligned bounding-box ou AABB)
- Une boîte englobante non-alignée sur les axes.




**Gauche :** Boîte englobante alignée sur les axes, **Centre :** Boîte englobante non-alignée sur les axes, **Droite :** Sphère englobante

# Volumes englobants

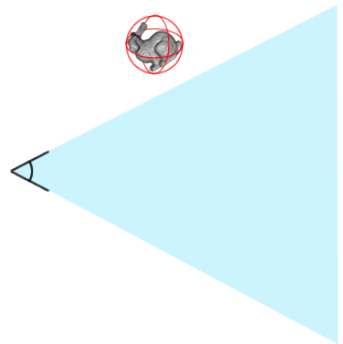
La construction d'un volume englobant est souvent un processus simple, impliquant une approximation très naïve de la forme géométrique. Par exemple :

- **Sphère englobante :** On prend le centre de notre objet géométrique, il devient le centre de notre sphère. La distance entre le vertex de la géométrie le plus loin du centre et le centre devient le rayon de la sphère.
- **Boîte englobante alignée avec les axes :** On prend le minimum/maximum en $X$, $Y$ et $Z$ des vertices de notre objet en espace monde, ce qui génère les limites de notre boîtes.

# Volumes englobants

Dans un contexte d'optimisation par la visibilité, on peut affirmer avec assurance que **si un volume englobant n'est pas visible, l'objet qu'il contient n'est pas visible non plus** et n'a donc pas besoin d'être rendu.

Ceci est justifié par le fait qu'un volume englobant occupe un espace plus grand ou égal à l'objet qu'il contient, il n'est donc pas possible de voir l'objet si le volume englobant lui-même n'est pas visible. En vérifiant si le volume englobant est visible, on peut donc vérifier si l'objet lui-même est visible, et l'éliminer ou le conserver dans la liste de rendu en conséquence.

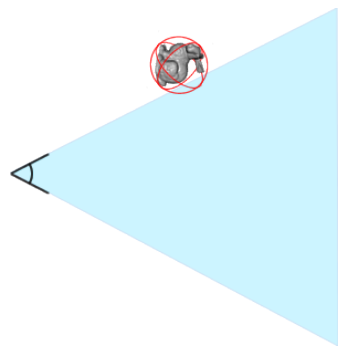


Le volume englobant n'est pas dans le champ de vision donc il n'est pas visible et l'objet qu'il contient ne l'est pas non plus.

# Volumes englobants

**L'avantage d'un volume englobant** est que vérifier sa visibilité est plus rapide (comme nous le verrons plus tard) que vérifier la visibilité de l'objet qu'il contient, puisque le volume englobant est plus simple. Il devient à ce moment plus rapide de vérifier si l'objet doit être affiché, via son volume englobant, que d'envoyer l'objet au processeur graphique et le laisser lui-même éliminer l'objet plus tard dans le pipeline, à l'étape du tronquage de vision.

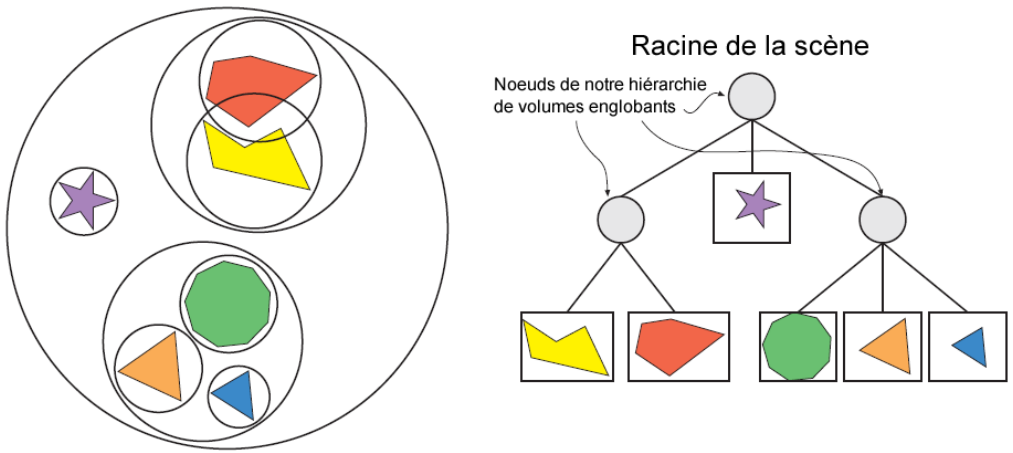
**Son désavantage principal** est qu'il ne fait qu'une approximation de la forme. Il est donc possible que le volume englobant soit partiellement visible alors que la forme géométrique qu'il englobe ne le soit pas. Dans un tel cas, la forme serait malgré tout soumise au pipeline, inutilement.



L'objet n'est pas visible mais le volume englobant l'est partiellement. L'objet est donc soumis malgré tout au pipeline.

# Hiérarchie de volumes englobants

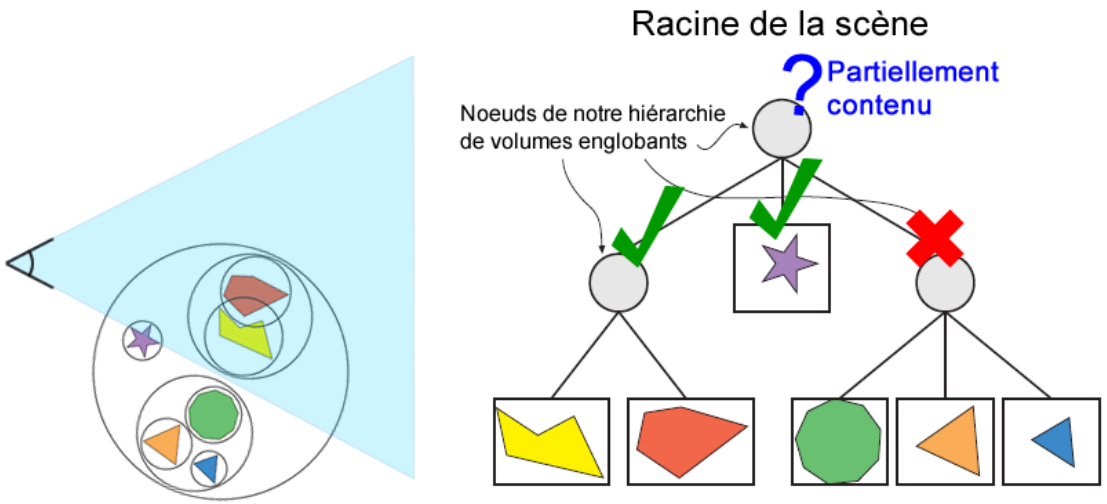
Continuant selon le même principe, il est possible d'organiser une scène en une hiérarchie de volumes englobants. Globalement, le tout revient à représenter une scène comme étant un arbre de volumes englobants, contenant des sous-volumes englobants. Les feuilles de l'arbre sont les formes géométriques contenues dans un seul volume englobant, la racine de l'arbre est un volume englobant contenant la totalité de la scène.




**Gauche :** Hiérarchie de volumes englobants à partir d'une scène. **Droite :** Représentation en arbre de cette même hiérarchie. (RTR3, p.648)

# Hiérarchie de volumes englobants

En vérifiant la visibilité des volumes englobants parents, il est possible d'éliminer plus rapidement des groupes de volumes englobants enfants. Ceci accélère encore plus la vérification de la visibilité puisqu'il n'est plus nécessaire de vérifier chaque objet individuellement pour effectuer le rendu.




Parcours de la hiérarchie de volumes englobant. On remarque qu'elle ne nécessite pas de vérifier chaque objet individuellement pour déterminer leur visibilité.

# Hiérarchie de volumes englobants

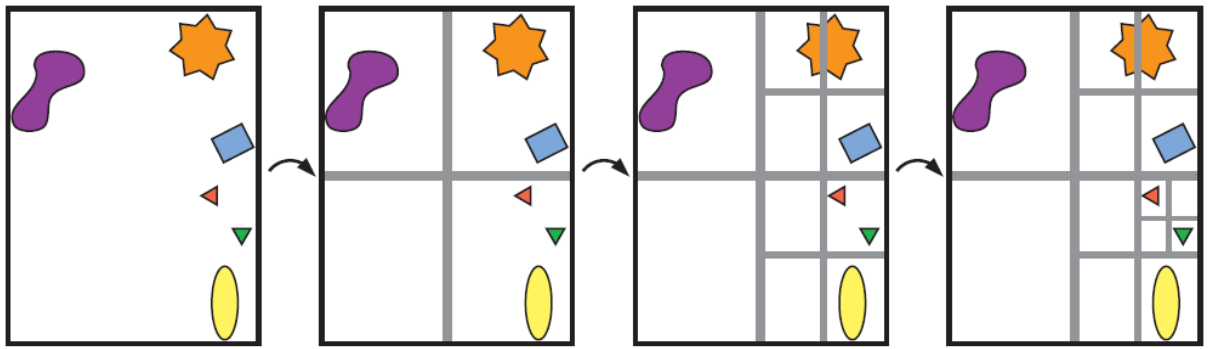
Globalement, la traversée d'une hiérarchie de volumes englobants pour la visibilité s'effectue selon l'algorithme suivant :

## Pseudo-code pour la traversée d'une hiérarchie de volumes englobants dans un contexte de détermination de la visibilité.

```
ParcourirVisibilité(NoeudCourant)
|
|   Si NoeudCourant est complètement visible
|   |   Ajouter toute la géométrie contenue dans NoeudCourant
|   |   et ses enfants à la liste de rendu.
|   |
|   Sinon Si NoeudCourant est partiellement visible
|   |   Si NoeudCourant possède des enfants
|   |   |   Pour chaque Enfant de NoeudCourant
|   |   |   |   ParcourirVisibilité(Enfant)
|   |   Sinon
|   |   |   Ajouter toute la géométrie contenu dans NoeudCourant
|   |   |   à la liste de rendu.
|   |
|
```

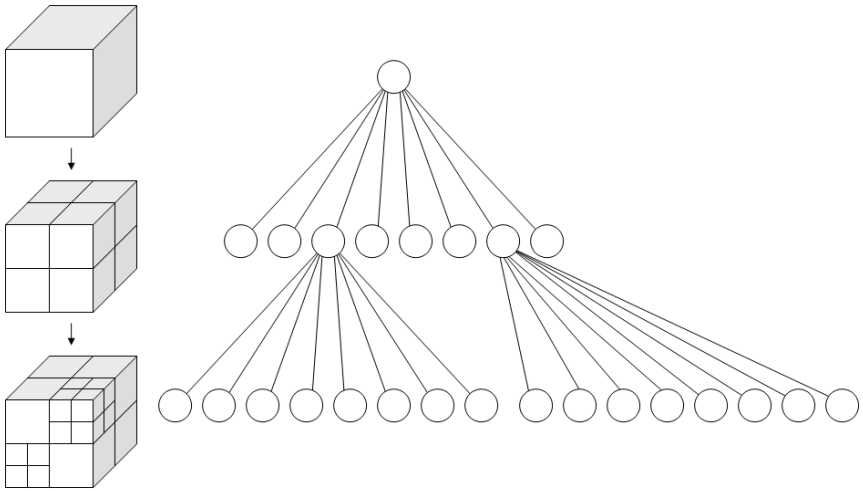
# Organisation d'une scène en octree

L'organisation de scène en octree est très similaire à l'organisation d'une scène par volumes englobants hiérarchiques. De manière générale, elle consiste à contenir une scène complète dans une boîte alignée avec les axes. La boîte est ensuite récursivement séparée en 8 sous-boîtes (1 séparation sur chaque axe) jusqu'à ce que le nombre d'éléments contenus dans une sous-boîte soit inférieur à un certain seuil. Notons que la séparation des boîtes en 8 boîtes plus petites s'effectue de manière indépendante. La séparation sera donc plus fine dans les sections de la scène contenant plus d'éléments.

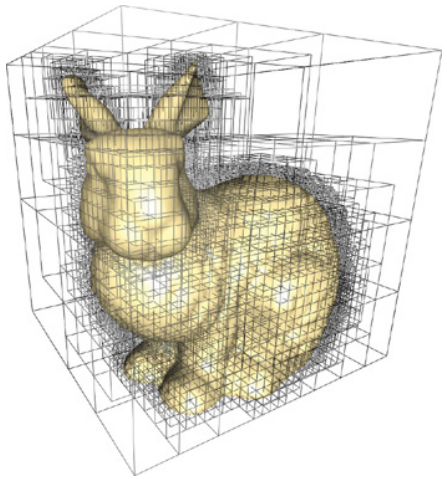


Exemple de construction d'un Quadtree, l'équivalent 2D d'un octree. (La construction est en tout point identique, à l'exception que l'axe des $Z$ n'existe pas. La séparation en sous-boîtes s'effectue donc seulement en $X$ et en $Y$, ce qui donne 4 sous-boîtes par niveau au lieu de 8.) (RTR3, p.655)

# Organisation d'une scène en octree



Niveaux d'un octree et arbre hiérarchique correspondant. (Wikipedia.org)



octree englobant un modèle 3D. Le seuil ici est en fonction du nombre de vertex par boîte. On remarque une division plus fine de l'octree où se trouve la géométrie, et une division plus grossière de cette dernière où il n'y en a pas. (Lefebvre, 2005)

# Organisation d'une scène en octree

L'avantage principal de l'octree est similaire à celui de la hiérarchie de volumes englobants, soit qu'il est possible de déterminer la visibilité d'une section de l'octree sans devoir se rendre au plus bas niveau de géométrie possible.

Les avantages de l'octree sont :

- Structure régulière où chaque noeud a 0 ou 8 enfants (représentation plus simple et compacte d'un point de vue programmatique).
- La structure est alignée avec les axes, il est donc très facile de vérifier si un point est contenu dans une boîte de la structure. (On vérifie simplement si le point est contenu dans les intervales formés par les limites de la boîtes sur chaque axe.)
- La structure étant alignée avec les axes, ceci accélère aussi la vérification de la visibilité. (Comme nous le verrons plus tard dans le chapitre.)

# Organisation d'une scène en octree

La construction d'un octree s'effectue selon le pseudo-code suivant :

## Construction d'un octree

```
ConstruireOctree(BoîteCourante,Seuil)
|
|   Si le nombre d'élements dans BoîteCourante est supérieur à Seuil
|   |   Ajouter 8 enfants à BoîteCourante, chaque enfant est un octant
|   |   de BoîteCourante
|   |
|   |   Pour chaque Enfant de BoîteCourante
|   |   |   ConstruireOctree(Enfant,Seuil)
|   |   |
|   Sinon
|   |   Ajouter les éléments contenus dans BoîteCourante à la liste
|   |   de géométrie de BoîteCourante.
|
```

# Organisation d'une scène en octree

Lors de la construction, si un élément est contenu dans deux boîtes séparées de l'octree, il doit être placé dans la liste du contenu de chacune des boîtes.

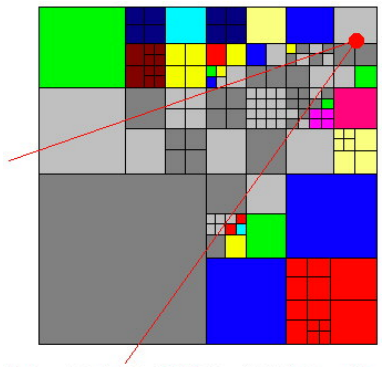
Il faut, dans un tel cas, s'assurer que l'élément ne sera pas ajouté deux fois à la liste de rendu si de multiples boîtes contenant un même objet sont affichées.

Seules les boîtes "feuilles" (soit les plus basses dans les niveaux d'octree) contiennent de la géométrie. Les boîtes parentes d'autres boîtes ne contiennent que la liste de leurs enfants.

# Organisation d'une scène en octree

Pour terminer, notons que le parcours de l'octree pour la visibilité s'effectue exactement de la même façon que pour la hiérarchie de volumes englobants *(voir acétate 12)*, puisque l'octree est un cas particulier de hiérarchie de volumes englobants.

La différence principale entre un octree et une hiérarchie de volumes englobants standard est l'évaluation de la visibilité d'un noeud de l'arbre. (Chaque division de l'octree étant une boîte alignée avec les axes, le test de visibilité se voit optimisé car l'évaluation de la visibilité est plus simple.)



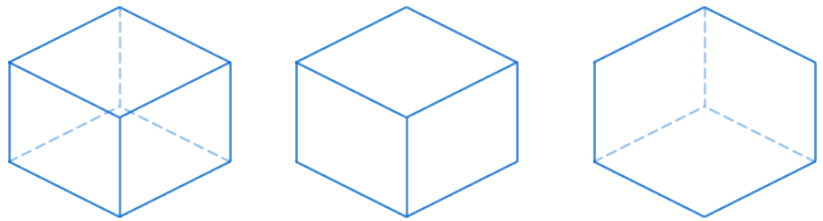
Traversée d'un Quadtree pour la visibilité. Les éléments en gris sont "visibles" car contenus dans le champ de vision. (Jaap Suter, 1999)

# Algorithmes d'optimisation par la visibilité.

# Face culling

Les algorithmes de face culling s'appliquent **dans le cas de volumes 3D fermés**. Par exemple, si vous regardez une sphère en 3D, à n'importe quel moment de la visualisation et sous n'importe quel angle, vous n'apercevrez qu'environ 50% de la sphère. L'arrière de cette dernière sera caché par l'avant. En va de même pour tous les volumes 3D fermés : une partie du volume est toujours située derrière une autre partie, et ne sera jamais vue par l'utilisateur.

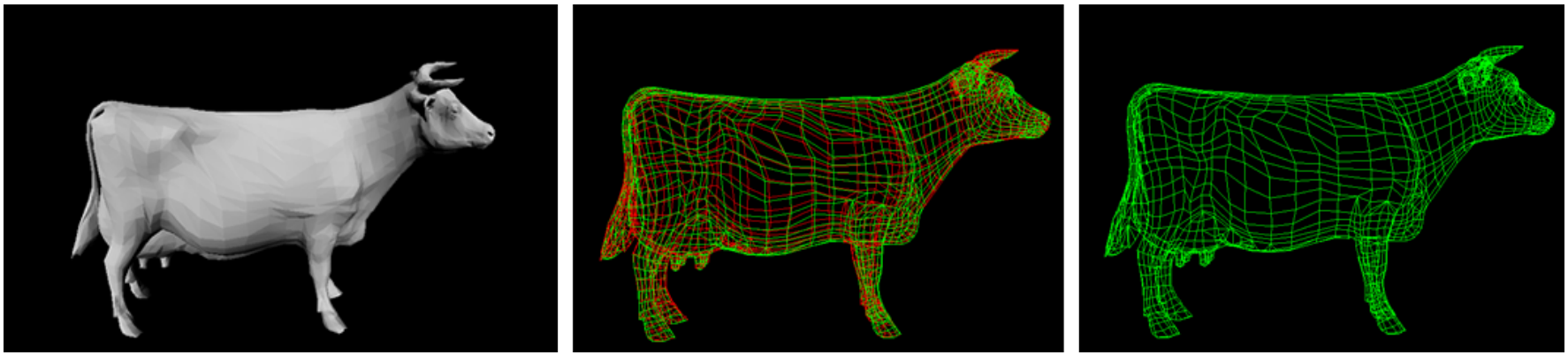
La partie du volume faisant face à la caméra est constituée des faces avant (front faces) alors que la partie du volume comprenant les faces faisant dos à la caméra est constituée des faces arrière (back faces).



Cube observé d'un point de vue où le lecteur est la caméra. **Gauche :** Géométrie complète du cube. **Centre :** Faces avant du cube (faces vers la caméra). **Droite :** Faces arrières du cube (Faces faisant dos à la caméra).

# Face culling

Dans de tels cas, il est inutile d'envoyer la totalité de la géométrie à travers le pipeline graphique, puisque la partie de la géométrie faisant face à la caméra cachera la géométrie lui faisant dos. On peut optimiser le tout en envoyant uniquement la géométrie faisant face à la caméra. Dans un tel cas, on effectue du "back-face culling". (Soit, on élimine les faces dos à la caméra.)

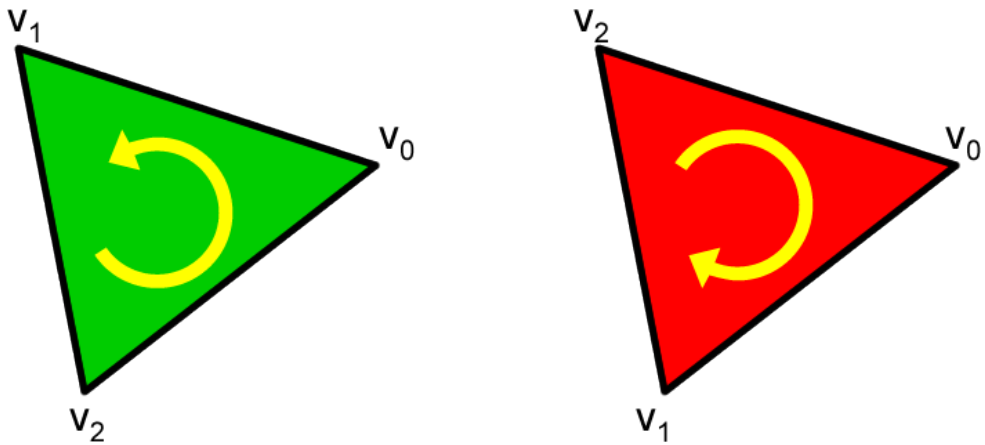


Modèle 3D observé d'un point de vue où le lecteur est la caméra. **Gauche :** Géométrie du modèle tel qu'aperçu par la caméra. **Centre :** Géométrie complète du modèle. Les segments rouges représentent la géométrie faisant dos à la caméra et les segments verts sont ceux faisant face à cette dernière. **Droite :** Géométrie optimisée de la vache. Toutes la géométrie des faces faisant dos à la caméra a été enlevée, ce qui réduit de beaucoup la complexité du modèle 3D et accélère le rendu.

# Face culling

La difficulté dans le face culling revient à déterminer si un polygone fait face ou non à la caméra. Pour déterminer si un polygone est face à la caméra ou non, on fixe l'heuristique suivante :

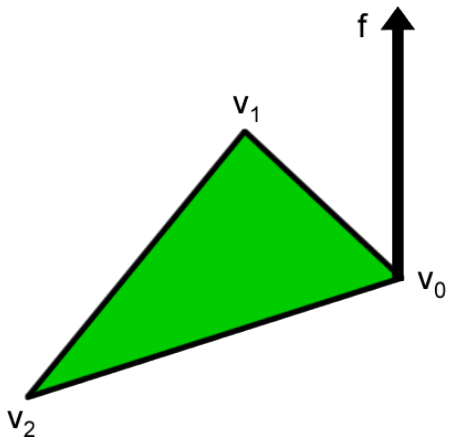
*Si les vertices des triangles sont définis dans le sens anti-horaire par rapport à la caméra, le triangle fait face à cette dernière. Si les vertices sont définis dans le sens horaire, le triangle lui fait dos.*




**Gauche :** Vertices du triangles définis dans le sens anti-horaire. **Droite :** Vertices du triangle définis dans le sens horaire.

# Face culling

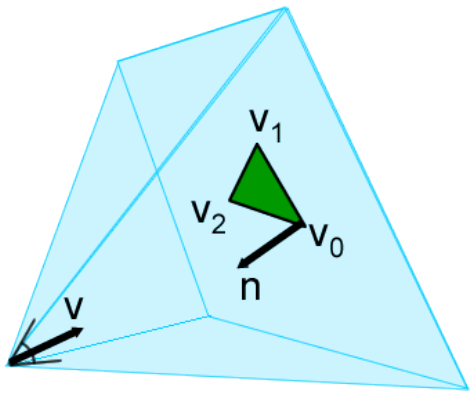
Les vertices formant le triangle dans un certain ordre, il est possible de produire un vecteur $f$ à partir de ceux-ci. Le vecteur en question se calcule tel que $f = (v_1 - v_0) \times (v_2 - v_0)$ et sera utilisé pour déterminer si les triangle est dans le sens horaire ou anti-horaire.(Et donc s'il fait dos ou face à la caméra.)



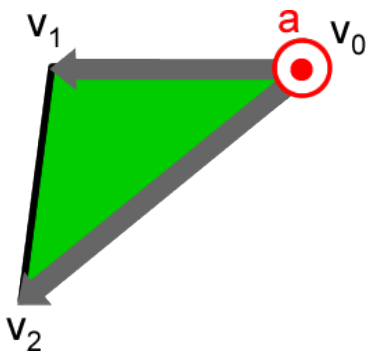
# Face culling

Il existe deux méthodes principalement utilisées pour déterminer si les vertices sont définis dans le sens anti-horaire ou dans le sens horaire.

**Méthode 1 :** En espace caméra, on effectue un produit scalaire entre le vecteur $f = (v_1 - v_0) \times (v_2 - v_0)$ et le négatif du vecteur de vision $(-v)$. Si le produit scalaire donne un nombre positif, les vertices sont dans le sens anti-horaire et le triangle est face à la caméra. Sinon, les vertices sont dans le sens horaire et le triangle fait dos à la caméra.




**Méthode 2 :** On transforme les vertices jusque dans l'espace image ($v_k$ devient $v'_k$). On effectue $f_{image} = (v'_1 - v'_0) \times (v'_2 - v'_0)$. On obtient un $f_{image}$ sous la forme $(0, 0, a)$. Si $a$ est positif, les vertices sont dans le sens anti-horaire et donc face à la caméra. Sinon, c'est qu'ils sont dans le sens horaire et donc dos à la caméra.

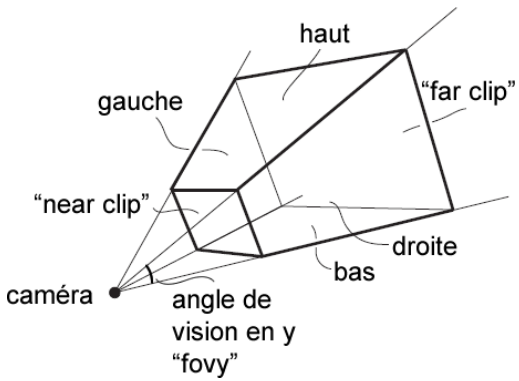



Pour un modèle fermé, si tous les $f$ des triangles pointent vers l'extérieur du modèle, ce dernier aura le comportement souhaité en regard au front/back face culling.

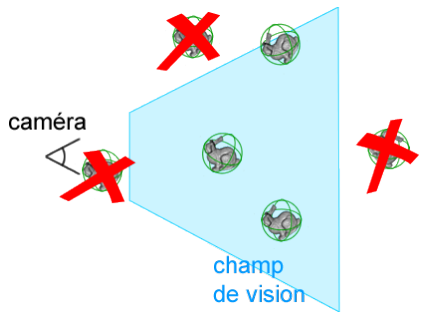
# Tronquage de vision

Une autre méthode d'optimisation du rendu par la visibilité est appellée le tronquage de vision(*frustum culling* en anglais). Globalement, le tronquage de vision consiste à afficher uniquement les objets se trouvant dans le champ de vision de l'utilisateur.

Le champ de vision dans un système graphique se modélise comme une pyramide tronquée, d'où le terme *frustum* culling. Cette optimisation consiste à **éliminer tous les objets n'étant pas situés à l'intérieur de cette pyramide tronqué**.




Champ de vision, ou "View Frustum" (RTR p.771)




Représentation schématique du tronquage de vision. Les objets avec un X rouge ne sont pas affichés car ils sont à l'extérieur du champ (frustum) de vision.

# Tronquage de vision

Tel que vu au chapitre 2.1, une étape fonctionnelle du pipeline graphique est appellée le **Tronquage de vision**. L'algorithme que nous verrons ici est similaire au tronquage de vision du processeur graphique.

La différence principale ici est que nous effectuons le tronquage sur les volumes englobants de notre géométrie. L'opération est relativement légère et évite de passer la géométrie inutilement à notre pipeline graphique.

En effet, bien que celle-ci serait correctement tronquée à l'étape de tronquage de vision, nous voulons éviter de faire passer inutilement notre géométrie dans le début du pipeline graphique si ce n'est pas nécessaire. De plus, le tronquage de vision du pipeline graphique est effectué pour chaque triangle, ce qui le rend beaucoup plus lourd. Pré-tronquer la géométrie avant le rendu enlève une charge importante de calcul associée traitement des vertices et au tronquage de vision au niveau processeur graphique.

# Tronquage de vision

Pour déterminer si un point est situé à l'intérieur ou à l'extérieur du champ de vision, on doit d'abord modéliser le frustum de vision.

On commence tout d'abord par multiplier notre matrice de projection $P$ et notre matrice modèle-vue $M$ afin d'obtenir une matrice $A$. Donc $A = PM$. Si on prend par la suite un vertex $v$ qu'on transforme en multipliant par $A$, on obtiendrait :

$$Av = v' \Rightarrow \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} xa_{11} + ya_{12} + za_{13} + wa_{14} \\ xa_{21} + ya_{22} + za_{23} + wa_{24} \\ xa_{31} + ya_{32} + za_{33} + wa_{34} \\ xa_{41} + ya_{42} + za_{43} + wa_{44} \end{pmatrix} = \begin{pmatrix} v \cdot ligne_1 \\ v \cdot ligne_2 \\ v \cdot ligne_3 \\ v \cdot ligne_4 \end{pmatrix} \quad (1)$$

où $v'$ serait un vecteur homogène.

# Tronquage de vision

Comme nous l'avons vu au chapitre 2.1, $v'$ est dans un espace normalisé dans la mesure où chaque composante de $v' = (x', y', z', w')$ est située entre $-w'$ et $w'$ si ils sont visibles dans notre image. En d'autres mots, chaque composante $(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'})$ est située entre $-1$ et 1, soit les frontières de notre espace de coordonnées image. (Les éléments situés à l'extérieur de ces limites sont situés à l'extérieur de l'image.)

On peut donc dire que les points $(x', y', z')$ sont situés dans le champ de vision s'ils répondent aux contraintes suivantes :

$$-w' < x' < w' \tag{2}$$
$$-w' < y' < w' \tag{3}$$
$$-w' < z' < w' \tag{4}$$

# Tronquage de vision

Un point est donc contenu dans le frustum de vision s'il répond aux 6 conditions suivantes :

$$-w' < x' \quad (5) \qquad w' > y' \quad (8)$$
$$w' > x' \quad (6) \qquad -w' < z' \quad (9)$$
$$-w' < y' \quad (7) \qquad w' > z' \quad (10)$$

Si on transforme l'inéquation **(5)** en utilisant l'équation **(1)**, on obtient :

$$-w' < x' \quad (11)$$
$$\Rightarrow \quad -(v \cdot ligne_4) < (v \cdot ligne_1) \quad (12)$$
$$\Rightarrow \quad 0 < (v \cdot ligne_4) + (v \cdot ligne_1) \quad (13)$$
$$\Rightarrow \quad 0 < v \cdot (ligne_4 + ligne_1) \quad (14)$$

# Tronquage de vision

En remplaçant les termes dans **(5)** à **(10)**, comme il a été fait en **(11)** à **(14)** on obtient les conditions suivantes pour qu'un point $v$ soit à l'intérieur du champ de vision :

$$
\begin{aligned}
0 &< v \cdot (ligne_4 + ligne_1) \\
0 &< v \cdot (ligne_4 - ligne_1) \\
0 &< v \cdot (ligne_4 + ligne_2)
\end{aligned}
\qquad
\begin{aligned}
0 &< v \cdot (ligne_4 - ligne_2) \\
0 &< v \cdot (ligne_4 + ligne_3) \\
0 &< v \cdot (ligne_4 - ligne_3)
\end{aligned}
$$

Nous pouvons donc caractériser les 6 plans de notre frustum de vision à l'aide des 6 soustractions/additions de ligne effectuées ci-haut. Lorsque nous voudrons vérifier si un point est à l'intérieur ou à l'extérieur du frustum, nous vérifions les 6 contraintes.

# Tronquage de vision

Les 6 plans de notre frustum de vision sont donc maintenant décrits par les 6 vecteurs :

$$P_1 = (ligne_4 + ligne_1)$$
$$P_2 = (ligne_4 - ligne_1)$$
$$P_3 = (ligne_4 + ligne_2)$$
$$P_4 = (ligne_4 - ligne_2)$$
$$P_5 = (ligne_4 + ligne_3)$$
$$P_6 = (ligne_4 - ligne_3)$$

Les équations de plan étant sous la forme $ax + by + cz + d = 0$, nos vecteurs $P_1$ à $P_6$ donnent les 4 coefficients $a, b, c, d$ de chaque plan.

Or, on sait que pour trouver la distance $dist$ entre un plan $P = (a, b, c, d)$ et un point $v = (x, y, z)$ , il suffit de faire $dist = ax + by + cz + d$ pour autant que le vecteur $(a, b, c, d)$ soit normalisé en fonction de la normale du plan, $(a, b, c)$.

# Tronquage de vision

On normalise donc les vecteurs $P_1$ à $P_6$ en fonction de leur normale $(a, b, c)$ respective. Nous obtenons alors 6 vecteurs décrivants nos plans normalisés $\pi_1$ à $\pi_6$ tel que :

$$\pi_k = \frac{P_k}{\sqrt{a_k^2 + b_k^2 + c_k^2}} \tag{15}$$

Nos tests visant à déterminer si un point $v$ est situé à l'intérieur du champ de vision sont donc, à la fin : (les 6 tests doivent être vrai pour que le point soit dans le champ de vision)

$$0 < v \cdot \pi_1 \tag{16}$$
$$0 < v \cdot \pi_2 \tag{17}$$
$$0 < v \cdot \pi_3 \tag{18}$$
$$0 < v \cdot \pi_4 \tag{19}$$
$$0 < v \cdot \pi_5 \tag{20}$$
$$0 < v \cdot \pi_6 \tag{21}$$

# Tronquage de vision

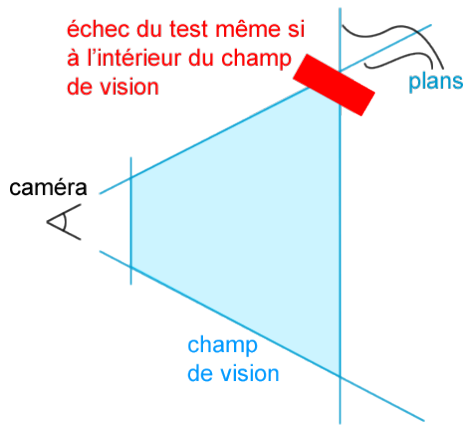
Vérifier si un volume englobant sous la forme de boîte (alignée ou non avec les axes) est dans le champ de visions revient donc à vérifier si ses sommets (nos points) sont à l'intérieur ou non du champ de vision, en vérifiant les inéquations **(16)** à **(21)** .

Dans le cas d'une sphère englobante, on utilise des tests similaires. Notre point $v$ correspond au centre de la sphère, mais on tient compte aussi du rayon de la sphère $r$. Nos tests deviennent alors :

$$\begin{aligned} 0 &< v \cdot \pi_1 + r \\ 0 &< v \cdot \pi_2 + r \\ 0 &< v \cdot \pi_3 + r \end{aligned} \qquad \begin{aligned} 0 &< v \cdot \pi_4 + r \\ 0 &< v \cdot \pi_5 + r \\ 0 &< v \cdot \pi_6 + r \end{aligned}$$

# Tronquage de vision

Puisque nous évaluons seulement les sommets du volume englobant, notons que le test peut échouer pour un volume englobant même si une partie de la géométrie est bel et bien située à l'intérieur du champ de vision.

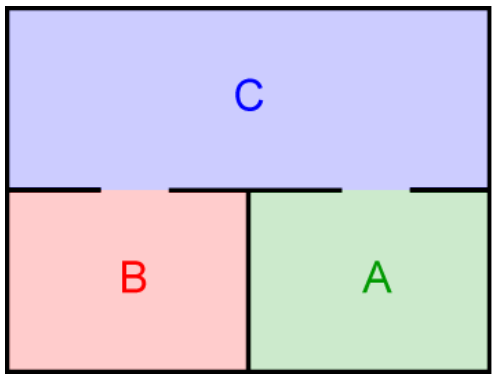



On peut décider d'ignorer de tels cas, puisqu'ils demeurent marginaux, ou d'évaluer des points intermédiaires sur les segments du volume englobant pour réduire les situations problématiques possibles.

# Groupes de visibilité

La méthode des groupes de visibilité (visibility groups ou vis-groups) est une méthode particulièrement simple mais aussi particulièrement efficace lorsque vient le temps d'optimiser la visibilité pour une scène quelconque.

Globalement, la méthode consiste à prendre une scène et la diviser en plusieurs cellules. Le concepteur de la scène doit manuellement diviser la scène en cellules et spécifier quelles cellules sont visibles lorsque la caméra se trouve dans une cellule en particulier.



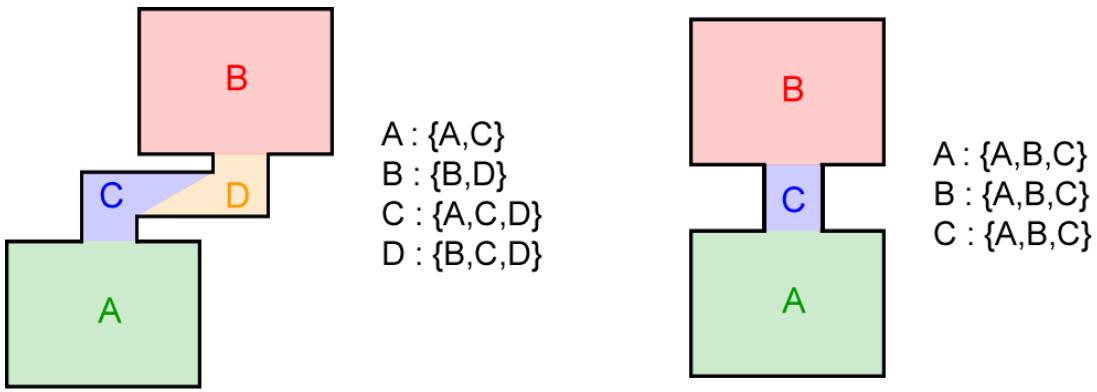

A : {A,C}
B : {B,C}
C : {A,B,C}

**Gauche :** Exemple de scène divisée en cellules (A,B,C). **Droite :** Spécification des cellules visibles lorsque la caméra se trouve dans une cellule particulière. Cette liste est spécifiée manuellement par le concepteur de la scène, au même titre que les cellules.

# Groupes de visibilité

Lors de la conception de la scène, il est important de garder en tête que l'optimisation est possible si la géométrie est organisée de manière à favoriser une division de scène en groupes de visibilité.

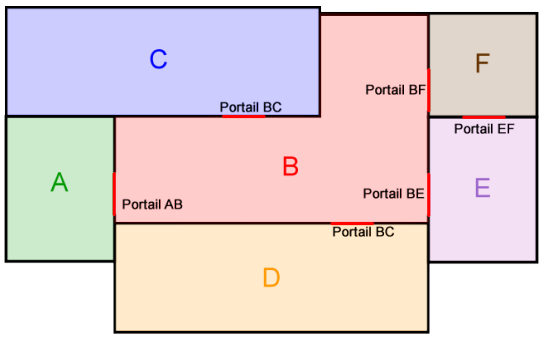
Dans l'exemple ci-bas, si les cellules $A$ et $B$ contiennent de la géométrie très complexes, la scène de gauche sera beaucoup plus optimale dans la mesure où elle est organisée de telle façon à ce que $A$ et $B$ ne soient jamais visibles en même temps. Dans la scène de droite, $A$ et $B$ sont potentiellement visibles à partir de n'importe quelle cellule de la scène, ce qui est peu optimal.



# Méthode des portails

La méthode des portails ressemble en plusieurs points à la méthode des groupes de visibilité. Sa différence principale est que la détermination de la visibilité des cellule à un endroit précis s'effectue automatiquement plutôt que d'être déterminée par le concepteur de la scène.

La méthode des portails consiste à diviser la scène en cellules comme c'était le cas avec les groupes de visibilité. Cependant, afin de donner des informations de connectivité entre les cellules, les points de jonction visibles entre celles-ci sont définis à l'aide de polygones que nous appellons des portails.




Scène intérieure vue de haut où chaque pièce est une cellule et chaque cadre de porte est un portail. Les portails servent de connection entre les différentes cellules de notre scène.

# Méthode des portails

L'objectif, avec la méthode des portails, est de rendre exclusivement les cellules qui sont visibles à partir de la cellule où nous nous trouvons et du champ de vision que nous avons. La méthode suit les étapes suivantes :

1. Trouver la cellule $V$ où la caméra (oeil) est située.
2. Initializer un rectangle englobant 2D $P$ de la taille de l'écran.
3. Rentre la géométrie de la cellule $V$ en utilisant le tronquage de vision pour la pyramide tronquée qui émane de la caméra et passe à travers le rectangle $P$ (initialement tout l'écran).
4. Récursivement, on parcour les portails qui connectent vers les cellules voisines de $V$. Pour chaque portail visible de la cellule courante, on projette le portail à l'écran et on trouve le rectangle englobant de la projection. On calcule l'intersection entre $P$ et le rectangle englobant.

# Méthode des portails

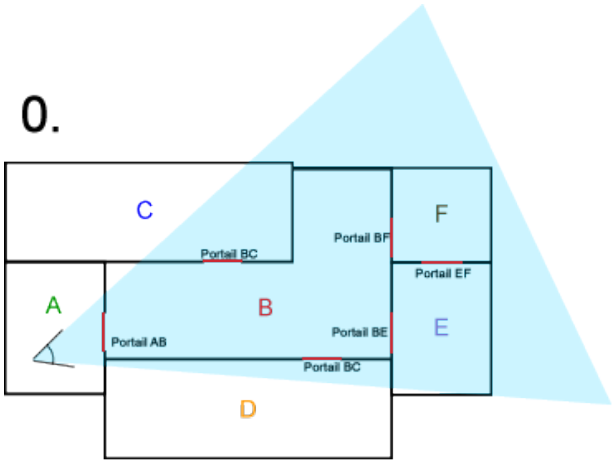
6. Pour chaque intersection : Si elle est vide, la cellule connectée via le portial est invisible du point de vu courant, et la cellule peut être ommise. Si l'intersection n'est pas vide, le contenu de cette cellule peut être rendu après le tronquage de la pyramide tronquée émanant de l'intersection.

7. Si l'intersection n'étais pas vide, alors les cellules voisines de ce voisin peuvent être visible. On effectue récursivement le point **(3)** avec $P$ maintenant égal à l'intersection.

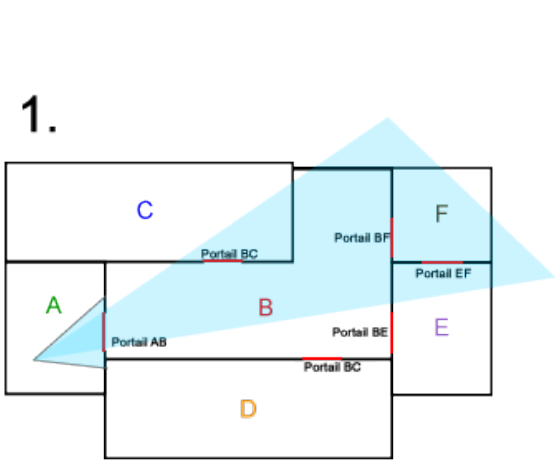
# Méthode des portails

Si on reprend notre scène exemple, on aurait schématiquement, étape par étape :

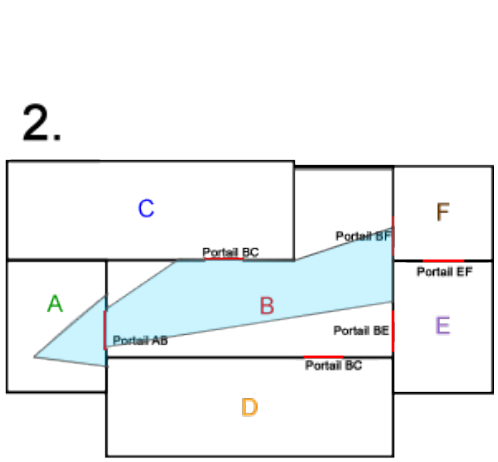
0.



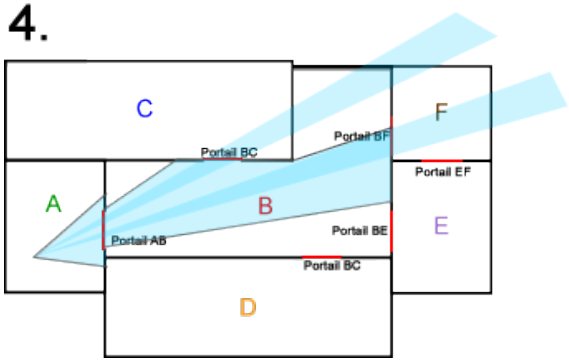

1.



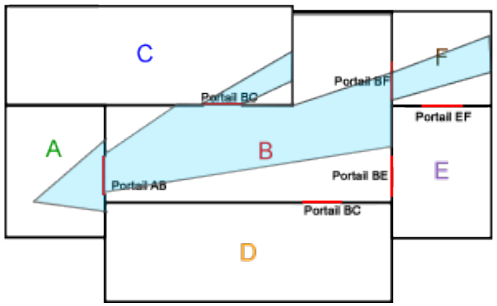

2.



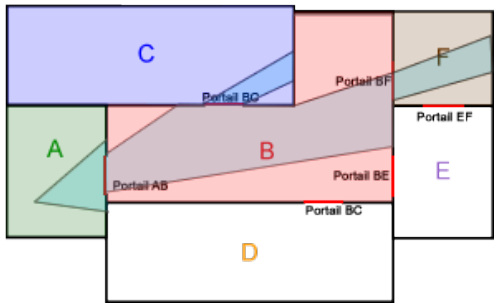

4.




5.

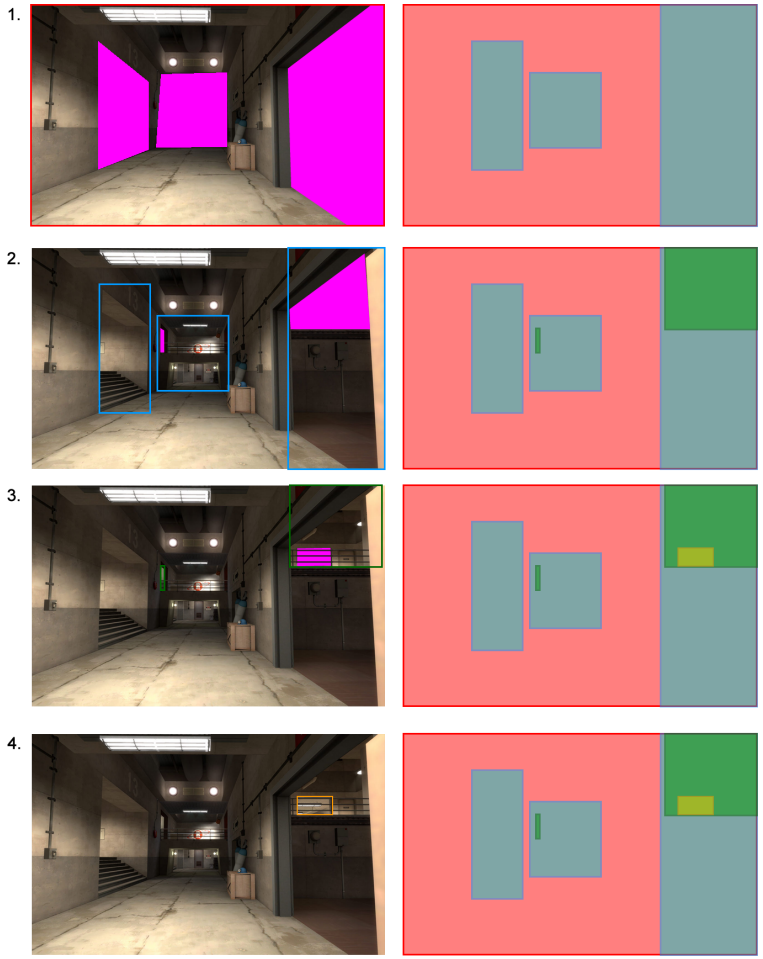



Fin



# Méthode des portails

Du point de vu de la caméra, pour une autre scène 3D, on aurait :



# Méthode des portails

Pour terminer sur la méthode des portails, notons que cette dernière est souvent étendue de manière à ce que seulement la géométrie se trouvant dans le champ de vision soit affichée.

Dans la mesure où chaque nouveau portail peut être considéré le point de départ d'un nouveau fustrum (pyramide tronquée représentant le champ de vision), il est possible de définir, pour chaque cellule, quelle partie de la cellule se situe dans le frustum et donc quelle partie doit être affichée. Il est en effet possible de faire un tronquage de vision, tel que vu précédemment, pour chaque nouveau frustum créé lors du parcours de la cellule.

# Optimisation selon la contribution

L'optimisation de rendu selon la contribution (contribution culling ou detail culling) est l'algorithme d'optimisation de la visibilité le plus simple que nous verrons dans ce chapitre.

À toute fin pratique, il suit le principe selon lequel un très petit élément dans une scène joue un rôle très peu important et qu'il peut être éliminé sans affecter de manière sensible l'apparence visuelle de la scène.

**L'optimisation selon la contribution effectue une approximation du nombre de pixels occupés par un modèle géométrique dans l'image rendue. Si le modèle occupe un nombre de pixel inférieur à un certain seuil, il n'est simplement pas affiché.**

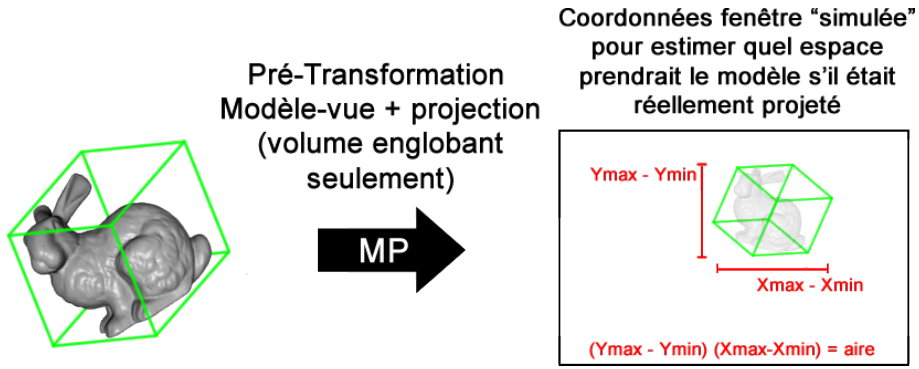
# Optimisation selon la contribution

Une méthode rapide pour estimer l'importance d'un modèle 3D dans l'image affichée est d'estimer l'aire occupée par sa projection dans l'image. Cette estimation s'effectue en projetant, dans l'image, le volume englobant du modèle.

À partir des points composants le volume englobant projeté, nous pouvons extraire le minimum et le maximum en $X$ et en $Y$ du volume englobant, puis multiplier les deux axes ainsi formés afin d'avoir une approximation de l'aire qui serait occupée par le modèle 3D dans l'écran.

Si l'aire est inférieure à un certain seuil, le modèle n'est simplement pas envoyé au rendu car il demande une quantité de calcul trop importante lors de son affichage pour le peu qu'il contribue à l'image.

# Optimisation selon la contribution



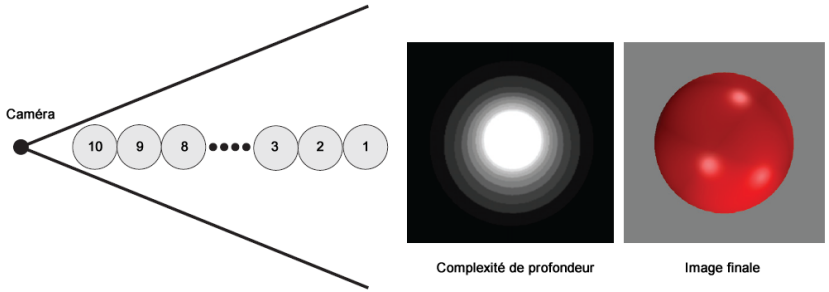

Représentation schématique des étapes effectuées pour estimer l’aire qui sera occupée par un objet une fois celui-ci projeté.

Exemple de cas où une optimisation selon la contribution peut être utile. Dans l’image de gauche, on observe plusieurs objets au niveau du relief lorsque la caméra est près de la surface. Dans l’image de droite, on observe le même relief vu de très haut. Les objets dont il était question dans l’image de gauche ont été enlevés car ils ne contribuaient plus suffisamment au rendu. (Johan Anderson, 2007)

# Optimisation selon l'occultation

L'optimisation selon l'occultation consiste à éviter d'afficher les éléments situés derrière des éléments déjà affichés.

En pratique, le Z-Buffer effectue déjà une telle opération dans la mesure où, si le test du Z-Buffer est correctement spécifié, il ne rendra que les éléments les plus près du point de vision, les éléments situés derrière ceux-ci seront simplement omis. Malheureusement, dans un cas où une scène serait rendue de l'arrière (l'élément le plus loin) vers l'avant (l'élément le plus près), rien n'empêche un pixel d'être réécrit plusieurs fois avant d'avoir sa couleur finale.




Les cercles sont affichés de l'arrière (cercle 1) vers l'avant (cercle 10). On remarque que la complexité de profondeur (le nombre de fois qu'un pixel est réécrit) est plutôt élevée, et que des pixels ont été réécrits plusieurs fois, même si l'image finale n'affiche qu'une seule sphère.

# Optimisation selon l'occultation

Notre premier réflexe serait donc de suggérer de rendre la même scène de l'avant vers l'arrière. Bien que ceci ait une influence positive sur le temps de rendu, on ne règle que partiellement le problème puisque la géométrie doit encore passer dans presque la totalité du pipeline avant d'être éliminée lors du test de profondeur.

Pour remédier à un tel problème, il faudrait être en mesure de vérifier si la géométrie sera occultée avant même d'être envoyée au pipeline.

La méthode employée pour effectuer une telle vérification en 3D en temps réel utilise une fonctionnalité récente des processeurs graphiques, soit **la requête d'occultation (occlusion query)**.

# Requêtes d'occultation

1. Une requête d'occultation débute en envoyant de la géométrie au processeur graphique à partir de l'application, via l'API graphique.
2. La géométrie est ensuite transformée par le processeur graphique, à travers le pipeline, jusqu'à l'état de fragments en coordonnées fenêtres.
3. Le processeur graphique évalue ensuite le nombre de fragments de cette géométrie qui passerait le test de profondeur en fonction des valeurs déjà contenues dans le tampon-z au moment de la requête (le tout sans écrire dans le tampon image ou le tampon-z).
4. Le nombre de fragments passant le test est ensuite retourné à l'application via l'API graphique.

Si la requête d'occultation retourne 0, c'est que la totalité de la géométrie envoyée était occultée. Si la requête d'occultation retourne un nombre supérieur à 0, c'est qu'une partie de la géométrie est visible.

# Requêtes d'occultation

Bien entendu, envoyer la géométrie complète d'un modèle dans la requête d'occultation serait presque aussi long que de simplement la rendre directement avec un test de profondeur, ce qui ferait des tests d'occultation peu efficaces.

L'astuce à ce niveau est d'envoyer une géométrie englobante simplifiée de notre modèle réel, par exemple un volume englobant tel que nous en avons vu en début de chapitre. Le nombre de polygones étant de beaucoup réduit, transformer le volume à travers le pipeline s'effectue beaucoup plus rapidement et permet d'effectuer un test d'occultation beaucoup plus rapide.

Si le test d'occultation pour le volume englobant retourne 0 et indique que le volume est complètement occulté, nous avons une garantie que le modèle complet sera lui aussi occulté et qu'il est donc inutile de l'envoyer afin qu'il soit rendu. Dans le cas inverse, le modèle peut être envoyé pour le rendu.

# Requêtes d'occultation

Pour qu'un rendu utilisant des requêtes d'occultation soit efficace, il est suggéré de suivre certains principes :

- Il est préférable de rendre la géométrie de l'avant vers l'arrière par rapport à la caméra, afin d'augmenter les chances que la géométrie de l'arrière soit occultée par la géométrie en avant et ne soit pas rendue inutilement.
- Il est peu utile d'effectuer des requêtes d'occultation au début du rendu d'une scène puisqu'il y a peu de chances que de la géométrie soit cachée s'il n'y a pas encore beaucoup d'éléments s'étant inscrit dans le tampon-z.
- Il est peu utile d'effectuer des requêtes d'occultation à la toute fin du rendu puisque le processeur (CPU) ne peut faire rien d'autre qu'attendre le résultat de la requête et perd son temps. (Autant tout de suite envoyer la géométrie et permettre au processeur de préparer une nouvelle image pendant que le GPU termine celle-ci).
- La règle d'or pour la géométrie transparente persiste : Toujours rendre la géométrie transparente à la fin.

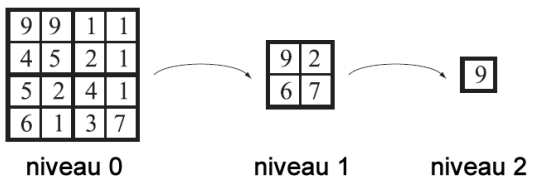
# Z-Buffer hiérarchique

Pour que les requêtes d'occultation puissent être si performantes au niveau des processeurs graphiques, une modification a été apportée à la façon dont est géré le tampon-z.

Pour accélérer les requêtes d'occultation, le tampon-z tel que retrouvé dans les processeurs graphiques modernes est en réalité un **tampon-z hiérarchique**, construit sous la forme d'une pyramide.

À son niveau le plus bas, le tampon-z possède exactement la taille de l'image rendue en pixels. Chaque niveau supérieur du tampon-z est une réduction correspondant à $\frac{1}{4}$ de la surface du niveau inférieur.

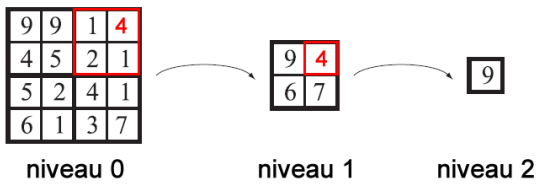
Chaque pixel d'un niveau du tampon-z hiérarchique contient la distance la plus élevée pour le groupe de $2 \times 2$ pixels correspondant dans le niveau inférieur du tampon-z.




Représentation d'un (petit) z-buffer hiérarchique (RTR3, p.678)

# Z-Buffer hiérarchique

Lorsqu'une valeur est écrite dans le z-buffer (niveau 0), la modification de valeur est propagée dans les pixels correspondants des niveaux supérieurs afin qu'ils soient mis à jour si cette dernière s'avère supérieure à la valeur déjà présente dans ceux-ci.




Mise à jour du z-buffer hiérarchique (RTR3, p.678)

# Z-Buffer hiérarchique

Lors d'une requête d'occultation, on conserve en mémoire la distance la plus petite de la géométrie soumise pour la requête.

On trouve ensuite le pixel du plus bas niveau de la pyramide où la région en espace fenêtre couverte par ce dernier englobe la géométrie projetée en espace fenêtre.

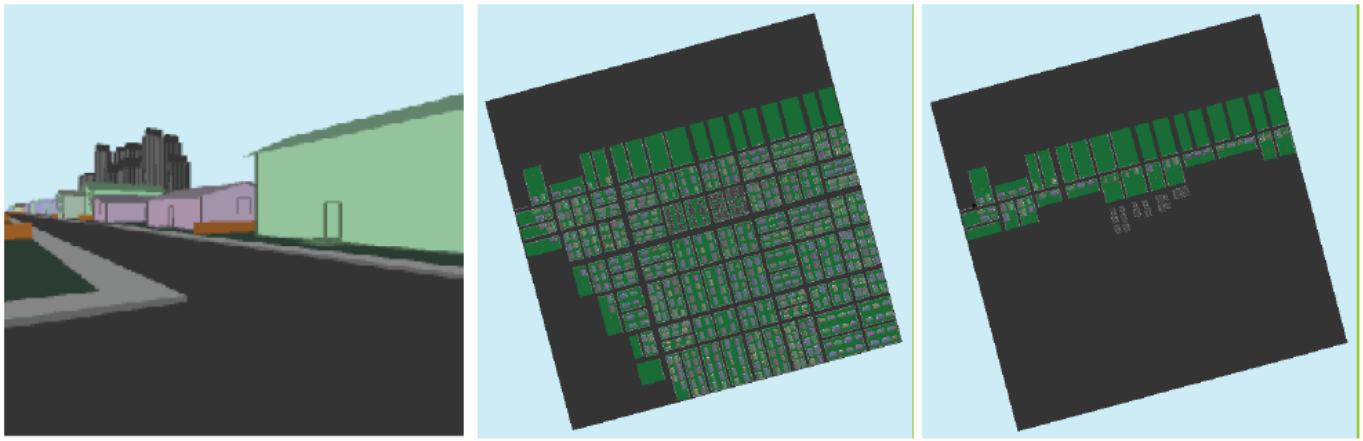
Si la distance la plus petite de notre géométrie s'avère être plus élevée que la distance la plus élevée déjà contenue dans le tampon-z hiérarchique à cet endroit, on sait d'or et déjà que la totalité de la géométrie soumise est occlue, sans avoir besoin de descendre plus bas.

Si le test n'est pas concluant, on vérifie récursivement les valeurs, pour chaque niveau de géométrie inférieur jusqu'à obtenir une valeur indiquant une occultation de la géométrie, ou jusqu'à atteindre le niveau 0, dans quel cas le tout revient à effectuer un test de tampon-z standard. (Il faut alors s'assurer qu'on subdivise aussi notre géométrie pour que ses profondeurs concordent avec les niveaux plus fins de tampon-z hiérarchique.)

# Optimisation selon l'occultation

Pour terminer, rappellons que l'utilisation de requêtes d'occultation possède un poids intrinsèque. Pour le rendu de géométrie simple, quel que soit l'état de la scène, il est parfois préférable d'envoyer directement la géométrie.

Nous concluons cette section avec un exemple d'optimisation du rendu selon l'occultation :



**Gauche :** Scène telle que vue par la caméra. **Centre :** Scène rendue, vue de haut, sans aucune optimisation. **Droite :** Scène rendue, vue de haut pour montrer l'effet de l'optimisation selon l'occultation lorsque cette dernière est appliquée pour le rendu de l'image de gauche. On remarque une diminution importante de la géométrie rendue, la géométrie omise étant occlue par la géométrie effectivement rendue.

# Références recommandées :


Livre T. Akenine-Möller, E. Haines et N.Hoffman, *"Real-Time Rendering", 3rd Edition*, A. K.Peters, Ltd., Natick, MA, USA, 2008. (pp. 645 à 679, pp. 771 à 778)

Article N. Greene, M. Kass, G. Miller, *"Hierarchical Z-Buffer Visibility"*, SIGGRAPH'93 : Proceedings of the 20th annual conference on Computer graphics and interactive techniques, New York, NY, USA, 1993 (pp. 231 - 238)