|  |
| --- |
| Universite de Technologie de Belfort-Montbeliard |
| TX52 |
| Implémentation d’algorithmes de visibilite sur GPU avec CUDA |
|  |
| **Arnaud VALLERENT – Nicolas SAID** |
| **Semestre de Printemps 2009** |

|  |
| --- |
|  |

Table des matières

[Introduction 3](#_Toc233134252)

[Presentation de nVidia CUDA 4](#_Toc233134253)

[Implementation des algorithmes 7](#_Toc233134254)

[Architecture de notre environnement de travail 7](#_Toc233134255)

[Stratégie de parallélisation 8](#_Toc233134256)

[Frustum Culling 8](#_Toc233134257)

[Frustum Pyramidal / Axis-Aligned Bounding Box 9](#_Toc233134258)

[Frustum Pyramidal / Sphere 11](#_Toc233134259)

[Frustum Spherique / Axis-Aligned Bounding Box 13](#_Toc233134260)

[Frustum Sphérique / Sphère 14](#_Toc233134261)

[Occlusion Culling 17](#_Toc233134262)

[Méthode utilisée 17](#_Toc233134263)

[Occlusion dans un Frustum Pyramidal 18](#_Toc233134264)

[Occlusion dans d’autres Frustums 19](#_Toc233134265)

[Résultats 20](#_Toc233134266)

[Approfondissements 21](#_Toc233134267)

[Structure de données pour l’accélération 21](#_Toc233134268)

[Construction de la structure 21](#_Toc233134269)

[Lecture de la structure 21](#_Toc233134270)

[Conclusion 23](#_Toc233134271)

[Bibliographie 24](#_Toc233134272)

# Introduction

Dans le cadre de l’UV TX52 (Travaux de Laboratoire) proposée à l’UTBM, nous avons étudié des algorithmes de calcul de visibilite comme le Frustum Culling ou l’Occlusion Culling.

L’objet de nos travaux était d’évaluer la parallélisation potentielle de ces algorithmes, afin de les adapter pour être exécutés sur des cartes graphiques programmables.

La possibilité d’exécuter ces algorithmes de manière parallèle permettrait de les utiliser afin, par exemple, d’évaluer la visibilité d’entités dans les champs de vision d’agents autonomes au sein d’une plateforme de simulation et/ou de réalité virtuelle.

Ces algorithmes proviennent du monde de l’imagerie numérique, et généralement, un seul calcul par rafraichissement d’image s’avère nécessaire, pour les applications considérées. Dans le cas d’une application de simulation avec des agents autonomes possédant chacun un champ de vision, ces algorithmes doivent donc s’exécuter plusieurs fois à chaque rafraichissement de l’univers. Notre tâche est donc d’évaluer la performance de ces algorithmes utilisés en parallèle sur des processeurs graphiques programmables.

Pour implémenter ces algorithmes, nous avons utilisé la technologie nVidia CUDA, qui permet de programmer directement en C des processeurs graphiques (GPU).

Dans une première partie, nous présenterons la technologie utilisée, ses spécificités, ses avantages et ses contraintes.

Ensuite nous étudierons les algorithmes parallélisés, ainsi que leurs implémentations adaptées pour CUDA.

Nous présenterons également les résultats que nous avons obtenus en utilisant ces algorithmes.

Puis nous verrons quelles améliorations qu’il serait possible d’apporter à nos travaux, notamment l’utilisation éventuelle de structures de partitionnement de l’espace.

# Presentation de nVidia CUDA

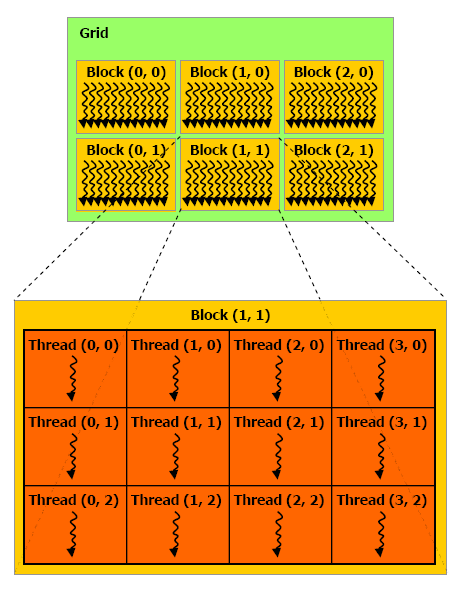
La technologie CUDA (*Compute Unified Device Architecture*) est une technologie de [GPGPU](http://fr.wikipedia.org/wiki/GPGPU) (*General-Purpose Computing on Graphics Processing Units*) qui introduit un nouveau modèle de programmation pour effectuer du calcul parallèle généraliste sur des processeurs graphique.

Globalement, CUDA permet de programmer les GPU (GPU nVidia) en utilisant une variante du langage C.

Ce modèle de programmation est base sur l’architecture SIMT, pour Single Instruction Multiple Thread.

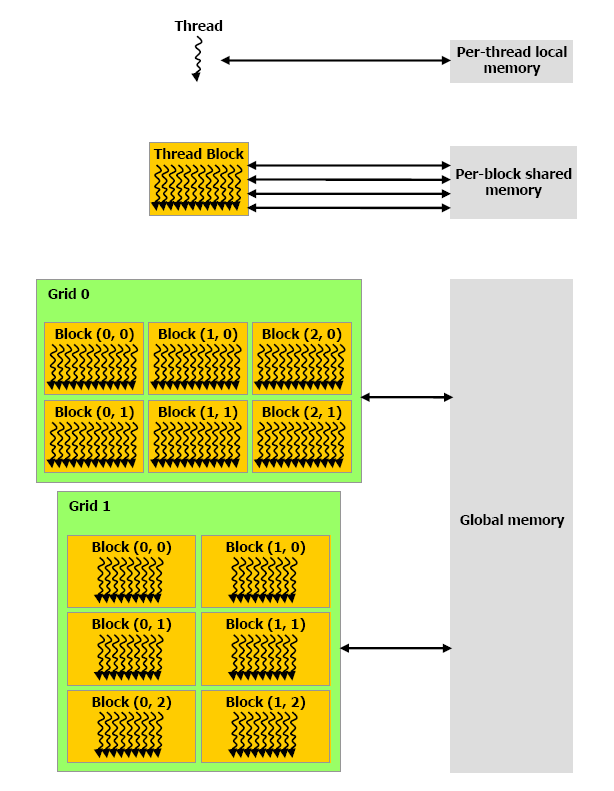
En effet, CUDA permet d’écrire des fonctions en langage C, appelles « kernels » qui, lorsqu’elles sont appelées depuis un programme, sont exécutées N fois en parallèle sur le GPU par N threads CUDA différents.

Ces threads sont organisés en blocs de threads, et les blocs sont organises dans une grille de blocs.



Les threads ont accès à différentes zones mémoire situées sur la carte graphique.

* La mémoire globale de la grille
* La mémoire partagée des blocs
* La mémoire locale aux threads



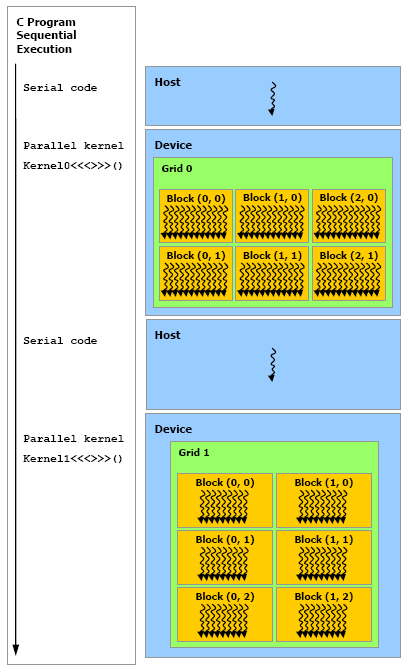
Il faut savoir utiliser les différentes mémoires au mieux. Par exemple, l’accès à la mémoire partagée est beaucoup plus rapide que l’accès à la mémoire globale. Cependant sa taille est limitée, d’autant plus qu’elle est partagée par tous les threads du même bloc. Il faut donc porter un intérêt tout particulier a l’implémentation en expérimentant différentes solutions.

Un autre aspect important de CUDA est la différenciation entre l’hôte et la carte graphique. En CUDA, on considère la carte graphique comme un coprocesseur installé sur la machine.

Ainsi, pour qu’un kernel puisse accéder à des données présentes sur la machine, il faudra tout d’abord les transférer sur la carte graphique.

Les lancements de kernels et les transferts de données vers/de la carte graphique étant couteux en termes de temps d’exécution, il faut également veiller à les limiter.

Ainsi, il est possible de représenter schématiquement l’exécution d’un programme CUDA comme une succession d’exécution de code séquentiel sur la machine hôte, et de code parallèle sur la carte graphique.



Cette introduction à CUDA couvre normalement toutes les notions nécessaires a la compréhension du travail que nous avons accompli dans le cadre de cette UV TX52.

Cependant, je recommande tout de même la lecture du guide de programmation CUDA officiel, disponible sur le site de nVidia.

# Implémentation des algorithmes

## 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 et de vérifier la compatibilité de sa carte sur le site de nVidia.  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 sous 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 en fonction des tableaux activés précédement. Cette fonction renvoi le résultat sous la forme d’un tableau où chaque case contient la classification d’un bounding volume par rapport à un frustum.

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

## Stratégie de parallélisation

Pour ce projet, il est facile d’identifier plusieurs stratégies pour la parallélisation. L’important pour déterminer cette stratégie est de déterminer s’il existe des dépendances temporelles entre les différentes étapes de calcul, ou des dépendances de données.

Premièrement, ce qui nous est demandé est d’effectuer du frustum culling pour chaque frustum. Chaque frustum étant indépendant, il est donc possible de paralléliser le calcul sur le nombre de frustum à traiter.

Deuxièmement, le résultat du culling d’une entité est indépendant des résultats de culling des autres entités. Il est donc possible de paralléliser plus encore notre application, sur le nombre d’entité à traiter.

Finalement, en considérant ces deux faits, il apparait que chaque opération de culling frustum/entité est indépendante des autres. Nos opérations de culling peuvent donc opérer en effectuant opérations de culling de manière parallèle. Ainsi nous avons choisi de tester chaque entité de l’univers pour le culling dans chaque frustum de l’univers.

Et finalement, dans certains cas, comme pour le cas du frustum pyramidal, il est possible de scinder l’opération de culling en de multiples sous-opérations en utilisant les propriétés géométriques du frustum considéré, ou bien encore les propriétés géométriques des entités à traiter (par exemple, les AABB sont composées de points, et les frustums pyramidaux sont composés de plans… Il est donc possible de paralléliser l’opération en considérant ces sous-structures géométriques).

## 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 un frustum.

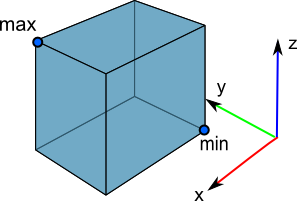
Soit la classification suivante (ici sur un frustum pyramidal et une boite) :

|  |  |
| --- | --- |
| 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

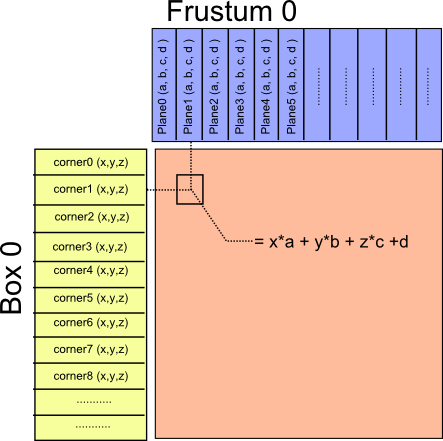
Une Axis-Aligned Bounding Box (AABBox) est une boîte orientée sur les axes du monde. Elle est généralement définie comme un couple de points 3d : un point max et un point min. A partir de ces deux points et grâce à l’alignement sur les axes il est possible de retrouver les huit points 3d définissant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

Une sphère est définie par la position de son centre, et son rayon. Ainsi, la définition d’une sphère peut se faire en utilisant quatre coordonnées en virgule flottante.

La classification d’une sphère avec un frustum pyramidal se fait généralement comme suit :

|  |
| --- |
| Algorithme |
| Counter = 0  for each frustum plane do  d = <plane.normal, sphere.position> + plane.distance  if( p > 0 )  if abs(d) >= sphere.radius  return outside;  End  Else  if abs(d) >= sphere.radius  Counter = Counter + 1  End  end  end  if Counter == 0  return inside  else if Counter == 6  return inside  else  return spanning  end |

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

L’implémentation CUDA est réalisée en deux passes:

* Un premier kernel évalue la pénétration de la sphère par rapports aux plans du frustum pyramidal (1 thread = 1 évaluation sphère/plan)
* Puis un deuxième kernel conclut sur la situation de la sphère par rapport au frustum, en évaluant les six valeurs calculées par le premier kernel (1 thread = 1 évaluation sphère/frustum)

Le premier kernel permet de « dérouler » la boucle présente dans l’algorithme initial, ce qui augmente le parallélisme.

|  |
| --- |
| Code d’exemple |
| int planeSize = frustumCount \* 6 \* 4;  int cornerSize = frustumCount \* 8 \* 4;  float\* frustumData = new float[ planeSize + cornerSize ];  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 );  gculPyramidalFrustumPlanesPointer ( frustumCount, GCUL\_FLOAT, frustumData );  gculPyramidalFrustumCornersPointer( frustumCount, GCUL\_FLOAT, &frustumData[frustumCount\*6\*4] );  gculEnableArray( GCUL\_PYRAMIDALFRUSTUMCORNERS\_ARRAY );  gculEnableArray( GCUL\_PYRAMIDALFRUSTUMPLANES\_ARRAY );  gculEnableArray( GCUL\_BSPHERES\_ARRAY );  gculProcessFrustumCulling( result );  delete[] result;  delete[] sphereData;  delete[] frustumData; |

### Frustum Spherique / Axis-Aligned Bounding Box

Pour classer des AABBox par rapport à des frustums sphériques nous utilisons une version modifiée de l’algorithme de Arvo.

|  |
| --- |
| Algorithme |
| s = computeBoundedDistance( frustum.x, box.min.x, box.max.x );  d += s\*s;  s = computeBoundedDistance( frustum.y, box.min.y, box.max.y );  d += s\*s;  s = computeBoundedDistance( frustum.z, box.min.z, box.max.z );  d += s\*s;  squaredRadius = frustum.radius\*frustum.radius;  if(d > squaredRadius)  return outside;  else if d == 0  && 2\*frustum.radius <=  min(box.min-box.max.x,box.min.y-box.max.y,box.min.z-box.max.z)  return inside;  else  a = 0;  for each box corner do  if distanceSquared(frustum.center, corner) <= squaredRadius  a = a + 1  end  if a == 8  return enclosing  else  return spanning  endif |

Afin de mieux visualiser l’algorithme voici les différents cas possibles :

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

Cet algorithme est implémenté avec un seul kernel, en une seule passe. Chaque thread s’occupe de tester une boîte avec un frustum sphérique.

|  |
| --- |
| Code d’exemple |
| float\* frustumData = new float[ frustumCount \* 4 ];  float\* boxData = new float[ boxCount \* 8 \* 4 ];  GCUL\_Classification\* result = new GCUL\_Classification[ frustumCount \* boxCount ];  FillFrustumData( frustumCount, frustumData );  FillBoxData( boxCount, boxData );  gculBoxesPointer( boxCount, GCUL\_FLOAT, boxData );  gculSphericalFrustumPointer( frustumCount, GCUL\_FLOAT, frustumData );  gculEnableArray( GCUL\_SPHERICALFRUSTUM\_ARRAY );  gculEnableArray( GCUL\_BBOXES\_ARRAY );  gculProcessFrustumCulling( result );  delete[] result;  delete[] boxData;  delete[] frustumData; |

### Frustum Sphérique / Sphère

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; |

## Occlusion Culling

### Méthode utilisée

Les méthodes d’occlusion culling existantes sont dédiées à l’imagerie numérique, et plus particulièrement au temps réel, comme dans les jeux vidéo. Elles utilisent des outils mis à disposition par le pipeline graphique utilisé pour le rendu. Ainsi, les techniques les plus courantes utilisent des propriétés du Z-Buffer.

Les autres approches qui peuvent être utilisées sont purement géométriques. La technique du Shadow Frustum a été envisagée pour notre application, mais finalement nous avons retenu une autre méthode, plus parallelisable.

La technique qui est utilisée dans notre application est celle du lancer de rayon. Un ensemble de rayon est généré, couvrant au mieux le champ de vision considéré. En calculant les intersections entre ces rayons et les objets présents dans le champ de vision, nous en déduisons la liste des objets qui sont directement visibles depuis le point de vue. Les autres sont considérés comme étant cachés.

Pour l’instant, seule l’occlusion d’AABB dans des Frustum Pyramidaux est implémentée. Elle n’est pas encore fonctionnelle. Elle ne fonctionne que pour un seul frustum. Son débogage était encore en cours lors de la rédaction de ce rapport.

Deux kernels (donc deux passes de calcul) sont utilisés pour l’occlusion culling :

**Un kernel de génération de rayon :**

Dans ce kernel, chaque thread généré un rayon au sein d’un frustum donné, à partir des coordonnées des informations géométriques du frustum (positions des vertices, dans le cas d’un frustum pyramidal).

Frustum 4

Frustum 3

Frustum 2

Frustum 1

Th

Th

Th

Th

Th

Th

Th

Th

Th

Th

Th

Th

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

**Un kernel d’intersection de rayon avec une AABB :**

Dans ce kernel, chaque thread calcule la distance d’intersection entre une AABB et un rayon.

Une troisième passe exécutée sur le CPU est ensuite utilisée pour calculer les boites pour lesquelles une distance d’intersection minimale a été trouvée, et mettre à jour l’information de visibilité des différentes AABB composant notre univers.

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Ray

Box

Th

Box

Box

Box

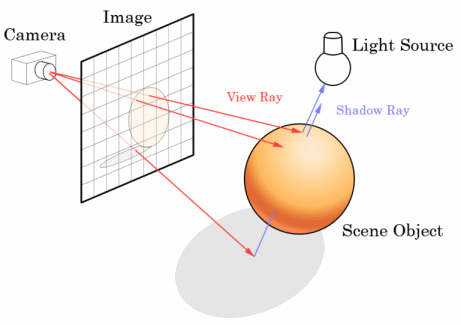
Box

Box

La méthode utilisée pour calculer l’intersection est celle décrite par Brian Smits.

### Occlusion dans un Frustum Pyramidal

La génération des rayons pour un frustum pyramidal correspond à ce que l’on pourrait trouver dans une application de lancer de rayon. Le point de départ des rayons se situe à la position du point de vue. L’ensemble de leurs directions décrit un balayage régulier de « l’écran ». Bien qu’ici l’écran soit en fait le plan proche de notre frustum pyramidal.



### Occlusion dans d’autres Frustums

L’implémentation de l’occlusion culling pour d’autres types de frustum ne diffère du cas pyramidal que par le kernel de génération de rayon. Il est possible d’adapter ce kernel pour qu’il génère l’ensemble de rayons correspondant au frustum à traiter.

Par exemple, pour une sphère, en fixant le point de départ des rayons au centre de la sphère et en les faisant pointer vers un ensemble de points régulièrement repartis sur la surface de la sphère, il serait possible d’obtenir un ensemble de rayons adéquat.

# Résultats

Des tests de performances ont été effectués afin juger de la pertinence de l’utilisation du GPU pour les algorithmes de frustum culling. Chaque type de frustum culling a été testé avec un nombre maximum de 10000 bounding volumes par 1000 frustums. Il est possible d’aller au-delà en séparant en plusieurs opérations le frustum culling souhaité.

Le processus de test choisi mesure le temps d’exécution nécessaire au frustum culling de n bounding volumes par m frustums. Les valeurs n et m sont étalonnée selon une échelle logarithmique. Nous obtenons ainsi une matrice contenant les différents temps d’exécution. Ces résultats sont alors visualisés sous forme d’une surface 3d à l’aide de matlab.

Voici quelques coupes de ces surfaces.

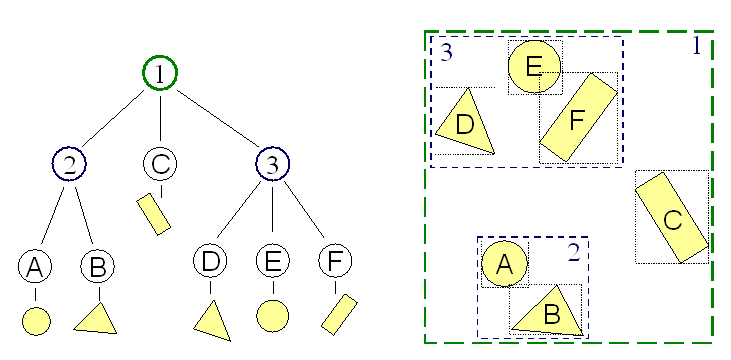
|  |  |
| --- | --- |
| C:\Users\Vallerent\Documents\MATLAB\pyrbox1000.png | C:\Users\Vallerent\Documents\MATLAB\pyrsphere1000.png |
| C:\Users\Vallerent\Documents\MATLAB\spherebox1000.png | C:\Users\Vallerent\Documents\MATLAB\spheresphere1000.png |

# Approfondissements

## Structure de données pour l’accélération

Notre approche pour le frustum culling ne prend pas du tout en compte la distribution spatiale des éléments. Elle se contente de tester exhaustivement tout les éléments.

Au cours du projet nous avons décidé d’étudier la possibilité d’utiliser une structure de division de l’espace pour tenter d’accélérer encore le traitement. Nous avons décide d’étudier l’utilisation d’une hiérarchie de volumes englobant (BVH), pour stocker les informations des entités de l’univers. Seules les AABB ont été considérées.



Malheureusement, nous n’avons pas pu mener à bien ces travaux, faute de temps. Le projet comporte le début de l’implémentation de ces modifications. Nous allons également expliquer les méthodes qui ont été envisagées pour la construction et l’utilisation de telles structures.

### Construction de la structure

La structure de partitionnement utilisée est un BVH binaire, construit sur le CPU. La construction de ce BVH se fait « top-down » en effectuant des tris sur les positions des centroides des AABB. La qualité de cet arbre est surement très faible, mais sa simplicité facilite la réflexion et l’implémentation.

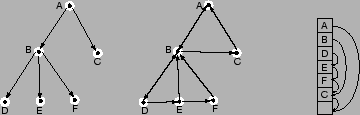
Il existe des méthodes de construction de BVH sur GPU dans la littérature, également des méthodes de construction de kD-Tree. Il serait donc avantageux de s’intéresser à une implémentation de la construction sur GPU plutôt que sur CPU, si les gains en terme de performance sont intéressants.

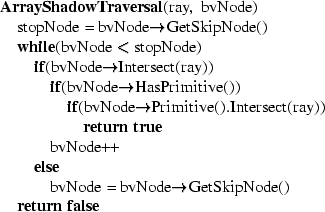
### Lecture de la structure

La lecture d’une structure hiérarchique en CUDA peut se révéler difficile. Premièrement, les appels récursifs dans les kernels sont interdits. Donc il est impossible d’adapter directement les algorithmes de traversée hiérarchique tel quel.

Une des solution consiste à construire en « device memory » une pile faisant office de pile pour les appels récursifs, et la gérer manuellement. Le problème est que pour que la traversée soit rapide, il faut que cette pile se trouve en mémoire partagée. Or, cette mémoire est en quantité limitée, et de plus elle est partagée par tous les threads du bloc, chacun exécutant sa propre descente dans la structure hiérarchique. Cela limite donc la profondeur de descente dans l’arbre, et il faut donc adapter l’arbre de manière a ce que sa profondeur soit adaptée.

La deuxième solution consiste à convertir l’arbre hiérarchique en « Threaded Tree », dans lequel chaque nœud possède un lien supplémentaire pointant vers le prochain nœud à visiter en cas d’échec de la descente sur le nœud courant.





Il est donc ensuite possible de convertir cet arbre en simple liste. Ainsi, l’ordre de visite des nœuds est stocke en même temps que les nœuds, dans la liste. Il n’est donc plus nécessaire d’utiliser des appels récursifs ou des piles.

Cette solution n’a malheureusement pas pu être testée, il serait intéressant d’étudiera les impacts sur la performance entre les différentes méthodes de traversée évoquées.

# Conclusion

Ce travail de laboratoire a été l’occasion pour nous d’étudier et de travailler sur un projet intéressant, aux applications concrètes et directes. De plus, cela nous a permis d’apprendre à utiliser la technologie nVidia CUDA, ainsi que d’approfondir nos connaissances en programmation parallèle. D’autant plus que cette technologie est de plus en plus utilisée autant dans l’industrie que dans le monde de la recherche. Il ne fait nul doute que ce que nous avons appris en réalisant ce projet nous sera utile pour notre vie professionnelle.

Malheureusement, il n’existe pas, à notre connaissance, de projets similaires avec lesquels nous pourrions comparer les résultats. De même qu’il nous était impossible de comparer ces résultats avec celles obtenues via les méthodes développées au laboratoire. C’est pour cette raison qu’il nous est difficile de conclure quant à l’intérêt de l’usage de GPGPU pour le calcul de visibilité. Nous laissons donc cette tache à nos professeurs et lecteurs.

Ce projet nécessiterait un peu plus de travail. Le développement avec CUDA peut être long car les comportements des programmes sont parfois imprévisibles. L’expérimentation, l’essai et l’erreur font partie intégrante de l’apprentissage de cet outil, et cela prend du temps.

Il aurait été donc intéressant de pouvoir finir certaines parties du projet qui sont restées malheureusement inachevées, comme l’Occlusion Culling qui n’est malheureusement pas tout à fait fonctionnel, ou bien l’utilisation de structure de partitionnement de l’espace. Cela pourra peut être faire l’objet d’une prochaine TX ou TO.

# Bibliographie

Christen, M. (s.d.). Ray Tracing on GPU. University of Applied Sciences Basel (FHBB).

*Frustum Culling.* (s.d.). Récupéré sur Flipcode: www.flipcode.com

Hudson, Manocha, Cohen, Lin, Hoff, & Zhang. (s.d.). Accelerated Occlusion Culling using Shadow Frusta.

Lauterbach, Garland, Sengupta, Luebke, & Manocha. (2009). Fast BVH Construction on GPUs. *EUROGRAPHICS.*

*nVidia Forum.* (s.d.). Récupéré sur nVidia website: http://forums.nvidia.com/index.php?showtopic=97118

nVidia. *nVidia CUDA - Programming Guide.*

Papaioannou, G., Gaitatzes, A., & Christopoulos, D. (s.d.). Efficient Occlusion Culling using Solid Occluders.

*Threaded Binary Tree.* (s.d.). Récupéré sur Wikipedia: http://en.wikipedia.org/wiki/Threaded\_binary\_tree

*View Frustum Culling.* (s.d.). Récupéré sur Lighthouse3d: http://www.lighthouse3d.com/opengl/viewfrustum/index.php?refs