## Architecture de notre environnement de travail

Nos algorithmes de frustum et d’occlusion culling peuvent être utilisés au travers d’une librairie C. Celle-ci offre différentes fonctions permettant de lancer différents types de frustum culling :

* Entre des frustums pyramidaux et des boîtes alignées sur les axes.
* Entre des frustums pyramidaux et des sphères.
* Entre des frustums sphériques et des boîtes alignées sur les axes.
* Entre des frustums sphériques et des sphères.

Pour fonctionner la librairie doit être utilisée sur un ordinateur doté d’une carte graphique compatible CUDA version 1.1 ou ultérieur. Chaque frustum culling utilise un ou plusieurs « programmes » CUDA appelé kernel.

|  |
| --- |
| Comment installer notre librairie de culling (gpuCuller) ? |
| Dans un premier temps il est nécessaire d’installer les derniers pilotes disponibles pour sa carte graphique.  Deuxièmement, il faut indiquer à votre projet quel fichier bibliothèque utiliser. Différentes versions compilées de GpuCuller sont disponibles : les versions dynamiques et les versions statiques (.dll et .lib). Chacune de ces versions est également déclinée en deux versions, debug et release. La librairie a été compilée sont Visual C++ 2008. L’utilisateur devra également inclure à son projet le fichier gpuCuller.h fourni, qui référence les fonctions disponibles. |

Quelque soit le frustum culling à calculer, la procédure est toujours la même :

* Renseigner les différents tableaux de donnés relatifs aux frustums et aux bounding volumes.
* Activer les tableaux qui seront utilisé pour le culling.
* Demander le calcul du culling.
* Exploiter les résultats.

Avant toute utilisation de la librairie, celle-ci doit être initialisée au travers de la méthode gculInitialize(...).

Pour transférer les données concernant les frustums (pyramidaux ou sphériques)  il faut utiliser les méthodes gculPyramidalFrustumPointers(...) et gculSphericalFrustumPointer(...). Chacune demande un tableau contigü en mémoire contenant les informations nécessaires au culling.

Comme pour les frustums, les données relatives au bounding volumes (boites ou sphères) doivent être fourniees au travers des méthodes gculBoxesPointer(...) et gculSpheresPointer(...).

La librairie permet de séparer l’envoi des données comme précédement expliqué et le calcul de frustum culling.

Pour définir quel frustum culling calculer, l’utilisateur doit activer ou désactiver les différents tableaux disponibles précédement renseignés. Pour cela les méthodes gculEnableArray(...) et gculDisableArray(...) sont indiquées.

Finalement le calcul de frustum culling est effectué avec la fonction gculProcessFrustumCulling(...). Celle-ci se chargera d’effectué le bon frustum culling est fonction des tableaux activés précédement.

Plusieurs exemples seront fourni par la suite pour chaque type de frustum culling.

## Frustum Culling

Le frustum culling est une technique qui permet d’éliminer des objets qui se trouvent dans un champ de vision/perception appelé frustum. Dans notre cas nous devons être capables de dire si un objet simple (sphère ou boîte) est complètement à l’intérieur d’un frustum, à l’extérieur d’un frustum, en intersection avec un frustum ou englobe le frustum.

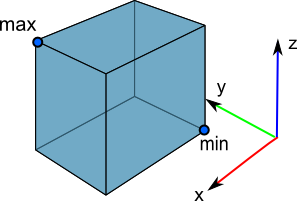
Soit la classification suivante (ici sur un frustum pyramidal et une boite) :

|  |  |
| --- | --- |
| C:\Users\Vallerent\Desktop\inside.png | C:\Users\Vallerent\Desktop\spanning.png |
| Inside | Spanning |
| C:\Users\Vallerent\Desktop\outside.png | C:\Users\Vallerent\Desktop\enclosing.png |
| Outside | Enclosing |

Le but de chaque opération de frustum culling est donc de classer des bounding volumes par rapport à une liste de frustums. La difficulté résidera dans la conversion d’algorithmes d’intersections bien connu en algorithmes utilisant au mieux les capacités de la carte graphique et permettant de traiter en temps réel un grand nombre de frustums et de bounding volumes.

### Frustum Pyramidal / Axis-Aligned Bounding Box

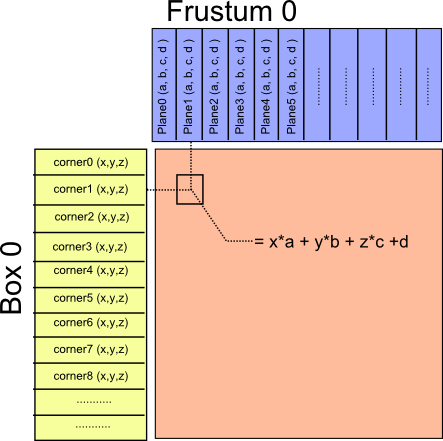
Une Axis-Aligned Bounding Box (AABBox) est une boîte orientée sur les axes du monde. Elle est généralement définie comme un couple de points 3d : un point max et un point min. A partir de ces deux points et grâce à l’alignement sur les axes il est possible de retrouver les huit points 3d définissant l’AABBox.



La classification d’une AABBox avec un frustum pyramidal est classiquement déterminée comme suit :

|  |
| --- |
| Algorithme |
| a = 0  for each frustum plane do  b = 0  for each box corner do  d = <corner, plane.normal> + plane.distance;  if d < 0  b = b + 1  end  if b == 8  return outside  if b == 0  a = a + 1  end  if a == 6  return inside  else  for each frustum corner do  if corner is outside the box  return spanning  end  return enclosing |

Cet algorithme est divisé en trois parties pour profiter au maximum de la parallélisation. Tout d’abord chaque coin de l’AABBox est classé par rapport au six plans du frustum. Cette opération correspond tout simplement à une multiplication de matrices si on organise comme suit les données :



La matrice résultante contient ainsi la position des coins de chaque boite par rapport aux plans de chaque frustum. Par convention nous avons orienté les normales des plans du frustum vers l’intérieur du volume.

La deuxième étape consiste en l’exploitation de la matrice calculée à l’étape une. Si les six coins d’une boite sont derrières au moins un plan du frustum, la boite est déclarée outside. A contrario si les six coins sont devant tous les plans du frustum alors la boite est définie inside.

Pour détecter les deux derniers cas (spanning et enclosing) une troisième étape est effectuée sur le gpu. Dans cette étape, à partir des coins de la boite et du frustum, on regarde si un coin du frustum est la l’extérieur de la boite. Dans ce cas la boite est déclarée spanning, enclosing sinon.

|  |
| --- |
| Code d’exemple |
| int planeSize = frustumCount \* 6 \* 4;  int cornerSize = frustumCount \* 8 \* 4;  float\* frustumData = new float[ planeSize + cornerSize ];  float\* boxData = new float[ boxCount \* 8 \* 4 ];  // result contiendra la classification des boites par rapport aux frustums  GCUL\_Classification\* result = new GCUL\_Classification[ frustumCount \* boxCount ];  // L'utilisateur rempli ici les tableaux relatifs aux  // boites et aux frustums.  FillFrustumData( frustumCount, frustumData );  FillBoxData ( boxCount, boxData );  // Définition des données sur lesquelles travailler  gculBoxesPointer( boxCount, GCUL\_FLOAT, boxData );  gculPyramidalFrustumPlanesPointer ( frustumCount, GCUL\_FLOAT, frustumData );  gculPyramidalFrustumCornersPointer( frustumCount, GCUL\_FLOAT, &frustumData[frustumCount\*6\*4 ]);  // Activation des tableaux nous intéressant  gculEnableArray( GCUL\_PYRAMIDALFRUSTUMCORNERS\_ARRAY );  gculEnableArray( GCUL\_PYRAMIDALFRUSTUMPLANES\_ARRAY );  gculEnableArray( GCUL\_BBOXES\_ARRAY );  // Calcul du culling  gculProcessFrustumCulling( result );  // La classification de la boite i par rapport au frustum j  // est accessible à l'indice i\*frustumCount + j  int i = 0; int j = 0;  GCUL\_Classification c = result[i\*frustumCount + j];  delete[] result;  delete[] boxData;  delete[] frustumData; |

### Frustum Pyramidal / Sphere

### Frustum Sphérique / Axis-Aligned Bounding Box

### Frustum Sphérique / Sphere

De part la nature géométrique de la sphère, le test de classification entre deux sphères simple et rapide.

|  |
| --- |
| Algorithme |
| d = distance( frustum.center, sphere.center )  if d > frustum.radius + sphere.radius )  return outside  else if d + sphere.radius <= frustum.radius  return enclosing  else if d + frustum.radius <= sphere.radius  return inside  else  return spanning |

Nous retrouvons ainsi nos quatre cas, illustrés ci-dessous.

|  |  |
| --- | --- |
| C:\Users\Vallerent\Desktop\shinside.png | C:\Users\Vallerent\Desktop\shspanning.png |
| Inside | Spanning |
| C:\Users\Vallerent\Desktop\out.png | C:\Users\Vallerent\Desktop\enclo.png |
| Outside | Enclosing |

Le calcul du frustum culling peut se faire en une étape. Chaque thread du kernel se voit attribué le calcul de la classification d’une sphère par rapport à un frustum sphérique. Nous avons ainsi un nombre de threads égal au nombre de frustums multiplié par le nombre de sphères. Le code exécuté par chaque thread est identique à l’algorithme présenté précédemment.

|  |
| --- |
| Code d’exemple |
| int planeSize = frustumCount \* 6 \* 4;  int cornerSize = frustumCount \* 8 \* 4;  float\* frustumData = new float[ frustumCount \* 4 ];  float\* sphereData = new float[ sphereCount \* 4 ];  GCUL\_Classification\* result = new GCUL\_Classification[ frustumCount \* sphereCount ];  FillFrustumData( frustumCount, frustumData );  FillSphereData( sphereCount, sphereData );  gculSpheresPointer( sphereCount, GCUL\_FLOAT, sphereData );  gculSphericalFrustumPointer( frustumCount, GCUL\_FLOAT, frustumData );  gculEnableArray( GCUL\_SPHERICALFRUSTUM\_ARRAY );  gculEnableArray( GCUL\_BSPHERES\_ARRAY );  gculProcessFrustumCulling( result );  delete[] result;  delete[] sphereData;  delete[] frustumData; |