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

|  |
| --- |
| [Type the abstract of the document here. The abstract is typically a short summary of the contents of the document. Type the abstract of the document here. The abstract is typically a short summary of the contents of the document.] |

Table des matieres

[Remerciements 3](#_Toc232965650)

[Introduction 4](#_Toc232965651)

[Presentation de nVidia CUDA 5](#_Toc232965652)

[Implementation des algorithmes 5](#_Toc232965653)

[Architecture de notre environnement de travail 5](#_Toc232965654)

[Strategie de parallelisation 5](#_Toc232965655)

[Frustum Culling 5](#_Toc232965656)

[Frsutum Pyramidal / Axis-Aligned Bounding Box 5](#_Toc232965657)

[Frustum Pyramidal / Sphere 5](#_Toc232965658)

[Frustum Spherique / Axis-Aligned Bounding Box 5](#_Toc232965659)

[Frustum Spherique / Sphere 5](#_Toc232965660)

[Occlusion Culling 5](#_Toc232965661)

[Methode utilisee 5](#_Toc232965662)

[Occlusion dans un Frustum Pyramidal 5](#_Toc232965663)

[Occlusion dans d’autres Frustums 5](#_Toc232965664)

[Resultats 5](#_Toc232965665)

[Approfondissements 5](#_Toc232965666)

[Structure de donnees pour l’acceleration 5](#_Toc232965667)

[Construction de la structure 5](#_Toc232965668)

[Lecture de la structure 5](#_Toc232965669)

[Conclusion 5](#_Toc232965670)

[Works Cited 6](#_Toc232965671)

# Remerciements

Nous tenons a remercier chaleureusement …

# Introduction

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

L’objet de nos travaux était d’evaluer la parallelisation potentielle de ces algorithmes, afin de les adapter pour etre executes sur des cartes graphiques programmables.

La possibilite d’executer ces algorithmes de manière parallele permettrait de les utiliser afin, par exemple, d’evaluer la visibilite d’entites dans les champs de vision d’agents autonomes au sein d’une plateforme de simulation et/ou de realite virtuelle.

Ces algorithmes proviennent du monde de l’imagerie numerique, et generalement, un seul calcul par rafraichissement d’image s’avere necessaire, pour les applications considerees. Dans le cas d’une application de simulation avec des agents autonomes possedant un champs de vision, ces algorithmes doivent donc s’executer plusieurs fois a chaque rafraichissement de l’univers. Notre tache est donc d’evaluer la performance de ces algorithmes utilises en parallele sur des processeurs graphiques programmables.

Pour implementer ces algorithmes, nous avons utilise la technologie nVidia CUDA, qui permet de programmer directement en C des processeurs graphiques.

Dans une premiere partie, nous presenterons la technologie utilisee, ses specifites, ses avantages et ses contraintes.

Ensuite nous etudierons les algorithmes parallelises, ainsi que leurs implementations adaptees pour CUDA.

Nous presenterons egalement les resultats que nous avons obtenus en utilisant ces algorithmes, et les commenterons.

Puis nous verrons quelles ameliorations il serait possible d’apporter a nos travaux, nottament l’utilisation eventuelle de structure de partitionnement de l’espace.

# Presentation de nVidia CUDA

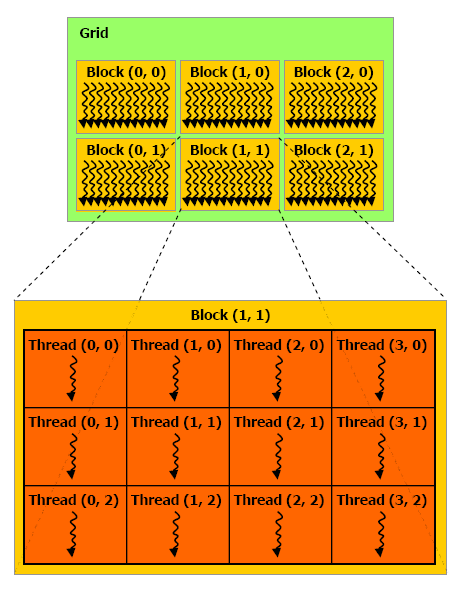
La technologie CUDA (***Compute Unified Device Architecture***) est une technologie de [GPGPU](http://fr.wikipedia.org/wiki/GPGPU) (*General-Purpose Computing on Graphics Processing Units*) qui introduit un nouveau modele de programmation pour effectuer du calcul parallele generaliste sur des processeurs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

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

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

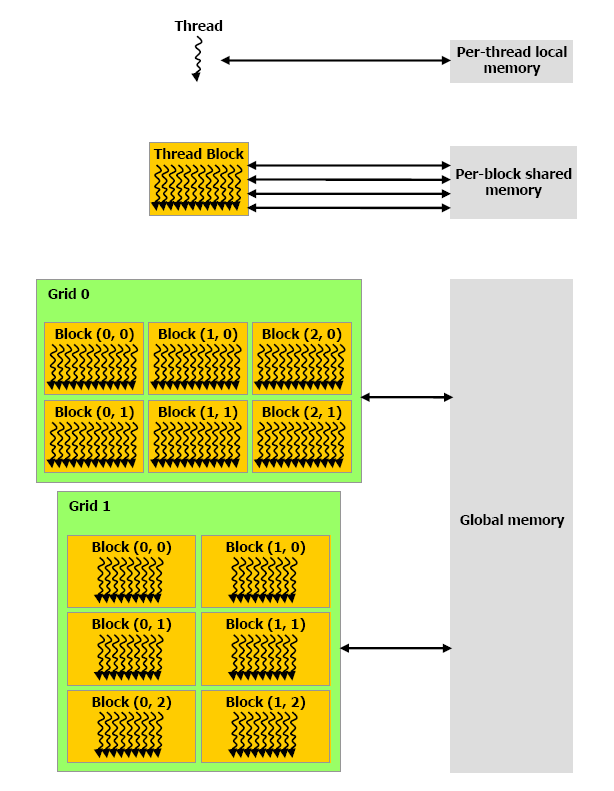
En effet, CUDA permet d’ecrire des fonctions en langage C, appelles « kernels » qui, lorsqu’elles sont appellees depuis un programme, sont executees N fois en parallele sur le GPU par N threads CUDA differents.

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



Les threads ont acces a differentes zones memoire situees sur la carte graphique.

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



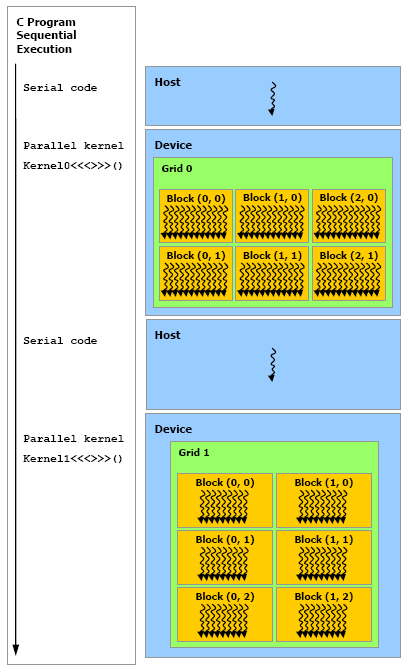
Il faut savoir utiliser les differentes memoires au mieux. Par exemple, l’acces a la mémoire partagee est beaucoup plus rapide que l’accees a la mémoire globale. Cependant sa taille est limitee, d’autant plus qu’elle est partagee par tout les threads du meme bloc. Il faut donc porter un interet tout particulier a l’implementation en experimentant differentes solutions.

Un autre aspect important de CUDA est la differenciation entre l’hote et la carte graphique. En CUDA, on considere la carte graphique comme un co-processeur installe sur la machine.

Ainsi, pour qu’un kernel puisse acceder a des donnees presentes sur la machine, il faudra tout d’abord les transferer sur la carte graphique.

Les lancements de kernels et les transferts de donnees vers/de la carte graphique etant couteux en terme de temps d’execution, il faut egalement veiller a les limiter.

Ainsi, il est possible de reprensenter schematiquement l’execution d’un programme CUDA comme une succession d’execution de code sequentiel sur la machine hote, et de code parallele sur la carte graphique.



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

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

# Implementation des algorithmes

## Architecture de notre environnement de travail

## Strategie de parallelisation

Pour ce projet, il est facile d’identifier plusieurs strategies pour la parallelisation. 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 demande 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 composes de plans… Il est donc possible de paralléliser l’opération en considérant ces sous-structures géométriques).

## Frustum Culling

### Frustum Pyramidal / Axis-Aligned Bounding Box

### Frustum Pyramidal / Sphere

### Frustum Pyramidal / Sphere

Une sphere est definie par la position de son centre, et son rayon. Ainsi, la definition d’une sphere peut se faire en utilisant quatre coordonnees en virgule flottante.

La classification d’une sphere avec un frustum pyramidal se fait generalement 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 |
|  |

L’implementation CUDA est realisee en deux passes:

* Un premier kernel evalue la penetration de la sphere par rapports aux plans du frustum pyramidal (1 thread = 1 evaluation sphere/plan)
* Puis un deuxieme kernel conclut sur la situation de la sphere par rapport au frustum, en evaluant les six valeurs calculees par le premier kernel (1 thread = 1 evaluation sphere/frustum)

Le premier kernel permet de « derouler » la boucle presente dans l’algorithme initial, ce qui augmente le parallelisme.

### Frustum Spherique / Axis-Aligned Bounding Box

### Frustum Sphérique / Sphère

## 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 a disposition par le pipeline graphique utilise 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 caches.

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 debogage était encore en cours lors de la redaction de ce rapport.

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

**Un kernel de génération de rayon :**

Dans ce kernel, chaque thread généré un rayon au sein d’un frustum donne, a 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

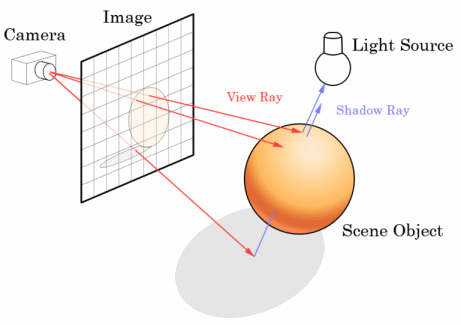
Box

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.

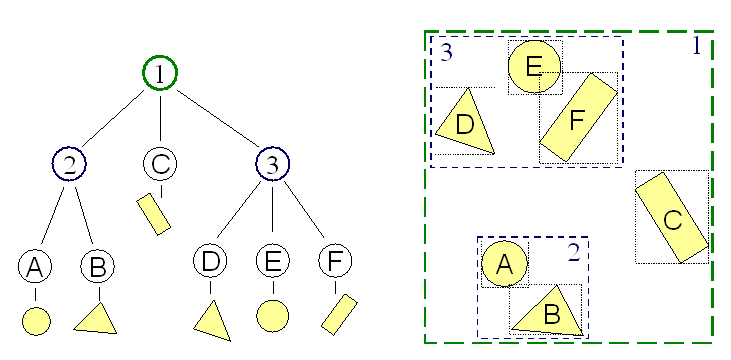
# Résultats

# Approfondissements

## Structure de données pour l’accélération

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

Au cours du projet nous avons decide d’etudier la possibilite d’utiliser une structure de division de l’espace pour tenter d’accelerer encore le traitement. Nous avons decide d’etudier l’utilisation d’une hierarchie de volumes englobants (BVH), pour stocker les informations des entites de l’univers. Seules les AABB ont été considerees.



Malheureusement, nous n’avons pas pu mener a bien ces travaux, faute de temps. Le projet comporte le debut de l’implementation de ces modifications. Nous allons egalement expliquer les methodes qui ont été envisagees pout la construction et l’utilisation de telles structures.

### Construction de la structure

La structure de partitionnement utilisee 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 qualite de cet arbre est surement tres faible, mais sa simplicite facilite la reflexion et l’implementation.

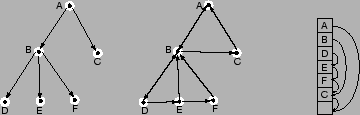
Il existe des methodes de construction de BVH sur GPU dans la littérature, egalement des methodes de construction de kD-Tree. Il serait donc avantageux de s’interesser a une implementation de la construction sur GPU plutôt que sur CPU, si les gains en terme de performance sont interessants.

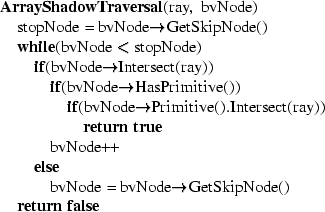
### Lecture de la structure

La lecture d’une structure hierarchique en CUDA peut se reveler difficile. Premierement, les appels recursifs dans les kernels sont interdits. Donc il est impossible d’adapter directement les algorithmes de traversee hierarchique tel quel.

Une des solution consiste a construire en « device memory » une pile faisant office de pile pour les appels recursifs, et la gerer manuellement. Le probleme est que pour que la traversee soit rapide, il faut que cette pile se trouve en mémoire partagee. Or, cette mémoire est en quantite limitee, et de plus elle est partagee par tout les thread du bloc, chacun executant sa propre descente dans la structure hierarchique. 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 adaptee.

La deuxieme solution consiste a convertir l’arbre hierarchique en « Threaded Tree », dans lequel chaque nœud possede un lien supplementaire pointant vers le prochain nœud a visiter en cas d’echec 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 meme temps que les nœuds, dans la liste. Il n’est donc plus necessaire d’utiliser des appels recursifs ou des piles.

Cette solution n’a malheureusement pas pu etre testee, il serait interessant d’etudier les impacts sur la performance entre les differentes methodes de traversee evoques.

# Conclusion

# Works Cited

**There are no sources in the current document.**