Université de Liège
Institut Montefiore

Département d'électricité, électronique et informatique

Projet d'infographie - Sujet 3

Pré-détermination des objets impactés par un rayon (accélération du rendu)

par

Deneux Thomas
Joniaux Gilles

Liège
Année académique 2009-2010

Dernière modification: 11 mai 2010, 0:07

Table des matières

Section 1

Présentation

Notre sujet pour ce projet d'infographie est la pré-détermination des objets impactés par un rayon. Ainsi, nous allons pouvoir accélérer le rendu réaliste d'une scène. Plusieurs questions viennent alors directement à l'esprit : comment cette accélération est-elle possible ? Que signifie la pré-détermination des objets impactés par un rayon ? Quels algorithmes peuvent être mis en oeuvre à cette fin ?

Réfléchissons tout d'abord à ce qu'il se passe lors d'un lancer de rayon. Nous avons un rayon pour chaque pixel de l'écran, un rayon d'ombre pour chaque source lumineuse, un rayon réfléchi, et éventuellement un rayon réfracté, par intersection. La quantité de calcul pour déterminer le rendu est fort élevée.

L'accélération de ce rendu peut-être possible en réduisant le nombre d'opérations pour chacun de ces éléments. Ainsi, nous allons vous présenter brièvement les différentes méthodes qui permettent cette optimisation et qui peuvent prédire si un ensemble d'objets risque d'être touché ou non par un rayon.

Section 2

Classification des méthodes d'accélération

Les différentes méthodes dont nous vous parlions peuvent être classées en plusieurs groupes. En effet, le nombre important de rayons et d'intersections gonfle le temps de calcul. Deux classes d'algorithmes sont donc présentes :

Moins de rayons : Diminue le nombre de rayons qui vont être l'objet d'un traitement.

Calcul d'intersections rapide : Détermine rapidement si un objet va être touché ou non par un rayon.

2.1 Moins de rayons

Nous ne nous attarderons pas sur ce principe. Voici certaines méthodes existantes regroupées en deux catégories :

- Profondeur adaptive, sous-échantillonage.
- Rayons généralisés : cônes, faisceaux, etc.

Nous n'avons pas choisi ce type d'implémentation parce qu'il semble moins adapté au canevas développé en groupe dans le cadre de ce projet.

2.2 Calcul d'intersections rapide

Pour ce principe, les algorithmes existants peuvent également être divisés en deux catégories distinctes :

Intersections optimisées : Primitives simples, calculs optimum, calculs parallèles.

Moins d'intersections : Multirésolution, structures spatiales, structures hiérarchiques.

Section 3

Notre choix

Les deux catégories qui ont le plus retenu notre attention sont les structures spatiales et les structures hiérarchiques. Pour la première d'entre elle, nous avons étudié plus attentivement les principes de grille régulière et de KD-tree. Pour la seconde, nous avons surtout découvert la méthode des volumes englobants.

3.1 Grille régulière

L'objectif de cette approche est de diminuer le nombre d'intersections entre les rayons et les objets. Pour cela, cette méthode va accélérer la détection des intersections.

Le principe de cette méthode est de diviser de manière régulière l'espace. Ainsi, chaque élément de la scène sera stocké dans un tableau à l'endroit représentant le sous-espace dans lequel il se situe. La diminution du temps de calcul vient du fait que les algorithmes de tracé de droites sont plus efficaces et que seules les intersections potentielles sont calculées.

Cette méthode est assez simple à mettre en oeuvre, mais son coût de stockage et sa sensibilité aux différences d'échelle la rendent moins performante.

3.2 KD-tree

Cette méthode est la plus performante. En effet, son objectif est également de diminuer le nombre d'intersections entre les rayons et les objets, aussi bien en accélérant la détection des intersections que celle des non-intersections.

Son principes est de diviser l'espace de manière irrégulière cette fois. Les objets sont alors regroupés dans un arbre. Ce dernier est alors équilibré en fonction du temps de la traversée du rayon dans les différents sous-espaces.

Cette méthode apporte les meilleurs résultats, mais elle est extrêmement difficile à implémenter. Le coût de programmation que demande le KD-tree nous semble trop important que pour le choisir.

3.3 Les volumes englobants

Nous avons donc décidé d'implémenter cette méthode dans le cadre de ce sujet. Nous allons donc vous exposer clairement son fonctionnement, déterminer le temps de calcul que nous pouvons gagner et expliquer quelque peu son implémenation.

Section 4

Les volumes englobants

4.1 Objectif

Comme pour la grille régulière et pour le KD-tree, l'objectif est de diminuer le nombre d'intersections entre les rayons et les objets. Cette fois, c'est la détection des non-intersections qui va être accélérée.

4.2 Principe

Le principe de cet algorithme est de créer une hiérarchie de volumes englobants. Ainsi, chaque élément va être encadré par une boîte. Les boîtes les plus proches vont alors être regroupées par deux et englobées par un nouveau volume. Nous allons donc créer un arbre binaire dont la racine contient la boîte englobant tous les éléments et dont les fils à gauche et à droite de chaque noeud représentent les deux boîtes englobées par celle parente, comme nous le montre ce schéma :

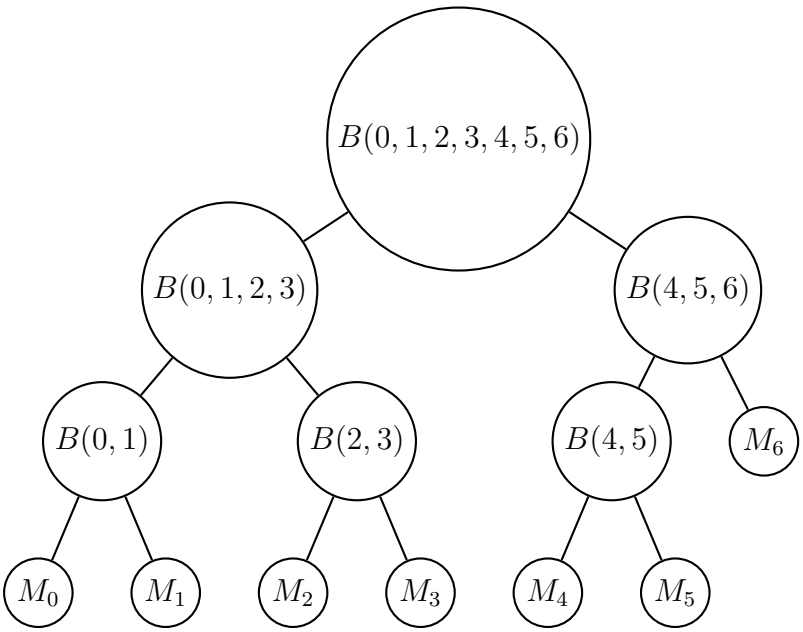

FIGURE 1 – Arbre binaire

Dans la FIGURE 1, $B(i)$ représente une boîte englobant tous les modèles M_i.

Cette méthode nous permet de gagner du temps de deux façons :

– Lors du parcours de l'arbre, si un rayon n'intersecte pas un volume englobant, nous sommes certains que ce rayon n'intersectera aucun élément interne à

ce volume (nous stoppons alors le parcours pour le fils à gauche et le fils à droite de cette boîte). Nous pouvons donc éliminer un bon nombre de calculs d'intersections inutiles.
– Le deuxième gain de temps vient du fait qu'il est beaucoup plus simple de calculer les intersections avec ces volumes englobants qu'avec des éléments plus complexes.

4.3 Algorithme

La première étape de l'algorithme est l'initialisation de l'arbre. Deux méthodes peuvent être utilisées à cette fin. Soit, nous procédons par regroupement des volumes englobants, soit, par séparation. Après quelques petites réflexions, nous avons décidé d'implémenter la première de ces deux méthodes. En effet, nous préférons créer pour chaque élément un volume englobant qui l'encadre en perdant le moins d'espace vide possible (voir FIGURE 2 et FIGURE 3) et ensuite, les regrouper par deux et par volumes les plus proches afin de remonter dans la hiérarchie et de construire l'arbre final.

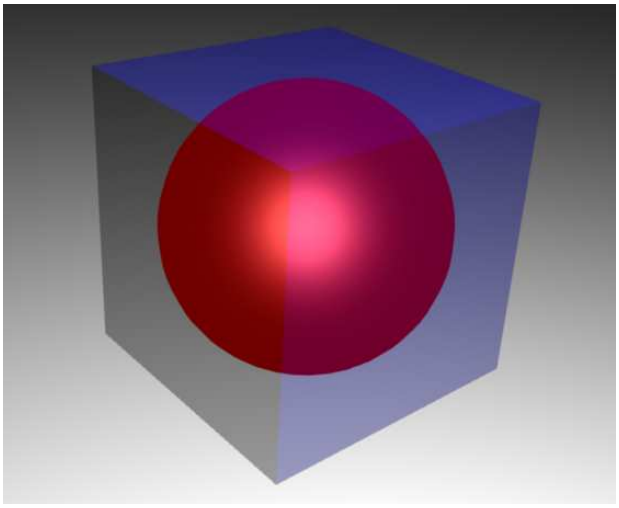

FIGURE 2 – Boîte englobante pour la sphère

La deuxième étape consiste à parcourir l'arbre créé, et ce pour chaque rayon. Ainsi, pour chaque noeud de cet arbre, nous allons tester si le rayon courant intersecte le volume englobant représenté par ce dernier ou non. Si c'est le cas, nous continuons le parcours de l'arbre pour le fils à gauche et le fils à droite de celui-ci. Si ce n'est pas le cas, nous stoppons ce parcours. Lorsque nous atteignons une feuille de l'arbre, nous ajoutons l'élément situé dans le volume englobant à la liste des éléments susceptibles d'être intersectés par le rayon courant.

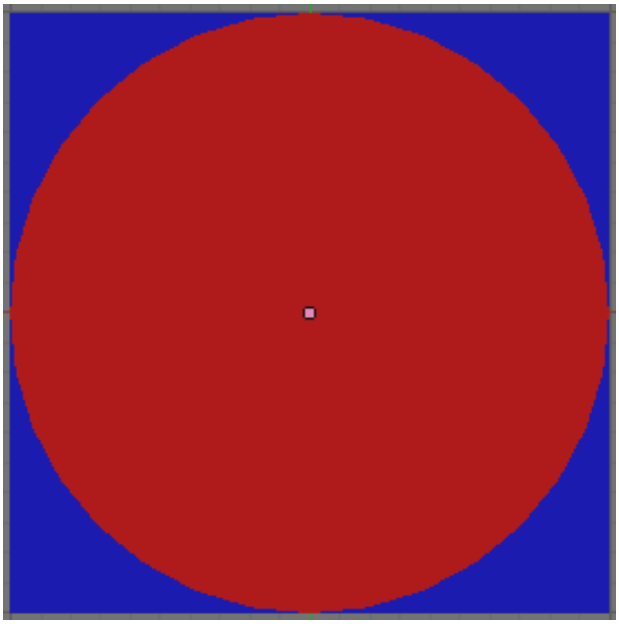

FIGURE 3 – Boîte englobante pour la sphère - Projection par rapport à X, Y ou Z

La troisième étape consiste à remplacer le parcours sur l'ensemble de tous les éléments de la scène présent avant notre optimisation par un parcours sur l'ensemble des éléments susceptibles d'être intersectés que nous avons défini à l'étape précédente.

Ainsi, cette méthode aura bien diminué le nombre de calculs d'intersections plus complexes (qui ne se fait plus sur tous les éléments, mais uniquement sur ceux retenus par l'algorithmes), et ce en accélérant la détection de la non-intersection (pas d'intersection avec une boîte $\Rightarrow$ aucune intersection avec les éléments intérieurs à celle-ci).

Section 5

Implémentation et insertion dans le canevas

Afin d'implémenter cet algorithme, nous avons créé trois nouveaux objets : un arbre, un noeud (BinaryTree.h) et un volume englobant (BoundingBox.h).

5.1 BinaryTree.h

L'arbre (tree) est simplement représenté par un pointeur vers son noeud racine et le nombre d'éléments qu'il contient. Le noeud (node), quant à lui, contient la BoundingBox qui lui est associée, ainsi que deux pointeurs vers son fils à gauche et son fils à droite.

Les différentes méthodes implémentées pour ces deux objets permettent l'accès aux éléments privés de ces derniers, de savoir si un noeud de l'arbre représente une feuille ou non et d'initialiser l'arbre (cette méthode est expliquée dans la section 5.3).

5.2 BoundingBox.h

La boîte (BoundingBox) est représentée par un vecteur contenant ses coins, par son centre, par une valeur permettant le tri des volumes englobants les uns par rapport aux autres et par l'élément qui lui est interne (pour les boîtes situées sur une feuille de l'arbre).

Nous avons implémenté deux constructeurs pour cette boîte. Le premier la construit à partir des deux boîtes qui en sont l'origine. Le deuxième, à partir du modèle qui y est présent. Ainsi, pour ce dernier constructeur, un appel à une fonction d'initialisation de la boîte a été créé pour chaque modèle existant. Notre algorithme est donc robuste à chaque volume utilisé dans ce canevas. Si une nouvelle figure y apparaissait, il serait simple de lui ajouter une fonction d'initialisation du volume englobant.

Les différentes méthodes implémentées pour cet objet permettent l'accès aux éléments le constituant et le calcul d'intersection de celui-ci avec un rayon. D'autres méthodes privées ont été développées afin d'assurer le bon fonctionnement de l'algorithme.

5.3 L'initialisation de l'arbre (fonction init dans BinaryTree.cc)

Pour cette initialisation, on distingue deux étapes :

– Lors de la première étape, nous allons créer l'ensemble des feuilles extérieures de l'arbre. A cette fin, nous allons parcourir l'ensemble des éléments de la scène. Pour chacun d'entre eux, nous allons construire la bounding box corres-

pondante. Nous insérons alors cette dernière dans une liste, en effectuant un tri par insertion sur une valeur calculée comme suit :

$$value = center.X + center.Y + center.Z \tag{1}$$

où $center.i$ est la composante en i du centre de la boîte.
Cette valeur est calculée en fonction de la somme des distances selon chaque composante entre chaque boîte. Nous pourrions calculer la distance euclidienne entre chacune, mais nous estimons que le temps de calcul gagné par l'algorithme serait alors contrecarré par le calcul de toutes les racines carrées nécessaires pour connaître cette distance.

– La seconde étape consiste en la construction de l'abre complet. Ainsi, nous allons remonter de niveau en niveau jusqu'au noeud racine en procédant par regroupement. Nous parcourons donc la liste formée à la première étape (et contenant les boîtes constituées d'un seul élément). Nous allons également créer une deuxième liste qui va contenir l'étage à reconstruire (liste des parents). Nous prenons alors deux éléments de la liste des fils. Nous leur créons un parent qui les contiendra comme fils à gauche et fils à droite, et nous ajoutons ce parent à la liste nouvellement créée. Nous répétons les opérations pour chaque groupe de deux éléments possible. Si un élément se retrouve seul à la fin, il est placé directement dans la liste des parents. A la fin de ces opérations, nous avons une liste contenant les noeuds au niveau P-1 et une au niveau P. De la même manière, nous construisons les niveaux P-2, P-3, etc. L'algorithme se termine lorsque le niveau 0 est atteint (Il reste alors 1 seul élément dans la liste : le noeud racine). L'arbre est alors construit entièrement.

5.4 Calcul de l'intersection entre un rayon et un volume englobant

Nous allons montrer que la vérification de l'intersection entre un rayon et une boîte est simple.

Le principe est le suivant : Regarder pour une des six faces de notre boîte si une intersection avec le rayon est présente ou non. Si il n'y en a pas, on effectue la même opération pour la face suivante. Si il y en a une, nous pouvons nous arrêter et dire qu'effectivement il y a intersection entre le volume englobant et ce rayon.

Nous allons donc au maximum répéter 6 fois les opérations qui suivent. Montrons alors que la vérification de l'intersection entre une face et un rayon est simple.

Imaginons que nous voulons vérifier l'intersection du rayon avec un plan perpendiculaire à l'axe des Y. Prenons maintenant la projection orthogonale du plan et du rayon (point d'origine et vecteur de direction) sur le plan formé par les axes X et Y. Puisque le plan est perpendiculaire à l'axe des Y, les ordonnées de tous les points de celui-ci sont égales. Soit y_{plan} cette ordonnée commune et soient $x_{direction}$, $y_{direction}$ les

coordonnées du vecteur de direction du rayon et $x_{position}$, $y_{position}$ les coordonnées du point d'origine du rayon.

Par Thalès, nous avons :

$$\frac{y_{position} - y_{plan}}{y_{direction}} = \frac{x_{position} - x_{intersection}}{x_{direction}} \tag{2}$$

où $x_{intersection}$ est l'abscisse du point d'intersecion du rayon avec le plan (seule inconnue de l'équation). Il suffit maintenant de calculer cette abscisse et de vérifier qu'elle se situe bien entre x_{min} et x_{max} de la face.

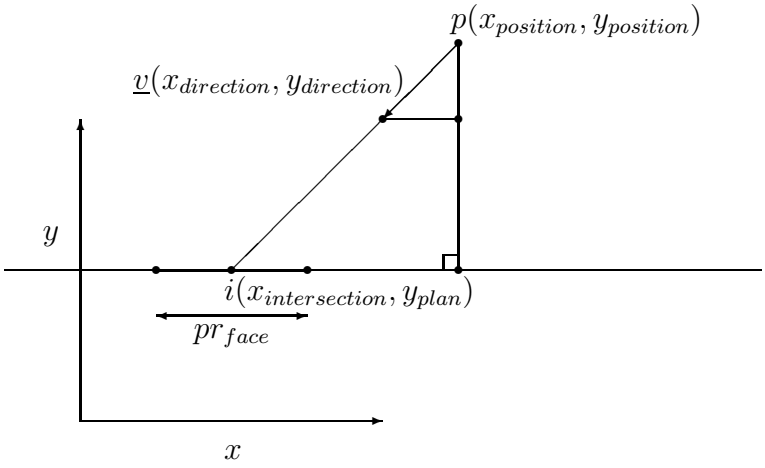

FIGURE 4 – Point d'intersection du rayon sur un plan perpendiculaire à Y

En appliquant le même théorème pour la coordonnée en Z, nous pouvons déterminer si il y a intersection ou non.

5.5 Parcours de l'arbre (fonction boudingIntersect de framebuffer.cc)

La parcours de l'arbre se fait alors tout simplement à l'aide des méthodes développées précédemment. Ainsi, nous appelons la fonction sur le noeud racine de l'arbre et nous vérifions si il y a intersection avec le noeud courant. Dans la positive, nous faisons un appel récursif à cette fonction pour le fils à gauche et le fils à droite de ce noeud. Dans la négative, nous stoppons le parcours. Lorsque le noeud est une feuille, nous ajoutons le modèle qu'il contient à la liste des modèles susceptibles d'être intersectés.

Le traitement qui avait lieu sur une liste contenant tous les modèles de la scène ne se fera maintenant plus que sur la liste créée ci-avant.

Section 6

Tests effectués et analyse des résultats

Pour mesurer l'efficacité de notre optimisation, nous nous basons sur le nombre d'appels à la fonction `getFirstIntersection` des modèles. En effet, cette fonction est relativement coûteuse en temps de calcul pour chaque modèle. Notre optimisation repose donc sur la dépendance de la vitesse du programme au nombre d'appels à celle-ci.

Voici les résultats des différents tests que nous avons effectués :

6.1 Test 1 : Sphères d'origine

Pour ce test, nous avons gardé le programme d'origine (7 sphères).

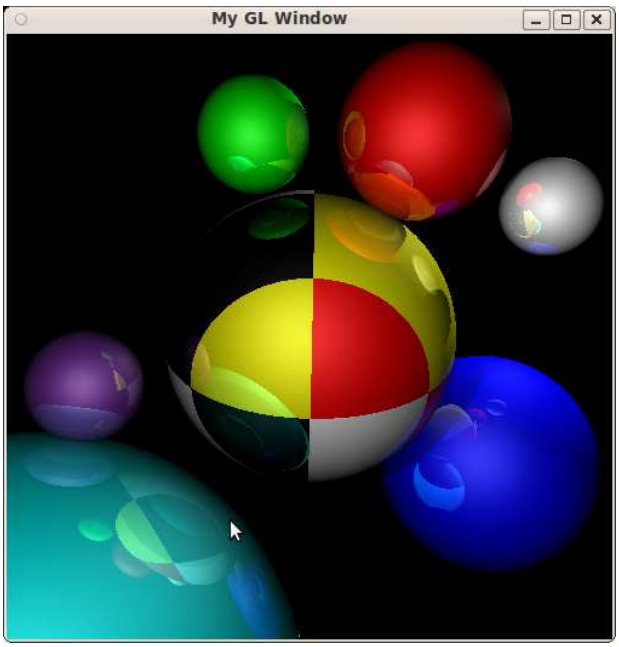

FIGURE 5 – Test 1

Le nombre d'appels à `getFirstIntersection` est de
- Avant optimisation : $2.51978 * 10^7$
- Après optimisation : $1.61632 * 10^7$

Nous avons donc un gain de 35,8%.

Ce premier test nous montre que notre optimisation fonctionne.

Son faible rendement est simplement dû au fait que sur si peu de sphères, la différence entre le nombre de niveau de l'arbre (4) et le nombre de sphère (7) est

petite. De plus la quasi totalité de l'image est occupée par une sphère, il y a donc peu de « non-intersection ».

6.2 Test 2 : Sphères du test 1 à l'échelle 1/10

Pour ce test, nous avons divisé le rayon de chaque sphère par 10. La majorité de l'image est donc noire.

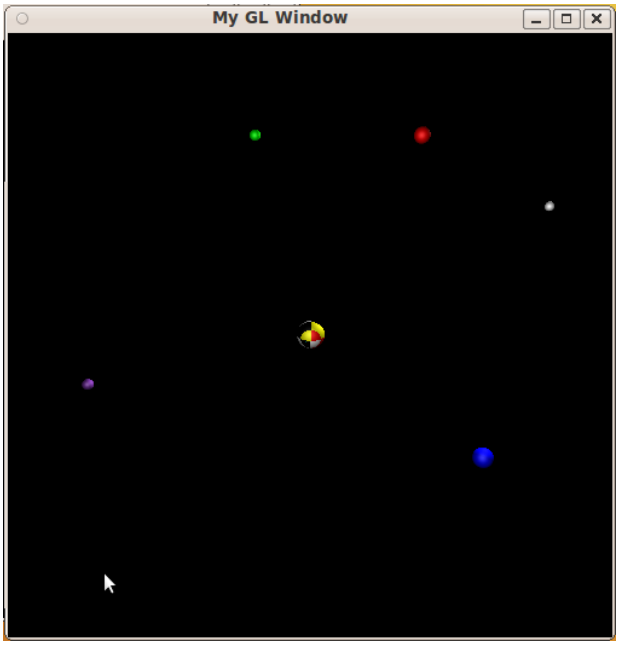

FIGURE 6 – Test 2

Le nombre d'appels à `getFirstIntersection` est de
- Avant optimisation : $1.83346 * 10^6$
- Après optimisation : 422762

Nous avons donc 76,9%.

Dans ce cas, la majorité des rayons n'intersecte rien. L'algorithme non-optimisé va calculer les intersections entre tous les rayons et tous les modèles. Notre méthode, quant à elle, va uniquement calculer les intersections entre les modèles et les rayons qui touchent une boîte contenant un seul élément. Puisque nous avons créé ces boîtes englobantes avec le moins d'espace vide possible, nous limitons le plus possible les appels à `getFirstIntersection`.

Grâce à notre optimisation, le programme va repérer les non-intersections beaucoup plus vite en parcourant l'arbre des modèles.

6.3 Test 3 : 100 petites sphères alignées

Pour ce test, nous avons changé le programme afin d'obtenir 2 lignes de 50 sphères.

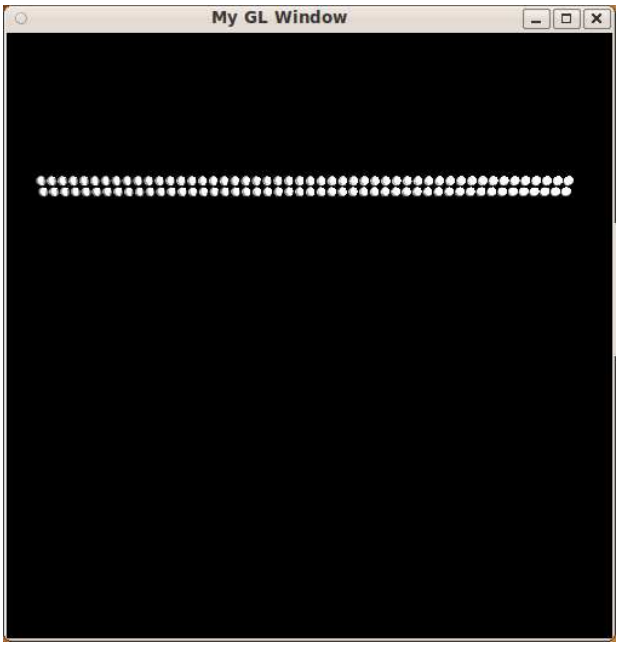

FIGURE 7 – Test 3

Le nombre d'appel à `getFirstIntersection` est de
- Avant optimisation : $3.61592 * 10^7$
- Après optimisation : 481126

Nous avons cette fois un gain de 98,6% ! !

Ce dernier test nous montre bien que plus le nombre d'objet est élevé plus le gain sera important.

6.4 Objets en dehors du champs de vision de la caméra

Grâce à notre optimisation, tous les objets se trouvant hors du champs de la caméra sont directement éliminés des calculs. En effet, lorsqu'un rayon va être testé, si l'objet se trouve hors champs, dés le test sur la racine de l'arbre, nous détectons une non-intersection.

Pour s'en convaincre, nous avons lancé un dernier test dans lequel 8000 sphères sont créées hors du champs. Le résultat est sans appel ! Sans optimisation, après plusieurs minutes de calcul, le rendu n'est pas encore arrivé à 2%, alors qu'avec l'optimisation, le rendu se termine en quelques secondes !

Section 7

Étapes et contributions

7.1 Recherche de documentation

- Deneux Thomas : 50%
- Joniaux Gilles : 50%

7.2 Comparaisons et choix des algorithmes

- Deneux Thomas et Joniaux Gilles : 100%

7.3 Conception et implémentation des structures de données utiles et de l'algorithme choisi

- Deneux Thomas et Joniaux Gilles : 100%

7.4 Tests et interprétation des résultats

- Deneux Thomas et Joniaux Gilles : 100%

7.5 Commentaires et nettoyage du code

- Deneux Thomas : 100%

7.6 Rédaction du rapport

- Deneux Thomas : 20%
- Joniaux Gilles : 80%