UNIVERSITE DE TECHNOLOGIE DE BELFORT-MONTBELIARD

TO52 – Travail Opérationnel

Construction de BVH sur le GPU pour le calcul de visibilité

Nicolas Said

12/16/2009

Contents

Introduction

Dans le cadre de l'UV TO52 (Travail Opérationnel) propose a l'UTBM, j'ai étudié la réécriture d'une bibliothèque logicielle implémentant le calcul de visibilite dans un champ de vision (Frustum Culling) pour de nombreux champs de vision, et ce en parallèle, en utilisant les capacités des processeurs graphiques.

Un premier travail auquel j'ai participé le semestre dernier dans le cadre de l'UV TX52 m'avait déjà permis de travailler sur le problème. Il est vite apparu que les résultats que nous avons obtenus n'étaient ceux que nous espérions. La méthode utilisée était certes simple, mais avait de bonnes chances d'aboutir à de bons resultats.

Nous avions également propose l'utilisation d'une structure de partitionnement de l'espace afin d'accélérer les opérations de Frustum Culling et d'Occlusion Culling. Ayant manqué de temps, nous n'avons malheureusement pas pu implémenter cette fonctionnalité.

C'est justement l'objet de ce projet. Réécrire la bibliothèque de calcul de visibilite sur GPU afin d'y intégrer la gestion d'une structure de partitionnement de l'espace et d'étudier l'impact de cette modification sur les performances atteintes. La gestion de la structure de partitionnement comporte sa création sur le GPU, ainsi que son utilisation pour l'accélération des opérations de Culling.

Dans une première partie, je ferai quelques rappels sur le travail qui avait été effectue par moi-même et mon binôme dans le cadre de la TX52 du semestre dernier.

Je présenterai ensuite les différentes méthodes de construction de différentes structures de partitionnement de l'espace sur GPU existantes dans la littérature. Puis je présenterai plus en détails celle qui a été choisie pour ce projet, ainsi que les détails de son implémentation.

Enfin, je comparerai les resultats obtenus en utilisant cette nouvelle méthode de calcul de Frustum Culling avec ceux obtenus en utilisant la première méthode.

Je finirai, avant de conclure, par documenter de manière succincte la bibliothèque logicielle qui a été développée dans le cadre de ce projet.

Rappel travail précédent

Lors du semestre précédent, et dans le cadre de l'UV TX52 (« Travaux de Laboratoire »), mon binôme (Arnaud Vallerent) et moi-même avions déjà commencé à travailler sur le sujet. Notre approche était une approche certes naïve, mais qui du point de vue de l'implémentation sous CUDA avait du sens. L'idée était d'effectuer des tests d'inclusions dans les champs de visions pour chaque champs de vision et pour chaque entité de l'univers, et ce en parallèle sur le GPU. Les tests d'inclusion étant indépendants les un des autres, et les frustums pyramidaux ainsi que les entités composant l'univers étant décomposables en primitives géométriques très simples, nous étions persuadés que cette implémentation permettrait de profiter du calcul parallèle de manière optimale.

Malgré cela, les performances furent quelque peu décevantes. Peu être que notre implémentation aurait mérité d'être revue… A la fin de ce projet, il fut décidé d'évaluer l'apport en termes de performances qu'apporterait l'utilisation d'une structure hiérarchique pour stocker les éléments de l'univers. Malheureusement, par manque de temps, ce travail n'a pas pu aboutir. Il fut malgré tout possible d'évaluer les techniques et les structures dont on aurait pu envisager l'utilisation. L'impossibilité d'utiliser des appels récursifs dans les kernels CUDA, nous a mené à envisager d'utiliser des « roped trees » dans lesquels l'ordre de traversée de l'arbre est directement codée dans celui-ci. Une autre solution eut été d'implémenter une pile pour la traversée de la structure hiérarchique. Et enfin, il a fallu déterminer s'il était possible oui ou non de construire une telle structure directement sur le GPU, si cela était, en termes de performance, un choix judicieux, en comparaison avec la construction séquentielle sur CPU.

État de l'art

A ce jour, il existe peu d'algorithmes permettant de générer des structures hiérarchiques sur le GPU. La plupart d'entre eux portent sur la génération de kD-Tree ou de BVH. Notamment (1) et (2). Je crois en fait que ce sont les deux seuls articles assez détaillés qui permettent d'écrire directement une implémentation correcte avec CUDA.

Algorithme de construction

L'algorithme de construction qui a été choisi pour ce projet est l'algorithme LBVH, développé et présenté dans (1). Il permet de réduire le problème de construction de BVH à un problème de tri par base (Radix Sort), en contrepartie d'une perte de contrôle sur la qualité, et l'équilibre de la hiérarchie générée. Il permet de construire rapidement une hiérarchie BVH directement sur le GPU.

Je vais tenter de présenter cet algorithme le plus clairement possible, puis je vais parler de l'implémentation que j'ai programmé avec nVidia CUDA.

Je ne présenterai pas les concepts et notions liées à la programmation CUDA, et je supposerai que le lecteur a déjà l'expérience du développement d'application avec cette technologie.

Présentation

Je vais tenter de vous présenter brièvement le principe de l'algorithme LBVH. LBVH veut dire « Linear Bounding Volume Hierarchy ». L'idée sous-jacente est qu'un algorithme « classique » de construction de BVH du haut vers le bas (« top-down ») ne peut pas tirer profit du calcul parallèle sur GPU, à cause du nombre limité de nœuds ouverts en début de construction. LBVH permet d'effectuer cette construction de manière parallèle.

Comme évoqué précédemment, LBVH réduit le problème de construction à un problème de tri. On dispose aujourd'hui d'excellents algorithmes de tri pour architecture parallèle et notamment sur GPU.

Dans un premier temps, toutes les primitives (dans notre cas présent, des AABB ou des OBB) sont placées sur une courbe fractale continue, une courbe de Lebesgue. Dans le cadre de ce projet, nous n'étudierons que le cas d'une courbe bidimensionnelle inscrite dans un plan de l'espace (afin de simplifier les explications).

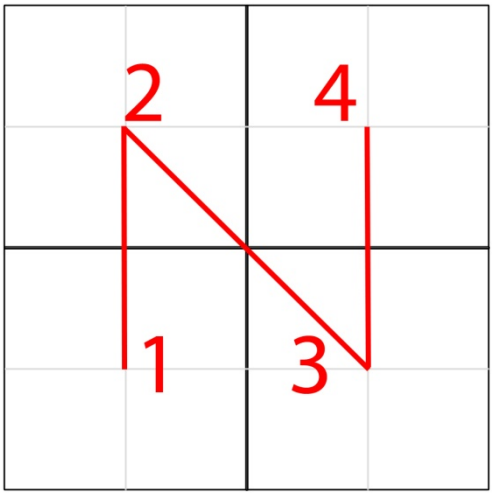

Figure 1

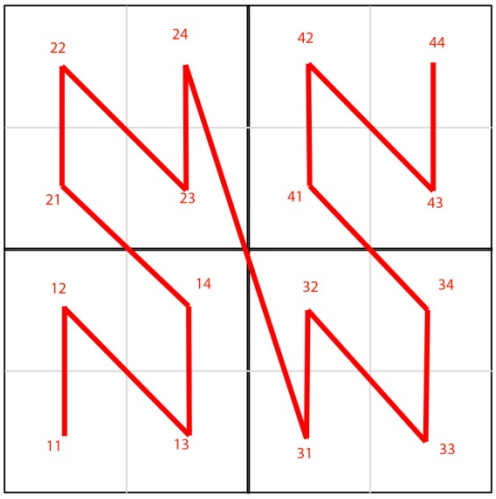

Figure 2

A chaque portion de l'espace traversée par la courbe correspond un code. Ainsi chaque zone de l'espace peut être subdivisé et identifié par un code unique.

Prenons un cas d'exemple pour illustrer le principe de linéarisation de l'univers. La première étape de linéarisation consiste à assigner à chaque primitive composant l'univers, un code permettant de la situer sur la courbe.

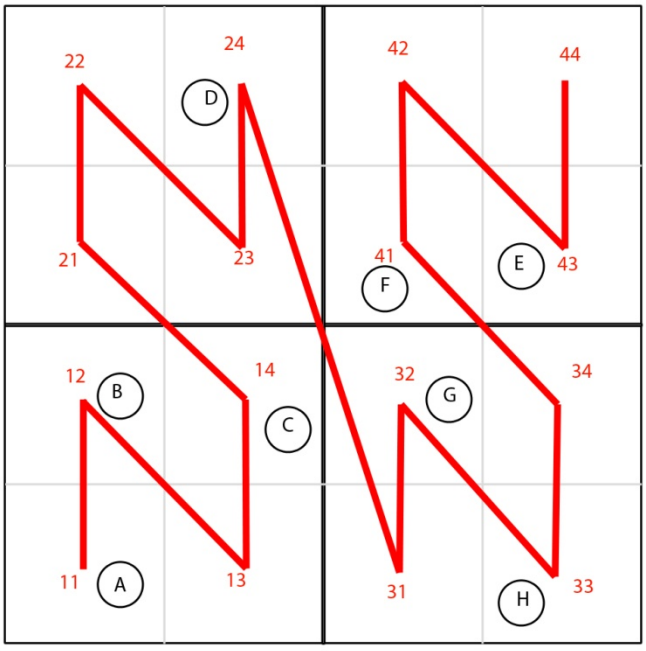

Figure 3

Assignons maintenant les codes à nos primitives :

Primitive	Code niveau 1	Code niveau 2
A	1	1
B	1	2
C	1	4
D	2	4
E	4	3
F	4	1
G	3	2
H	3	3

Figure 4

La seconde étape consiste à ordonner les primitives sur la courbe. Cette étape se réduit à un tri par base sur les codes. Cette étape nous donne la nouvelle table de primitives triées :

Primitive	Code
A	11
B	12
C	14
D	24
G	32
H	33
F	41
E	43

Figure 5

Par suite, en comparant deux à deux les codes des primitives, il est possible de connaitre les indices dans la liste des primitives des « splits », c'est-à-dire, les endroits dans la liste où deux primitives n'appartiennent plus à la même région de l'espace. Si les codes diffèrent en position i, alors cela veut dire que les deux primitives ne sont pas dans les mêmes régions de l'espace pour toutes les subdivisions de niveau supérieures.

Cela revient à regrouper les primitives par « paquets » :

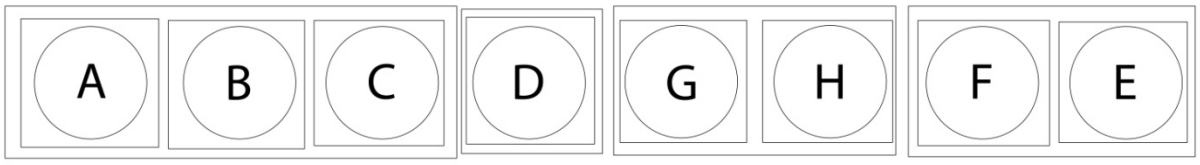

Figure 6

Continuons notre exemple et construisons notre liste de « splits » pour notre univers :

Emplacement du split	Niveau du split
0	1
0	2
1	2
2	2
3	1
3	2
4	1
4	2
5	2
6	1
6	2
7	2
8	1
8	2

Figure 7

Si nous trions notre liste de splits par niveau de split, nous obtenons ceci :

Emplacement du split	Niveau du split
0	1
3	1
4	1
6	1
8	1
0	2
1	2
2	2
3	2
4	2
5	2
6	2
7	2
8	2

Figure 8

A partir de cette liste, en comparant les splits deux à deux, il est possible de connaitre les intervalles sur la liste de primitives, composant notre hiérarchie :

Nœud	Niveau	Intervalle de primitives
0	1	[0 ;3[
1	1	[3 ;4[
2	1	[4 ;6[
3	1	[6 ;8[
4	2	[0 ;1[
5	2	[1 ;2[
6	2	[2 ;3[
7	2	[3 ;4[
8	2	[4 ;5[
9	2	[5 ;6[
10	2	[6 ;7[
11	2	[7 ;8[

Figure 9

Après cette opération, nous avons donc bien les informations nécessaires sur les différents regroupements de primitives composant la hiérarchie.

Il ne reste plus qu'à rendre ces informations utilisables pour un parcours en profondeur, en calculant les indices des nœuds fils, pour chaque nœud, et de calculer les boites englobantes des nœuds, en utilisant une réduction du bas vers le haut (« bottom-up »). Ces deux étapes seront expliquées plus en détails dans le prochain chapitre.

Implémentation

Mon implémentation de l'algorithme de construction LBVH comprend 7 étapes différentes.

Étape 1

La première étape consiste à assigner à chaque primitive un code sur la courbe de Lebesgue. Le kernel associé à cette opération a une complexité $O(\ d\ )$ où d est la profondeur maximale du BVH. Cette étape génère une liste de primitive comportant un champ supplémentaire pour stocker le code (structure **`bvhnode_t`**)

Étape 2

La deuxième étape consiste à trier notre liste de bvhnode_t selon leur code. Cette étape utilise une fonction de tri par base mise à disposition par la bibliothèque logicielle « thrust » (3), qui propose différentes structures de données et algorithmes implémentés avec CUDA.

Étape 3

Cette étape consiste à calculer la liste de splits, tel qu'il a été expliqué précédemment. Cette opération génère une liste de structure **`lbvhsplit_t`**. Le kernel associé à cette opération a une complexité $O(\ d\ )$ où d est la profondeur maximale du BVH.

Étape 4

La quatrième étape consiste à trier la liste de split par niveau de split. Cette opération utilise également un algorithme de tri pour CUDA disponible dans la bibliothèque « thrust ».

Étape 5

Cette étape (avec l'étape 6) est la première étape de construction de la structure hiérarchique à proprement parler. Une structure explicitant la hiérarchie, et directement utilisable pour un parcours ultérieur. Elle génère une liste de structure **`hnode_t`**, structure contenant les informations de nœuds de la hiérarchie.

L'étape 5 commence par calculer les intervalles sur la liste des primitives pour chaque nœud. Le kernel associé à cette étape a une complexité $O(1)$.

Étape 6

L'étape 6 va calculer les intervalles sur la liste de nœuds, donnant pour chaque nœud de la hiérarchie, la liste de ses enfants.

Dans un premier temps, nous allons trier la liste de nœuds par le début de leur intervalle sur la liste de primitives. En comparant deux à deux la borne inférieure sur l'intervalle des primitives, nous pouvons déduire la borne inférieure des intervalles des enfants. Si leurs bornes inférieures sur l'intervalle des primitives sont égales, et que la différence entre leur niveau de split est égale à 1, alors nous pouvons calculer la borne inférieure du nœud supérieur. Regardons l'exemple :

Nœud	Niveau	Intervalle de primitives	Intervalle d'enfants
0	1	[0 ;3[	[4 ; ??]
4	2	[0 ;1[	
5	2	[1 ;2[	
6	2	[2 ;3[	
1	1	[3 ;4[	[7 ; ??]
7	2	[3 ;4[	
2	1	[4 ;6[	[8 ;??]
8	2	[4 ;5[	
9	2	[5 ;6[	
3	1	[6 ;8[	[10 ;??]
10	2	[6 ;7[	
11	2	[7 ;8[	

Figure 10

Par la suite, nous retrions la liste en fonction de la borne supérieure des intervalles sur les primitives, puis en déduisons les bornes supérieures des intervalles d'enfants pour chaque nœud de niveau inférieur au niveau maximal (les branches). Cette opération est similaire à la précédente.

Si nous reprenons notre exemple, nous obtenons :

Nœud	Niveau	Intervalle de primitives	Intervalle d'enfants
4	2	[0 ;1[	
5	2	[1 ;2[	
0	1	[0 ;3[	[4 ; 6]
6	2	[2 ;3[	
1	1	[3 ;4[	[7 ; 7]
7	2	[3 ;4[	
8	2	[4 ;5[	
2	1	[4 ;6[	[8 ;9]
9	2	[5 ;6[	
10	2	[6 ;7[	
3	1	[6 ;8[	[10 ;11]
11	2	[7 ;8[	

Figure 11

Enfin, un dernier tri est effectué sur les identifiants des nœuds, de manière à rendre la structure utilisable pour un parcours.

Nœud	Niveau	Intervalle de primitives	Intervalle d'enfants
0	1	[0 ;3[	[4 ; 6]
1	1	[3 ;4[	[7 ; 7]
2	1	[4 ;6[	[8 ;9]
3	1	[6 ;8[	[10 ;11]
4	2	[0 ;1[	
5	2	[1 ;2[	
6	2	[2 ;3[	
7	2	[3 ;4[	
8	2	[4 ;5[	
9	2	[5 ;6[	
10	2	[6 ;7[	
11	2	[7 ;8[	

Figure 12

Étape 7

La septième et dernière étape consiste à calculer pour chaque nœud sa boite englobante. Il s'agit d'une réduction bottom-up. Après cette opération, la hiérarchie peut être utilisée pour un parcours.

Algorithme de Frustum Culling

Présentation

L'algorithme de Frustum Culling que j'ai implémenté utilise la structure hiérarchique du BVH pour éviter le plus de traitement inutile que possible. Il s'agit d'un algorithme récursif assez simple. Tout d'abord, nous chargeons le premier niveau de la hiérarchie. Pour chaque nœud, nous testons l'inclusion de sa boite englobante dans le frustum. Si elle est en intersection ou incluse, alors nous testons ses fils. Si le nœud courant est un nœud de niveau maximal, alors nous testons chacune des primitives qui lui sont associées.

Le test d'inclusion qui est utilisé est une version très basique (pas d'optimisation superflue, car cela entrainerait le nombre de branches divergentes lors de l'exécution du kernel) :

```
// tests if a AaBox is within the frustrum
int Frustrum::ContainsAaBox(const AaBox& refBox) const
{
	Vector3f vCorner[8];
	int iTotalIn = 0;

	// get the corners of the box into the vCorner array
	refBox.GetVertices(vCorner);

	// test all 8 corners against the 6 sides
	// if all points are behind 1 specific plane, we are out
	// if we are in with all points, then we are fully in
	for(int p = 0; p < 6; ++p) {

		int iInCount = 8;
		int iPtIn = 1;

		for(int i = 0; i < 8; ++i) {

			// test this point against the planes
			if(m_plane[p].SideOfPlane(vCorner[i]) == BEHIND) {
			iPtIn = 0;
				--iInCount;
			}
		}

		// were all the points outside of plane p?
		If(iInCount == 0)
			return(OUT);

		// check if they were all on the right side of the plane
		iTotalIn += iPtIn;
	}

	// so if iTotalIn is 6, then all are inside the view
	if(iTotalIn == 6)
		return(IN);

	// we must be partly in then otherwise
	return(INTERSECT);
}
```

Implémentation

L'implémentation du frustum culling, avec utilisation du BVH a posé quelques problèmes. Premièrement, les kernels CUDA ne peuvent pas utiliser d'appels récursifs. J'ai donc du implémenter moi-même ma propre structure de pile, pour pouvoir parcourir en profondeur le BVH.

Cette pile est stockée en mémoire locale (ce qui s'avère être le meilleur choix, en ce qui concerne les performances, en comparaison avec les implémentations en mémoire partagée et globale). Ce choix limite le nombre de thread par bloc pour le frustum culling, mais cela n'est pas très grave.

Cette version de l'algorithme de culling sous CUDA reprend la structure de sortie qui avait été adoptée pour la première version. Un tableau de $m \times n$ valeurs permet de stocker les résultats des test d'inclusion des n différentes primitives dans les m différents frustums.

Autre point important à noter. La mémoire texture permet d'accélérer les temps d'accès aux données, lorsque ceux-ci sont effectués d'une manière aléatoire, et c'est le cas pour notre application. Ainsi, j'ai décidé de tester l'accès aux informations des nœuds à partir de la mémoire texture sous CUDA. J'ai noté une légère amélioration des performances, mais rien de majeur.

Dans tout les cas, c'est une technique d'accès aux données de hiérarchie très courante dans les différents projets de raytracing que j'ai pu voir sur internet. Même si le raytracing et le calcul de visibilité son assez éloignés, pour ce qui est des implémentations de la traversée d'une structure hiérarchique sous CUDA, le code est très similaire.

Résultats

Construction de la hiérarchie

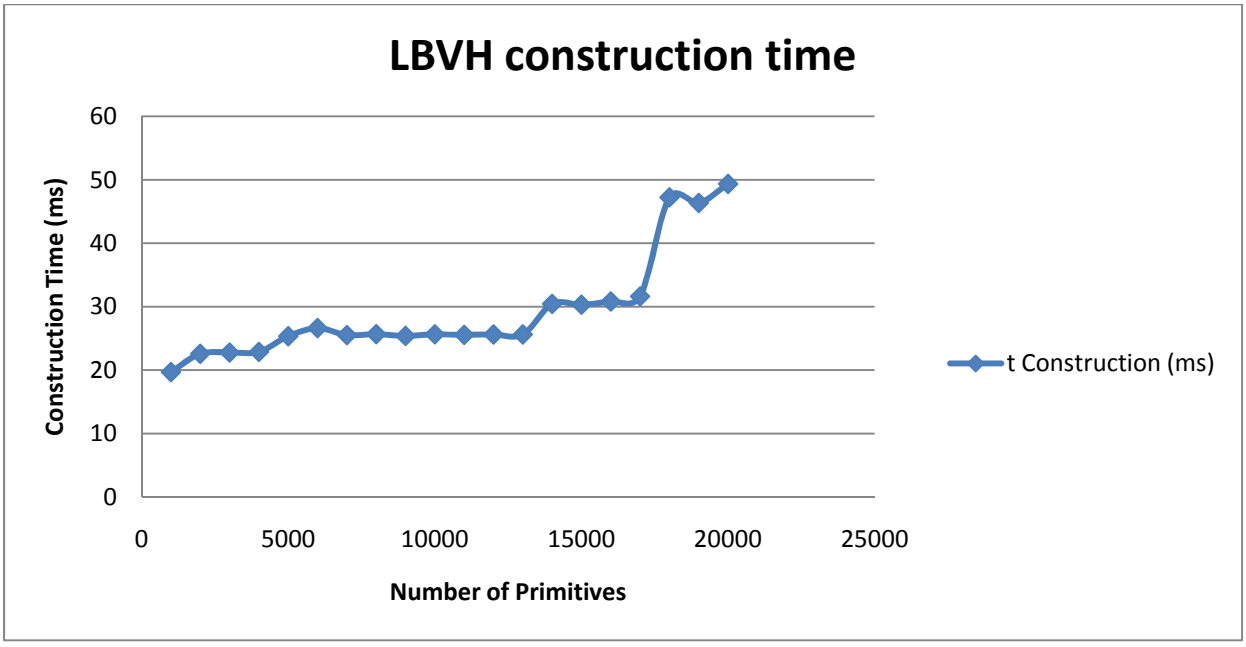

Figure 13

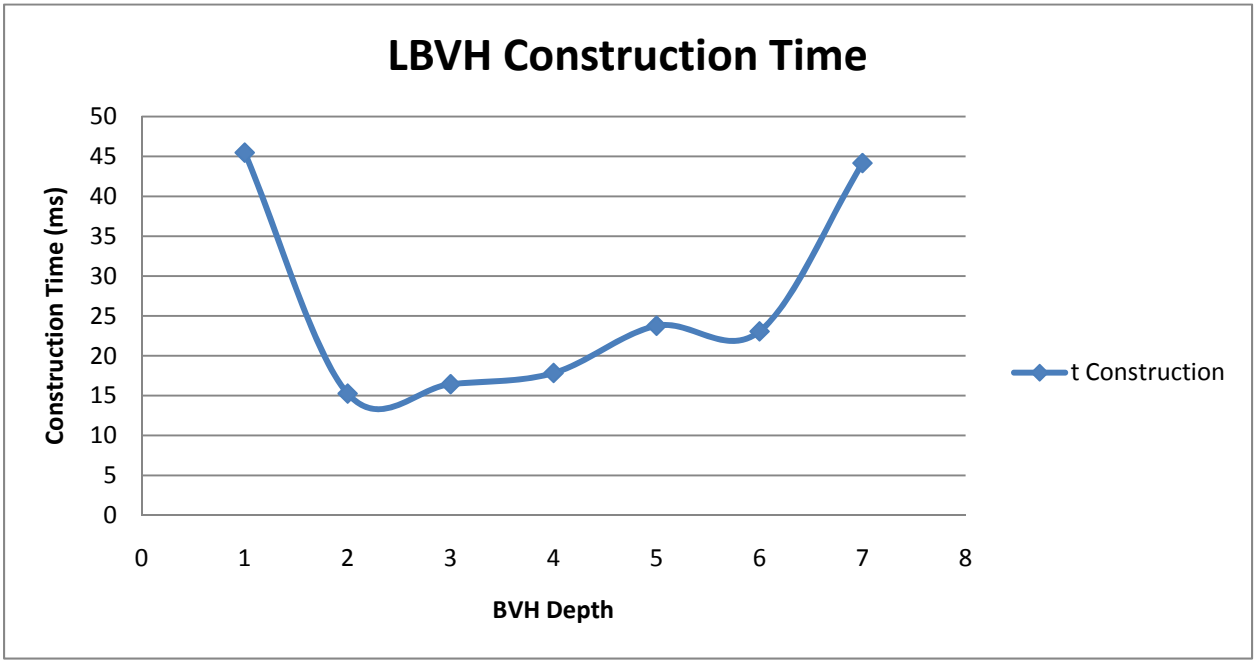

Figure 14

La forme singulière de cloche que l'on peut observer peut être expliquée de la manière suivante : Pour un nombre fixé de primitives, lorsque la profondeur du LBVH n'est pas assez importante, le kernel de calcul de boite englobante (BVH Refit) sont exécutés avec très peu de threads effectuant d'importants parcours linéaires sur la liste des primitives. Un phénomène similaire se produit lorsque la profondeur est importante, chaque étape du calcul de boite englobante pour un niveau étant dépendant de celui du niveau précédent, il faut lancer n fois le kernel successivement, ce qui induit un cout en termes de temps…

Calcul de visibilité

Afin de tester l'algorithme de Frustum Culling, l'application de test propose de générer des hiérarchies à partir de différents types de distributions spatiales.

Les mesures portant le libellé « t Bruteforce » font référence aux temps d'éxecution obtenus en utilisant la précédente version de gpuCuller.

Les mesures portant le libellé « t BVH » font référence aux temps d'éxecution obtenus en utilisant la structure hiérarchique.

Distribution Uniforme

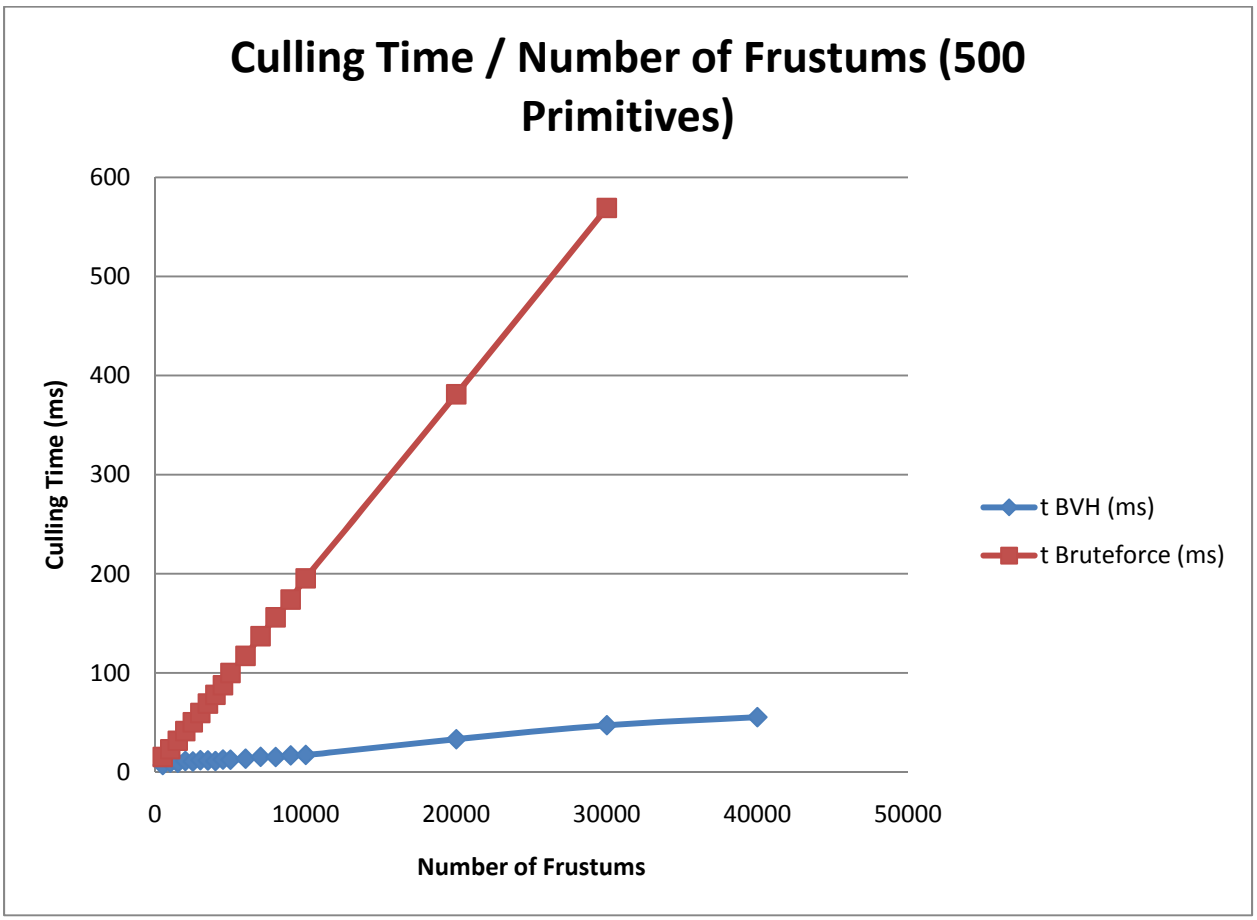

Figure 15

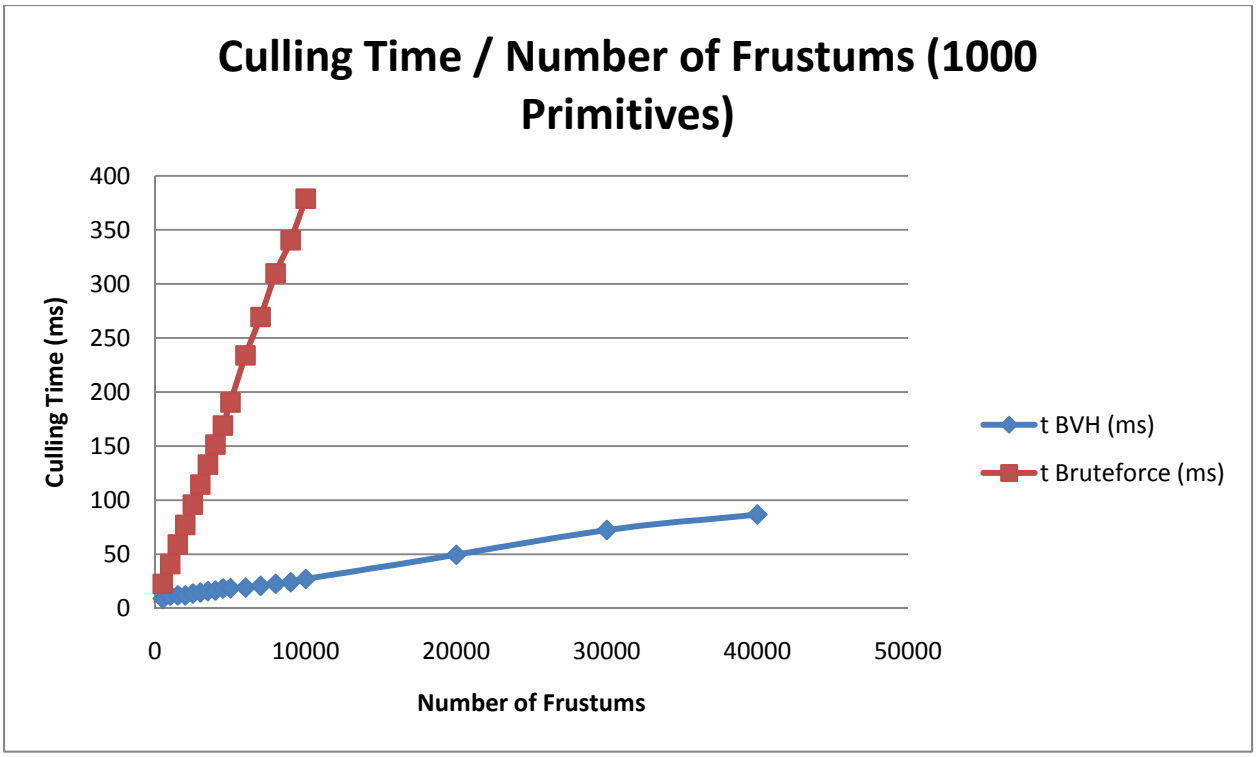

Figure 16

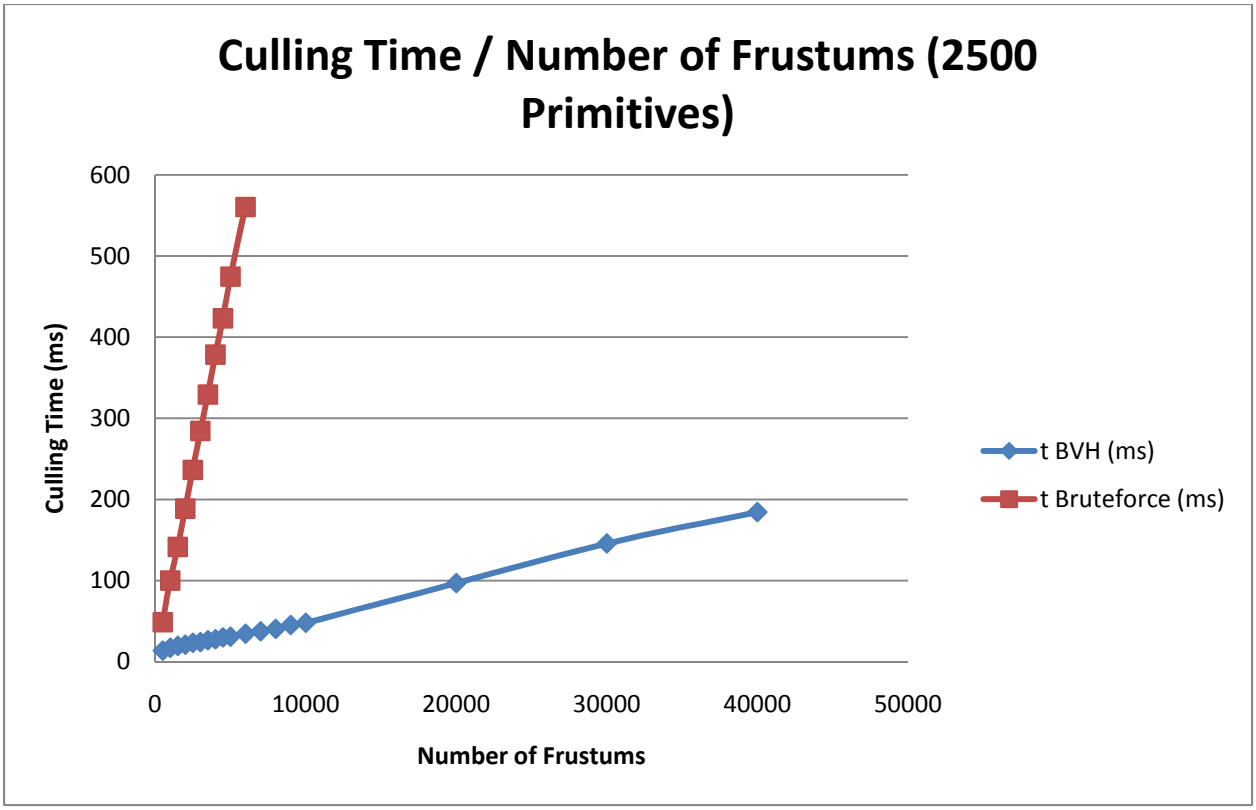

Figure 17

Distribution avec amas

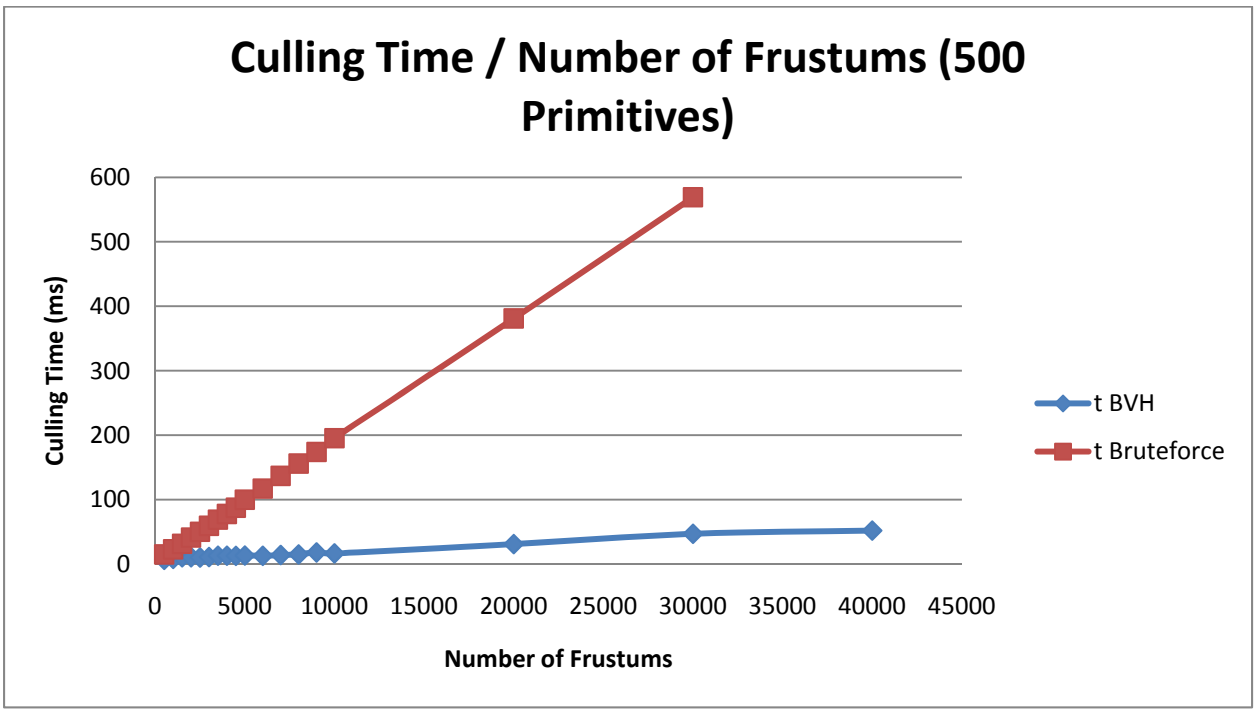

Figure 18

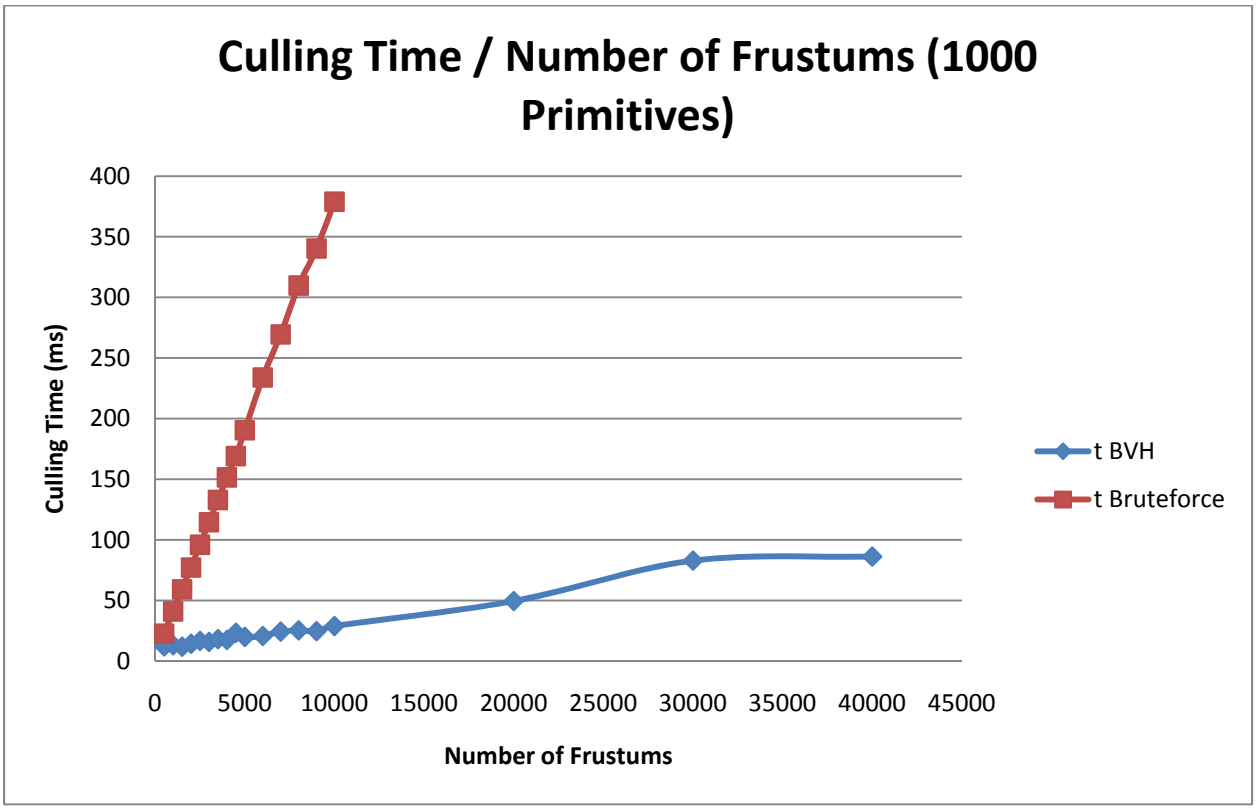

Figure 19

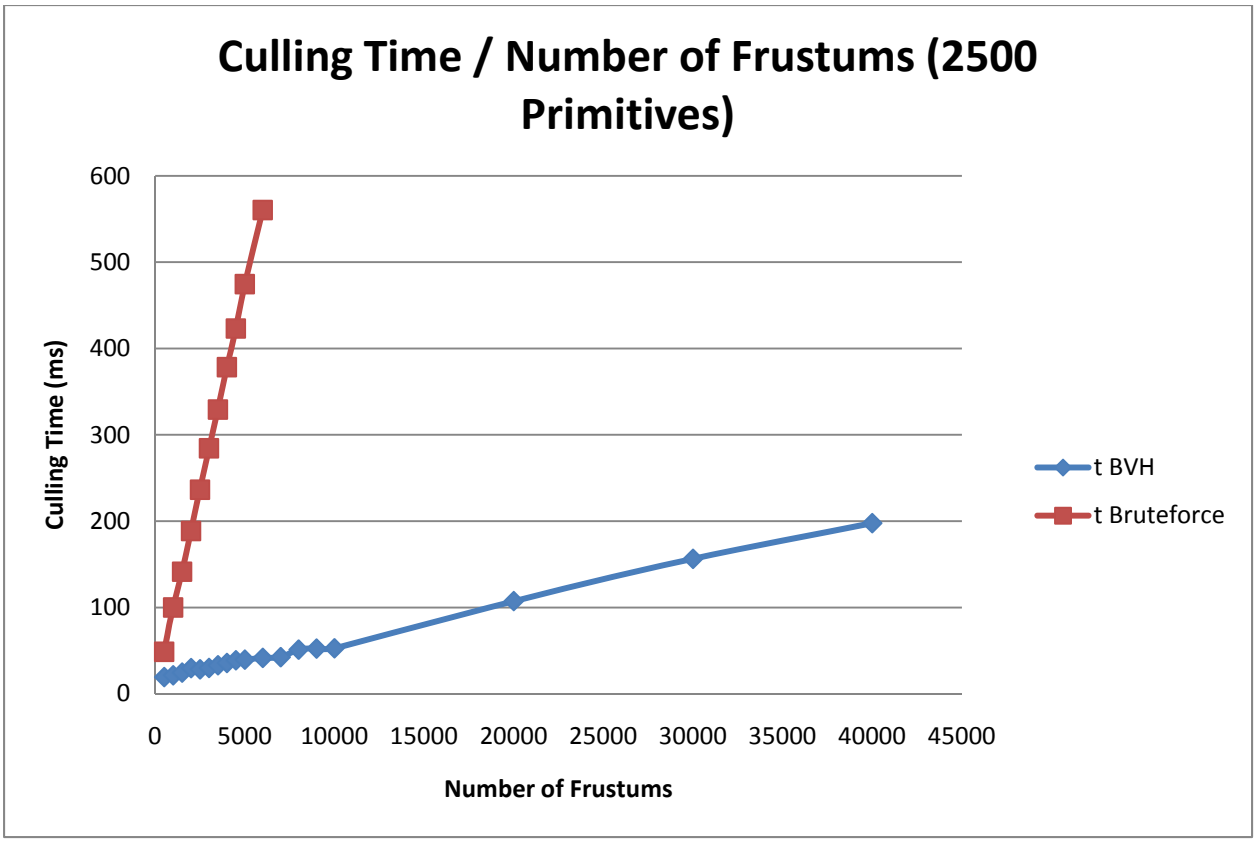
Culling Time / Number of Frustums (2500 Primitives)
Culling Time (ms)
600
500
400
300
200
100
0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
Number of Frustums
t BVH
t Bruteforce

Figure 20

Distribution uniforme avec amas

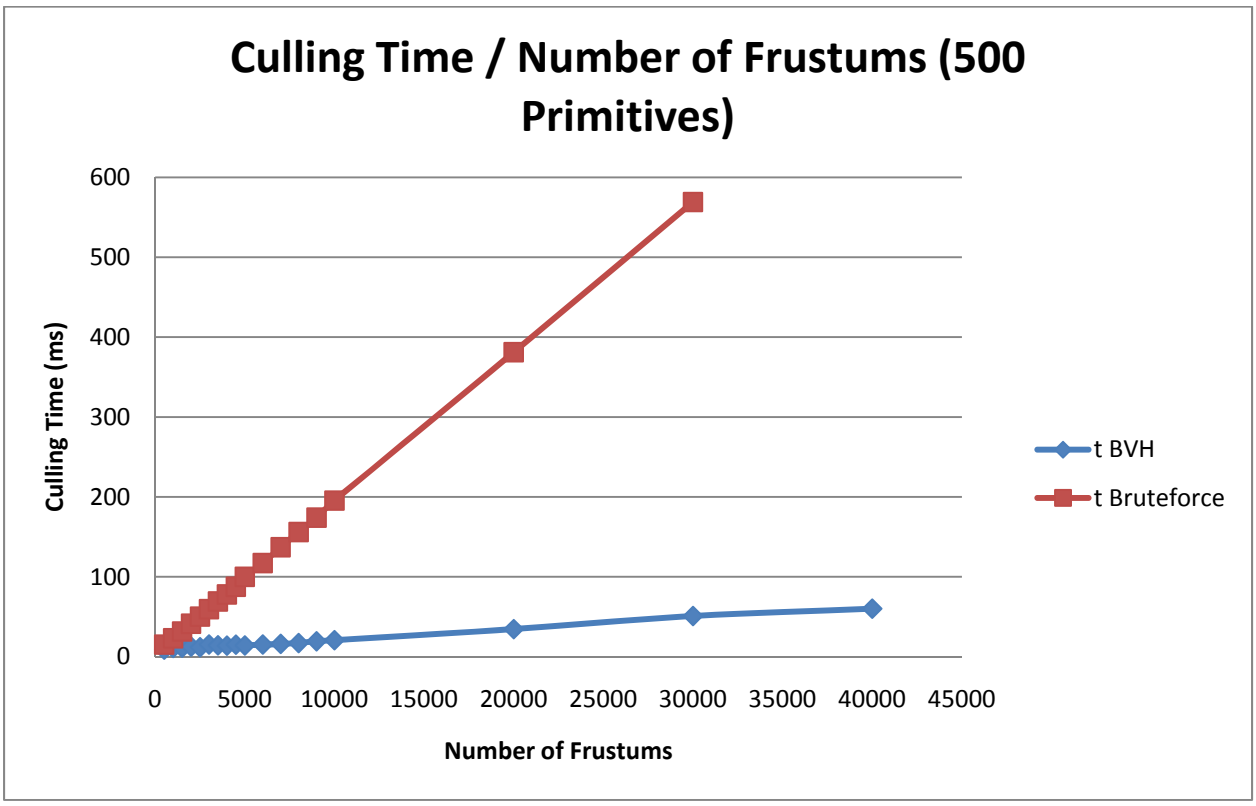

Figure 21

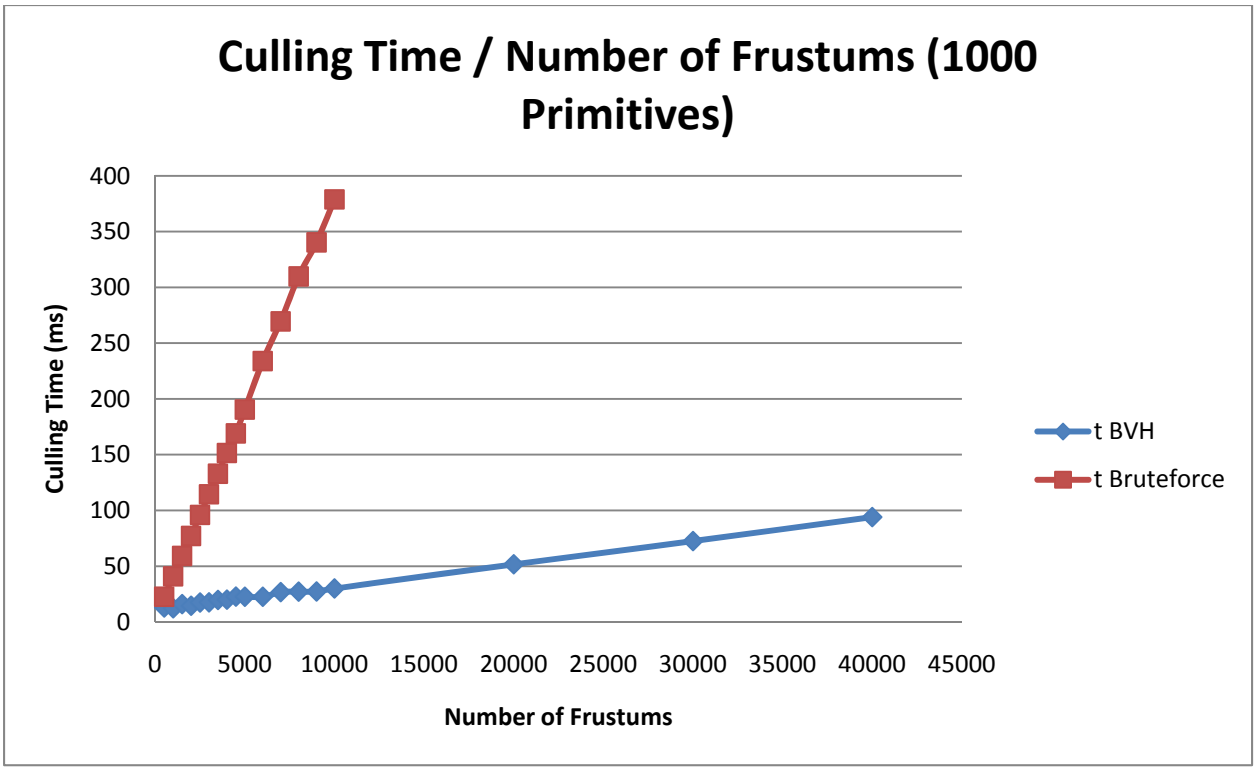

Figure 22

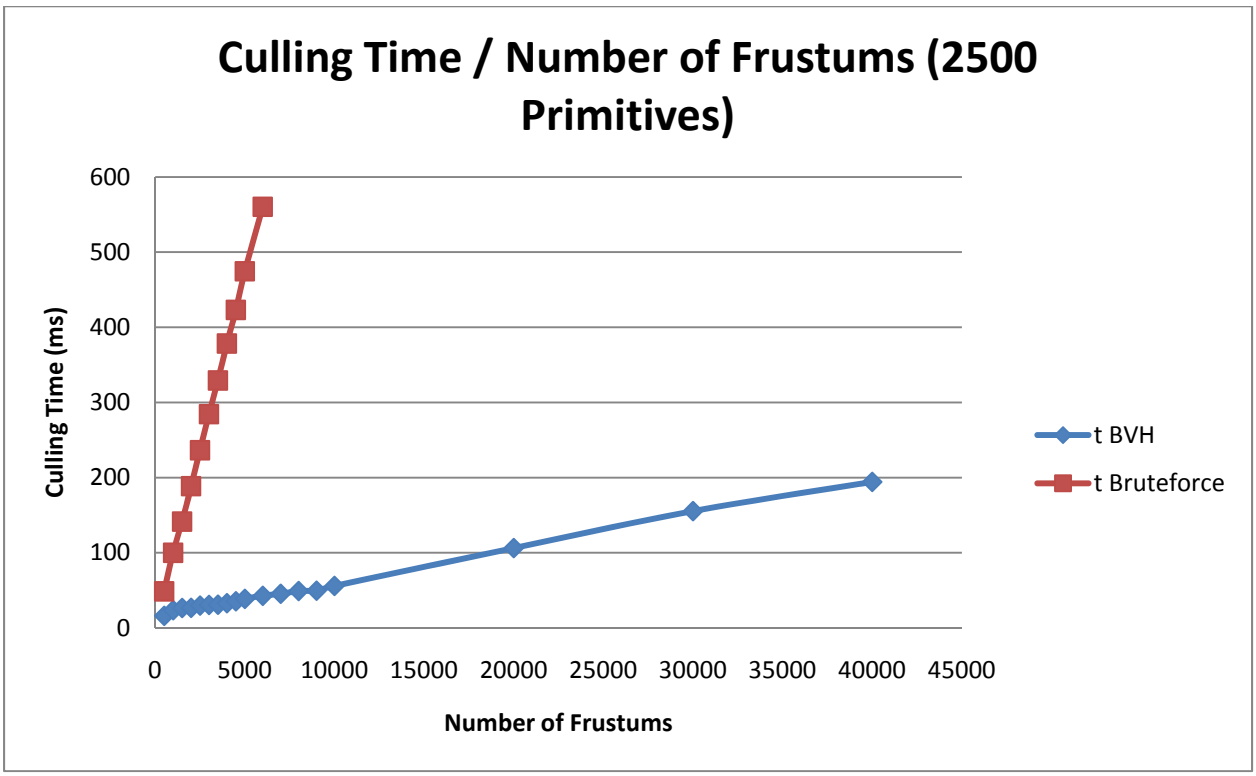

Figure 23

Approfondissements

Tel qu'il est proposé dans l'article sur lequel je me suis basé pour réalisé ce projet, il est possible d'associer une première étape de construction LBVH avec des étapes de construction permettant de générer des hiérarchie plus performantes et plus équilibrées, basées, par exemple sur SAH (Surface Area Heuristic) ou autres.

De la même manière, il aurait été intéressant d'étudier la faisabilité d'une modification de la hiérarchie sur le GPU, permettant de supporter des environnements dynamiques. Possibilité par ailleurs évoquée dans (3).

Et enfin, dernièrement, il aurait été sympathique d'implémenter d'entrée de jeu le support pour différent type de primitives (AABB, OBB, Sphères…) et pas seulement les AABBs.

Ces différents travaux pourront faire l'objet d'un nouveau projet à l'UTBM, qui sait ?

Documentation

Voilà la documentation de la bibliothèque logicielle développée dans le cadre de ce projet, et qui implémente la construction du BVH et le calcul de visibilité.

`void gculInitialize( int argc, char** argv )`

Cette fonction permet d'initialiser la bibliothèque gpuCuller. Elle prend en arguments les informations concernant les paramètres de ligne de commande de l'application hôte.

`void  gculLoadAABB( unsigned int size, const void* ptr );`

Cette fonction charge en mémoire GPU les informations concernant les primitives (à savoir, des AABB). Le premier paramètre est le nombre d'AABB à charger, et le deuxième est le pointeur vers la zone de mémoire où sont stockées les AABB. Les AABB au sein de gpuCuller sont composées de 6 float (3 pour le point minimal et 3 pour le point maximal). Ne pas oublier que le repère utilisé au sein de la bibliothèque est le repère OpenGL.

`void  gculLoadFrustumPlanes( unsigned int size, const void* ptr );`

Cette fonction charge en mémoire GPU les informations concernant les plans définissant les Frustum pyramidaux. Le premier paramètre est le nombre de Frustum. Le deuxième paramètre est le pointeur vers la zone mémoire où sont stockées les informations. A savoir que chaque Frustum est composé de 6 plans, et chaque plan est composé de 4 valeurs float (normale + distance à l'origine). Les normales aux plans sont considérées comme orientées vers l'extérieur du volume.

`void  gculFreeAABB();`

Permet de libérer la mémoire GPU allouée aux AABB

```
void  gculFreeFrustumPlanes();
```

Permet de libérer la mémoire GPU allouée aux Frustum.

```
void  gculBuildHierarchy();
```

Lance la construction du BVH

```
void  gculFreeHierarchy();
```

Permet de libérer la mémoire allouée à la hiérarchie générée.

```
void  gculProcessCulling();
```

Lance le calcul de visibilité.

```
void  gculSetBVHDepth( unsigned int depth );
```

Permet de fixer la profondeur maximale du BVH à générer.

```
void  gculSetUniverseAABB( float min_x, float min_y, float min_z,
float max_x, float max_y, float max_z );
```

Permet de fixer les dimensions de la boite englobante globale de l'univers.

```
unsigned int  gculGetHierarchySize();
```

Retourne la taille du tableau contenant les informations de hiérarchie.

```
void  gculGetHierarchyInformation( void* data );
```

Permet d'extraire les informations de hiérarchie, pour d'éventuelles opérations de débogage.

```
void  gculGetResults(void* data);
```

Permet d'extraire les résultats du calcul de visibilité. Il s'agit d'un tableau de char de taille m par n, où m est le nombre de Frustum et n est le nombre de primitives.

```
void  gculSaveHierarchyGraph(char* outputFile);
```

Cette fonction permet de générer un fichier .dot permettant de visualiser la hiérarchie générée.

Dépôt Subversion

Pour avoir plus d'informations sur le projet, ou bien consulter les différents programmes d'exemple d'utilisation de la bibliothèque logicielle gpuCuller, je vous invite à visiter le dépôt SVN qui est disponible à l'adresse suivante :

http://code.google.com/p/gpufrustum/

Conclusion

Ce projet se trouvait être dans la directe continuité du travail que j'avais pu effectuer lors du semestre précédent. J'ai pu approfondir mes connaissances concernant le développement d'application massivement parallèles sur GPU avec la technologie nVidia CUDA.

Ca a été l'occasion pour moi d'implémenter par moi-même un algorithme issu de la littérature scientifique, algorithme a priori, pas trivialement parallelisable.

J'en profite par ailleurs pour remercier Michael Garland, auteur de l'article sur lequel est basé ce projet (1), pour m'avoir aidé à implémenter les dernières phases de construction du BVH.

Bibliographie

1. *Fast BVH Construction on GPUs.* **Lauterbach, Christian, et al.**

2. *Real-Time KD-Tree Construction on Graphics Hardware.* **Zhou, Kun, et al.** 2008.

3. *gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries.* **Lauterbach, Christian, Mo, Qi et Manocha, Dinesh.** 2009.

4. *Thrust: A Parallel Template Library.* **Hoberock, Jared et Bell, Nathan.** 2009.