UNIVERSITE DE REIMS

Projet de mars

Skinning multi-instance et LOD

Paul Demeulenaere

01/04/2010

Sommaire

I. Introduction

Ce projet personnel a été réalisé en un délai relativement restreint ; pour résumer son objectif, il est intéressant de cerner ses motivations. Deux implémentations de skinning multi-instance ont été proposées par Nvidia et ATI. Je remercie Guillaume Dada de m'avoir présenté ses travaux à ce sujet.

ATI propose un modèle où le skinning est réalisé sur le GPU, il n'y a pas de niveau de détails et les déplacements sont calculés sur le CPU. Nvidia propose relativement le même modèle mais avec des groupes de niveaux de détail (LOD) mais encore une fois choisis par le CPU.

Mon objectif était de ramener tout ces calculs relatifs à l'animation de foule sur le GPU : les déplacements, le skinning mais aussi la création des groupes de LOD.

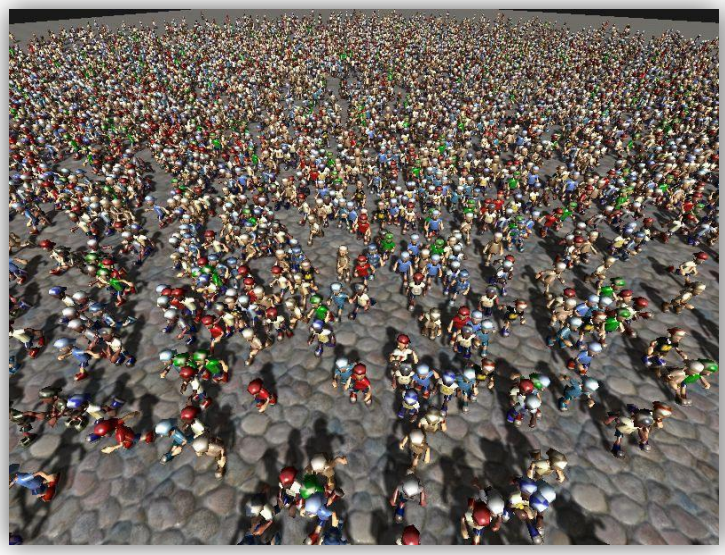

Figure 1 : Application d'ATI

Figure 2 : Application de Nvidia

II. Animation de foule

J'ai commencé par chercher à porter les algorithmes de simulation de foule sur GPU. Cette implémentation a permis de gérer beaucoup plus d'individus qu'avec une implémentation séquentielle sur CPU. Dans un premier temps, l'algorithme de simulation de foule sera rappelé pour ensuite s'intéresser à l'implémentation en parallèle sur GPU.

En 1987, Craig Reynolds proposa un modèle de simulation de foule (Reynolds, 1987) visant à approcher les mouvements d'une nuée d'oiseaux. Le principe est assez simple : chaque agent a une perception locale de l'environnement. En fonction de la position et de l'orientation de ses voisins, il va corriger son orientation en vue de se rapprocher de la masse mais aussi d'éviter les collisions. En d'autres termes, chaque individu de la foule va évaluer trois vecteurs, à savoir la séparation, l'alignement et la cohésion, pour déterminer sa nouvelle direction.

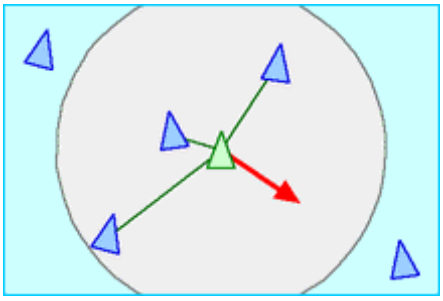

Figure 3 : Séparation

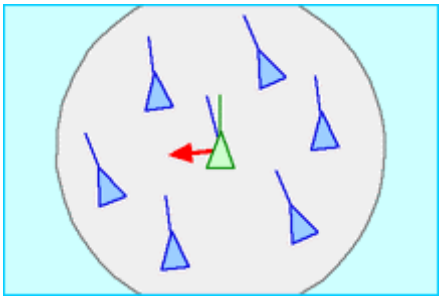

Figure 5 : Alignement

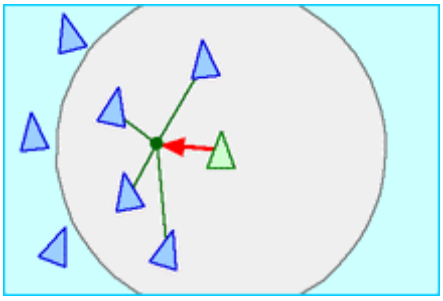

Figure 4 : Cohésion

Nous n'allons pas détailler les calculs ni exposer toutes les améliorations possibles pour, par exemple, éviter des obstacles, ou rapprocher cette nuée d'oiseaux d'une foule de piétons. Retenons simplement que ce schéma est relativement efficace et assez modulable.

Cette simulation, si elle est implémentée naïvement (c'est-à-dire sans découpage de l'espace), a l'inconvénient d'être de complexité quadratique. En effet, chaque agent doit évaluer la position de tous les autres pour ne garder que ses voisins. S'il existe un découpage de l'espace permettant à chaque agent de rapidement accéder à ses voisins, alors, la complexité est linéaire.

Revenons maintenant à l'implémentation de ce genre d'algorithme sous GPU. Tout d'abord, il est important de définir le contexte et ses limitations. Durant l'année 2009, Microsoft a proposé une nouvelle version d'API de développement graphique pour le temps réel : DirectX 11. Parmi toutes ses fonctionnalités, une des plus intéressantes est l'arrivée des « compute shaders ». Ces nouveaux types de shader permettent d'exécuter des algorithmes parallèles sur carte graphique en utilisant un langage de haut niveau. Ces shaders sont compatibles avec les cartes DirectX 10 à condition de ne

pas utiliser certaines fonctionnalités comme les variables atomiques, des multiples buffers d'accès non ordonné ou les AppendBuffer et ConsumeBuffer. Malgré ses limitations, je suis parvenu à implémenter un modèle de simulation de foule sur GPU.

Tout d'abord, une recherche bibliographique a été effectuée. Une des premières pistes d'exploration a été un article "Improving Boids Algorithm in GPU using Estimated Self Occlusion" (Silva, et al., 2008). Cette recherche propose une implémentation GPU d'une simulation de foule mais qui nécessite une exécution CPU pour placer les agents dans une grille. En effet, il n'est pas concevable de réaliser une simulation foule sur GPU avec une complexité quadratique d'autant plus que les accès mémoire sont très couteux sur ce genre d'architecture. Cette approche est intéressante mais me semblait assez limitée au niveau des performances en raison de ce passage sur le CPU. C'est pourquoi j'ai relancé mes recherches mais en ciblant maintenant la simulation physique d'un grand nombre de particules sur GPU.

Ce sont les travaux de Takahiro Harada et notamment l'article « Sliced Data Structure for Particle-Based Simulations on GPUs » (Harada, 2007) qui ont alors guidé cette implémentation. En effet, Takahiro Harada s'est intéressé à la simulation de particules sous GPU. En quelque sorte, la simulation de foule est une simulation de particules : les agents d'une simulation de foule peuvent être assimilés à des particules interagissant entre elles. L'article de Takahiro Harada propose d'utiliser une grille pour stocker les indices des particules sur GPU, ainsi l'accès aux voisins est considérablement simplifié. Dans cet article, la grille découpe l'espace en plans de tailles variables, pour diverses raisons que je ne détaillerai pas ici, j'ai simplifié ce modèle en adaptant la taille de la grille à mon ensemble de particules. Cette opération consiste à calculer la boîte englobante en cherchant les maxima et minima par une opération de réduction sous GPU (voir présentation « Sliced Grid On GPU »).

Le problème du découpage de l'espace étant résolu, les premières animations de foule ont pu être réalisées, j'ai alors choisi un maillage représentant une sardine pour tester l'implémentation de l'algorithme de Craig Reynolds sur GPU.

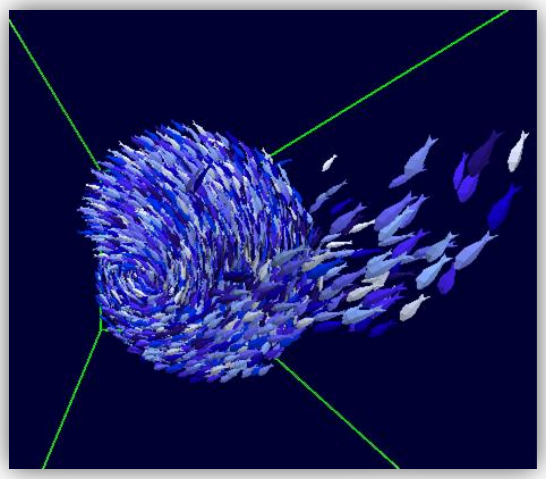

Figure 6 : Un aperçu de la simulation de foule sur GPU implémenté

Pour voir en mouvement :

- http://vimeo.com/9822654
- http://vimeo.com/9822584
- http://vimeo.com/9150570

III. Skinning de foule

Le problème de skinning multi-instance apparaît fréquemment depuis l'arrivée des fonctionnalités d'objet multi-instanciés sous DirectX 10. Le but étant de réduire toujours plus le nombre de batch sur le CPU.

ATI et Nvidia proposent chacun un modèle d'animation où toutes les informations sur l'ossature et les poids des vertex sont stockées dans une texture commune à toutes les instances. Les transformations nécessaires sont alors effectuées pour chaque vertex dans un vertex shader.

Dans les projets de ce genre, un des problèmes bloquants des programmeurs est de trouver des modèles skinnés et animés qu'il est possible de facilement charger. Par ailleurs, ces modèles étant souvent très complexes à réaliser, les auteurs ne partagent pas souvent leurs créations. C'est pourquoi j'ai choisi un format très simple et très courant pour ne pas gaspiller trop de temps en recherche de modèles. Ce format est le MD2.

Ce format date de 1997 et a été créé pour Quake 2, il n'est donc plus trop d'actualité mais il a l'avantage d'être très utilisé par la communauté de moddeur de Quake.

Les fichiers MD2 contiennent une suite de maillage de triangles, il n'y a pas d'ossature, les animations sont simplement une interpolation de deux modèles. Le modèle au format MD2 sont très légers, au plus quelques centaines de triangles, c'est pourquoi ils n'ont pas forcement besoin d'avoir des os. Certains paramètres ne sont pas animés : l'index buffers représentant les triangles et les coordonnées de textures. Les autres, les normales et positions sont animés. Ce format est très intelligent dans la taille de ces données. En effet, les composantes XYZ de la position sont stockées dans un unsigned char, les normales sont indexées, l'indice est contenu dans unsigned char lui aussi. Les informations animées pour chaque vertex sont alors contenues dans quatre unsigned char. Ainsi, il est possible de facilement stocker les animations dans une texture au format 32 bits, c'est à dire, un des formats les plus rapides en accès sur GPU.

Revenons à notre simulation de foule, on a ajouté une variable entre 0 et 1 pour chaque particule pour représenter l'avancement de l'animation. Avec les sampler corrects et en quelques lignes shaders il est alors possible d'avoir un modèle animé très peu coûteux en performance.

Pour résumer, avec une texture d'informations, une variable par agent et un vertex shader légèrement modifié, il a été possible d'animer nos maillages comme vous pouvez le constater ci-dessous. Notez que chaque instance peut avoir une frame d'animation différente.

Skinning multi-instance: ***http://vimeo.com/10004228***

IV. Gestion de LOD

La gestion des LOD est une fonctionnalité très importante dans une application temps réel : les LOD, pour level of details, permettent d'afficher un maillage plus ou moins fin en fonction de la distance à la caméra par exemple.

Cette technique permet de considérablement réduire la complexité géométrique d'une scène. Dans notre cas, plusieurs milliers de maillages sont affichés, cette quantité de vertex à transformer est un goulot d'étranglement majeur.

Nvidia propose, par son application d'exemple, un modèle de gestion de LOD, c'est cette implémentation qui a guidé ma démarche. Dans cette application, les entités sont groupées dans quatre niveaux de LOD en fonction de la distance à la caméra, chacun de ces groupes est ensuite affiché en utilisant le maillage approprié et en profitant du multi-instancing.
La création des groupes de Lod est une opération réalisée par le CPU pour Nvidia. En plus d'être potentiellement lente car séquentielle, la création de groupes sur le CPU impose plusieurs limites : on a besoin de copier les données de la mémoire du GPU vers le CPU et vice-versa, or, la bande passante du bus PCI est très faible comparée aux transferts GPU / GPU. Par ailleurs, ce genre d'opération impose au CPU d'être synchronisé par rapport au GPU.

Mon défi a donc été de ne pas ramener d'informations sur le CPU et de réaliser cette création de groupe de niveau de détails en parallèle sur le GPU.

J'avais alors besoin d'une structure de type liste me permettant d'ajouter des indices sur GPU. L'idée est la suivante : on assigne nos particules à un shader, ce dernier ajoute à un des quatre buffers (représentant différents niveaux de LOD) un indice.

La première piste a été d'utiliser les Append Buffer et Consume Buffer disponibles dans les Compute Shader. Or, cette fonctionnalité n'est accessible qu'en Shader Model 5, même si cette méthode n'a pas été retenue, elle me semble être la plus adaptée dans le cas où les fonctionnalités de niveau DirectX 11 sont disponibles.

Je me suis donc tourné vers les geometry shaders ; en effet, ces derniers permettent d'ajouter des données à un vertex buffer qui peut être réutilisé par la suite. Etrangement, alors que Direct X 10 propose d'associer quatre stream output (sous certaines conditions de taille de données), Direct X 11 requiert un niveau de compatibilité à 11 pour utiliser le multi-stream

output. Ce problème a été simplement contourné, en effectuant quatre passes de geometry shader.

Ainsi, par cette dernière méthode, et en utilisant une topologie de vertex adéquate, il est possible de dessiner indépendamment les quatre niveaux de LOD.

Reste un problème : celui du nombre d'instances par groupe de LOD. Deux méthodes se sont alors dégagées ; faute de temps, seule la première a été implémentée.
La première solution est relativement simple : elle consiste à utiliser les queries DirectX pour connaître le nombre d'append réalisés sur chaque stream output. Cette méthode a l'inconvénient d'imposer une synchronisation entre le CPU et le GPU. C'est pour éviter ce mode synchrone qu'a été dégagée une autre solution : on calcule par une réduction de nos données le nombre d'éléments de chaque niveau de LOD. (On reprend nos positions pour évaluer un uint4 pour chaque particule et les réduire par addition en CS). Ensuite, on remplit un buffer destiné à être utilisé lors d'un DrawIndirect ou plus précisément un DrawIndexedInstancedIndirect.

Ainsi, toutes les opérations sont réalisées par le GPU et il n'y a pas de synchronisation. Notons qu'avec les append buffer, il est directement possible de connaître le nombre d'éléments par niveau de LOD, c'est pourquoi ce type de buffer est très intéressant dans notre situation.

En définitive, cette méthode m'a permis de déporter l'ensemble les calculs sur GPU. La gestion des LOD permet aussi d'exclure du rendu les instances non visibles (en évaluant le produit scalaire entre le vecteur d'observation et le vecteur différentiel caméra/instance). Mon implémentation est perfectible en conséquence de certaines approximations mais j'y reviendrai plus loin.

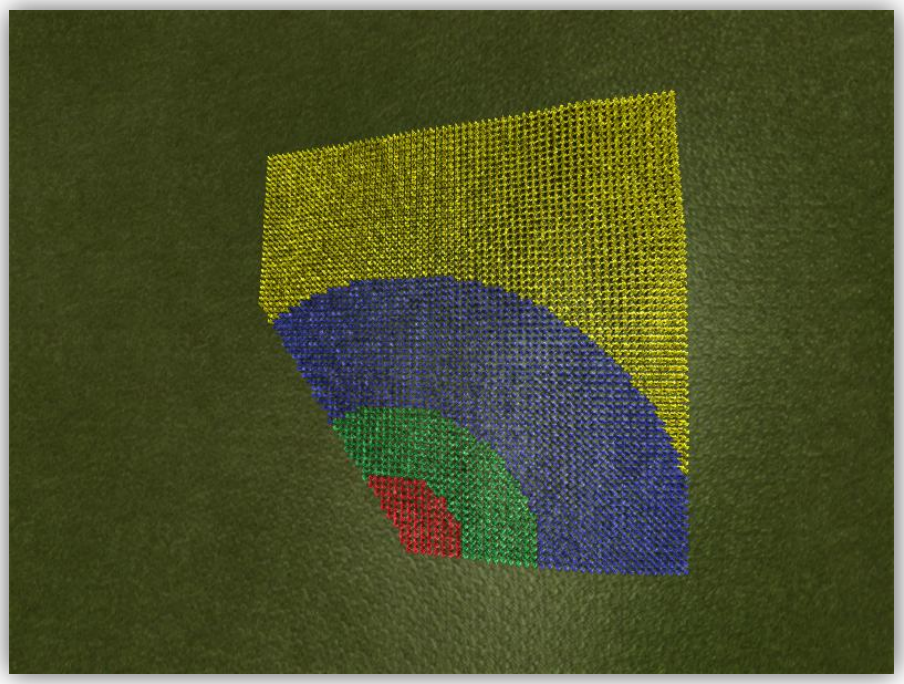

Figure 7 : mise en évidence du culling et des LOD

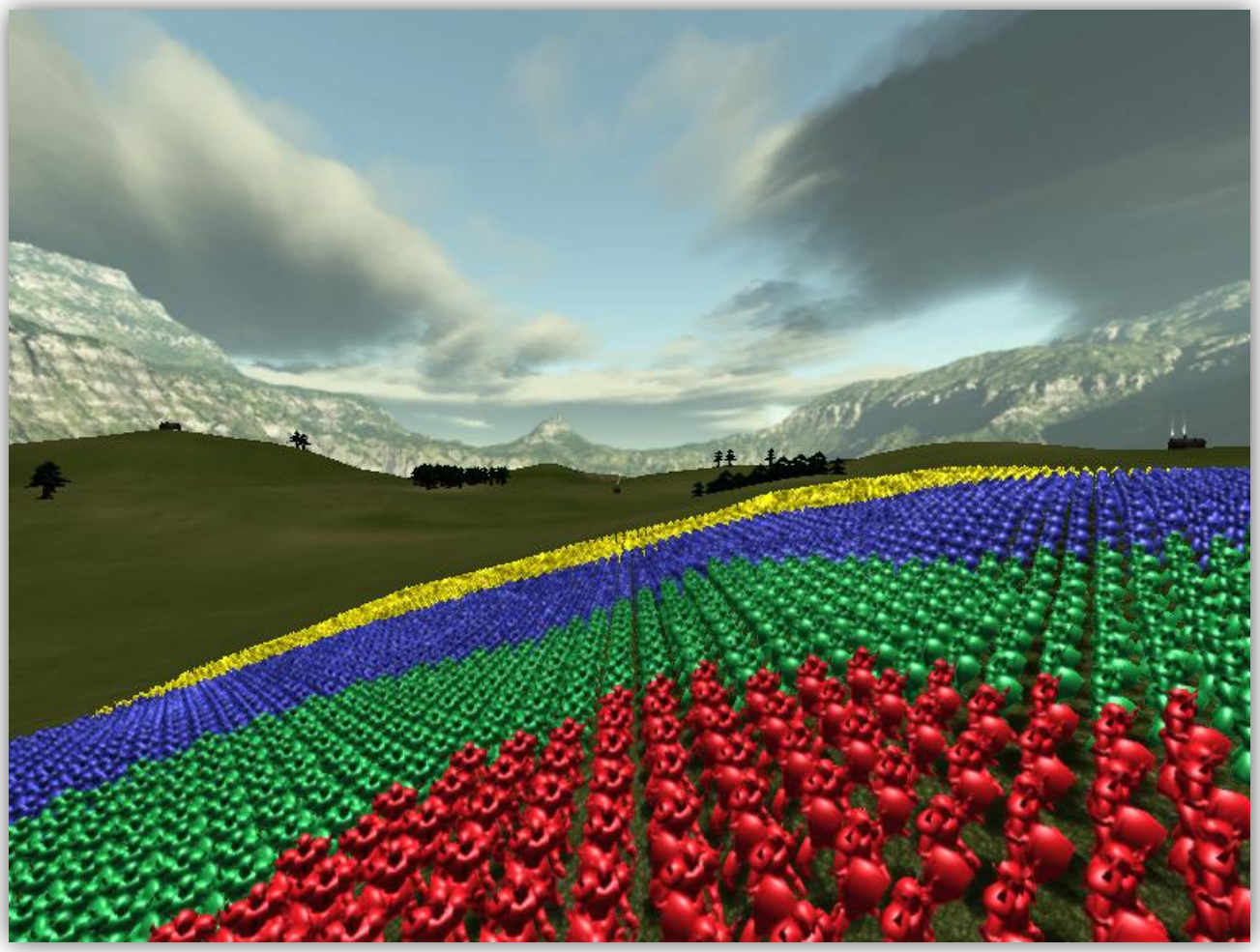

Figure 8 : mise en évidence des niveaux de LOD, en rouge, les instances proche, en jaune, les lointaines

V. Autres implémentations

Le sujet principal de ce projet est le skinning multi-instance avec gestion des LOD mais le framework développé m'a permis de réaliser plusieurs expérimentations.

1. Shadow maps

Les Shadow map améliorent significativement le rendu, l'implémentation est assez triviale. On notera simplement l'utilisation de l'offset dans les sampler pour atténuer ces ombres. Il aurait été intéressant d'utiliser des cascaded shadow maps.

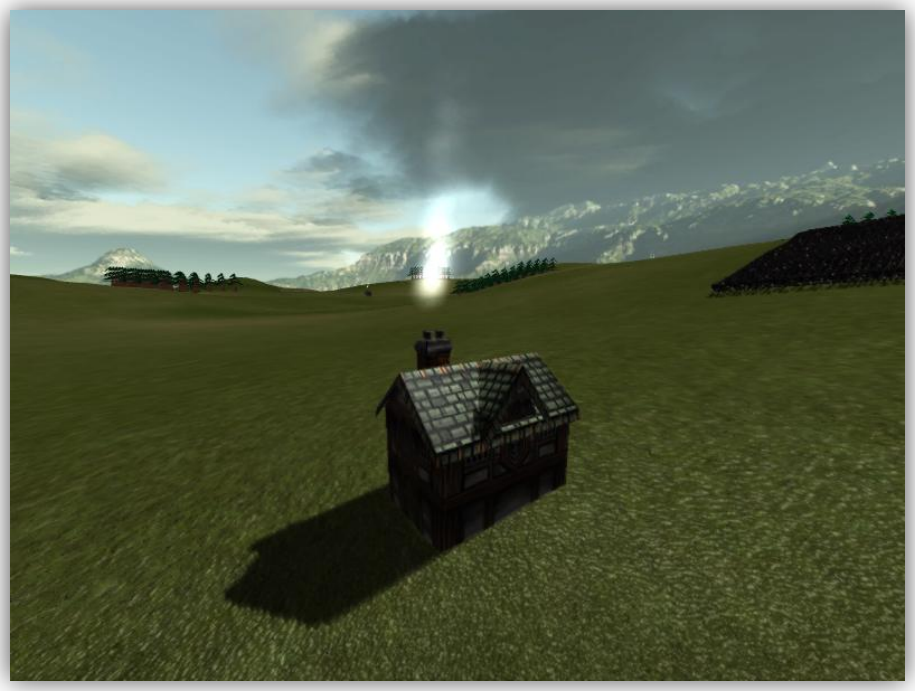

Figure 9 : Une illustration des shadow map

2. Light Pre-Pass

Ayant étudié le deferred rendering l'année précédente, j'ai voulu essayer le light pre-pass rendering. Cette technique est une variante du deferred, elle consiste à n'avoir que les normales et la profondeur dans le G-Buffer, les informations diffuses sont transmises par un seconde passe. Cette technique a l'avantage de bien supporter l'anti-aliasing mais je n'ai pas eu l'occasion de tester cette fonctionnalité de façon exhaustive.

3. Fumée

De la fumée a été générée (dessin de sprites) et animée (mise à jour par le stream-output) par les geometry shaders. Cette fonctionnalité permet de dessiner de la fumée venant des cheminées.

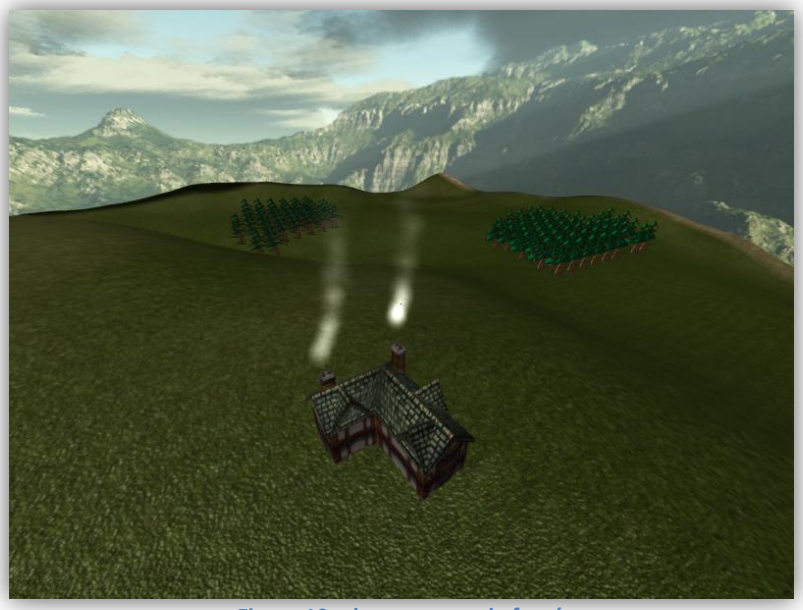

Figure 10 : deux sources de fumée

4. Parallax et bump mapping

Il y a un mélange de bump mapping et de parrallax mapping sur le sol. De même, cet objet a deux textures représentant la couleur diffuse, elles sont modulées en fonction de la pente du terrain.

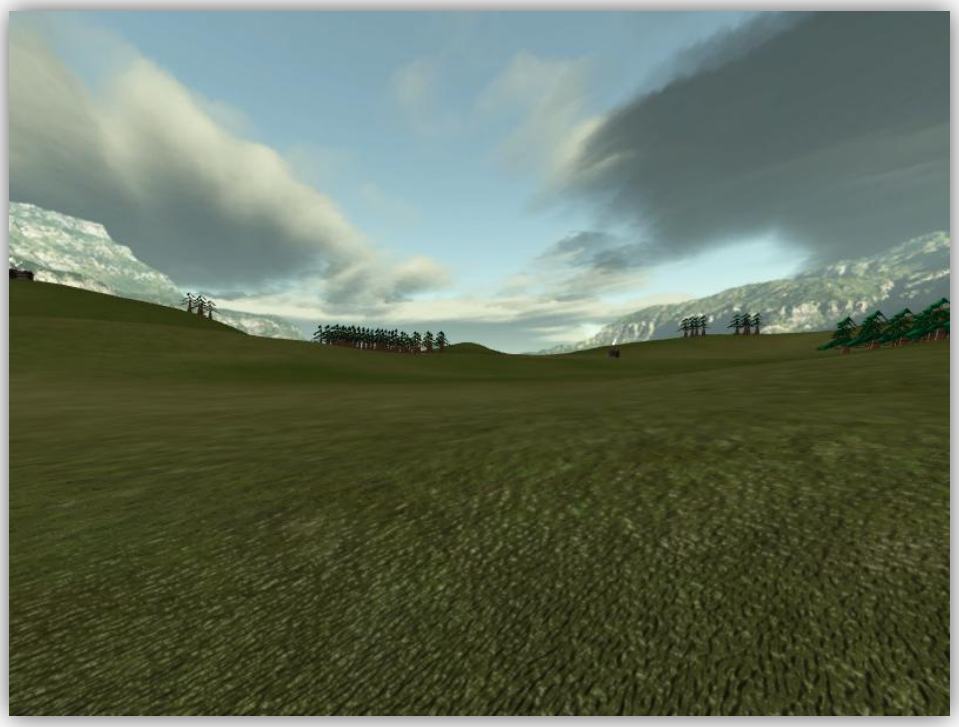

Figure 11 : Le rendu particulier du sol

5. Tessellation

J'ai profité de cette implémentation pour tester la Phong-tessellation sur carte graphique DirectX 11, je ne l'ai pas intégré dans la version finale, le code est disponible dans le dossier branche de serveur de subversion.

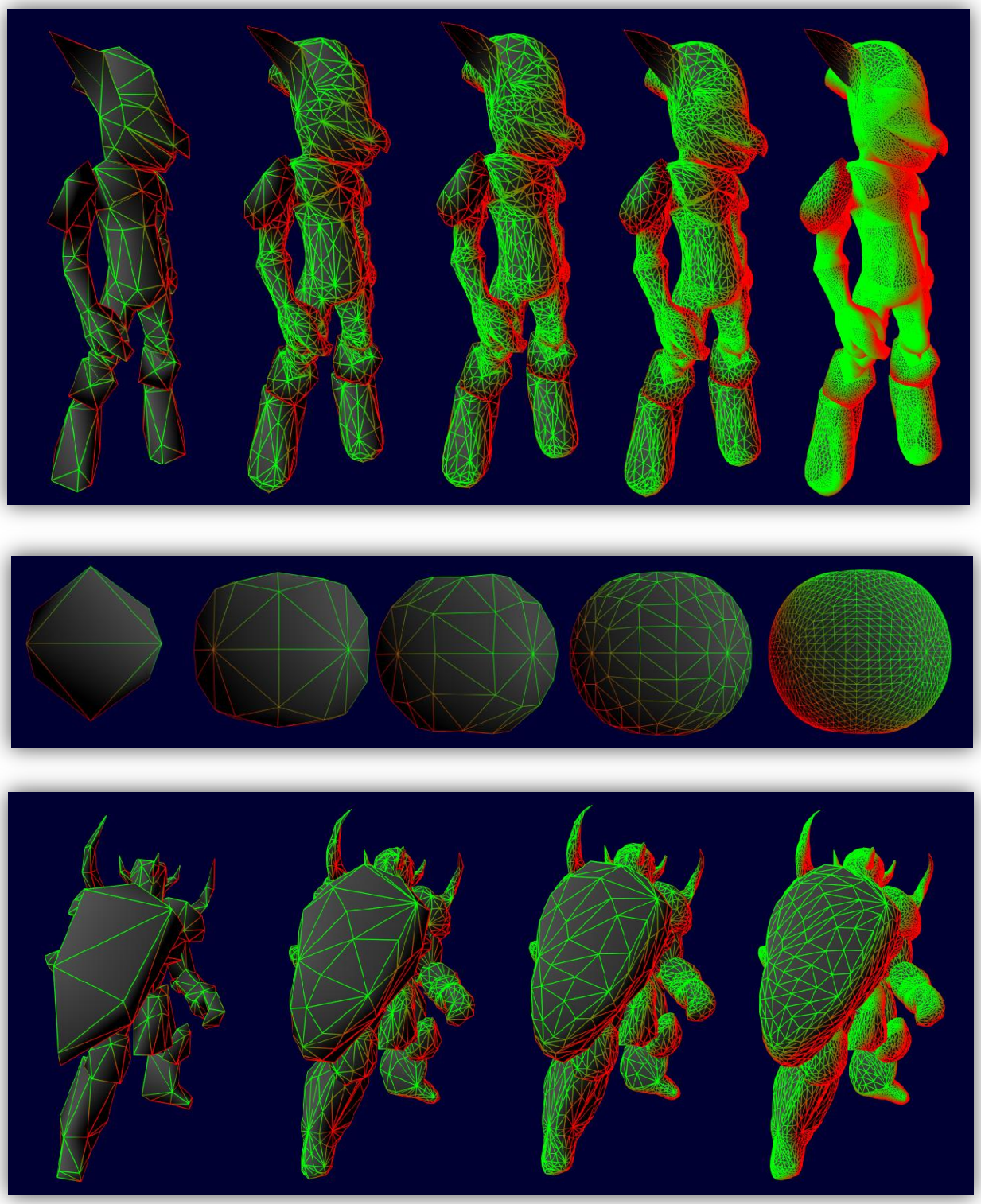

Figure 12 : Aperçu de la tessellation de Phong sur GPU

6. Alpha to coverage

Enfin, j'ai aussi pu tester les fonctionnalités d'alpha to coverage disponibles dans la configuration des états de blending de DirectX 11, cette fonctionnalité m'a permis de rendre des feuillages.

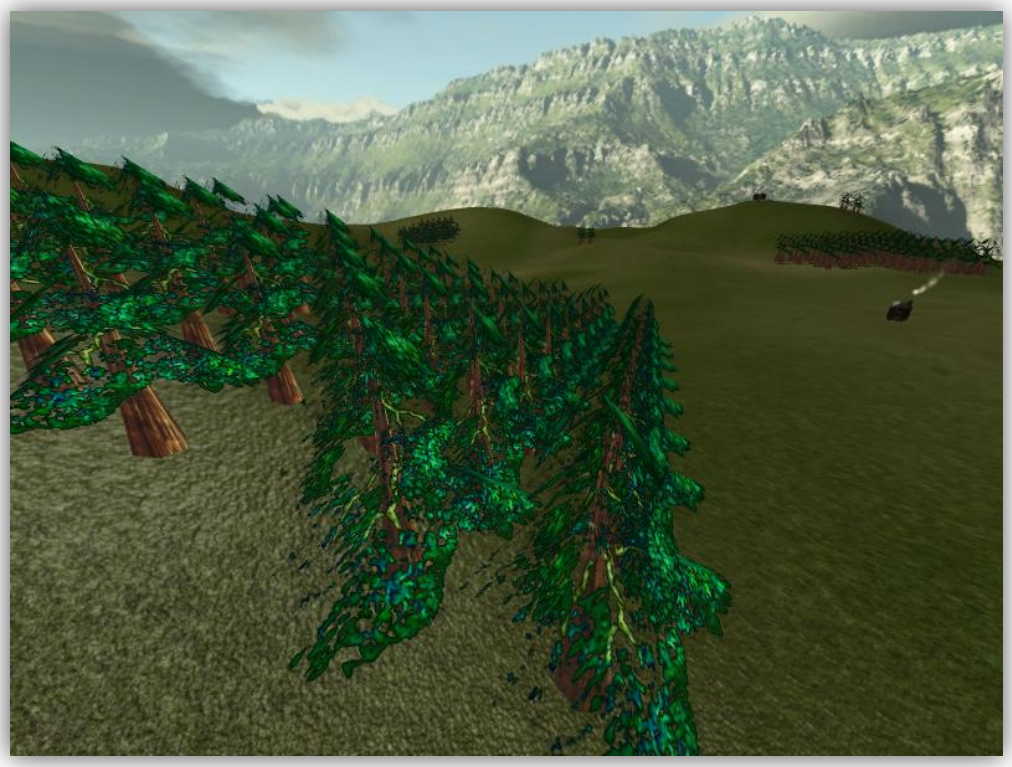

Figure 13 : Démonstration de l'alpha to coverage (les arbres sont multi-instancié)

VI. Conclusion

Comme il a été précisé précédemment, certaines approximations ont été faites pour simplifier les itérations de développement : par exemple, la position des particules utilisée lors de la détermination des niveaux de LOD est approximative car ne tient pas compte du décalage en hauteur dû au terrain. C'est un détail mais il explique quelques légers bugs.

En définitive, cette application est très satisfaisante, elle démontre les possibilités offertes par l'API de DirectX 11, je n'ai pas trouvé, pour le moment, d'équivalent pour réellement gérer une animation de foule sur GPU de cette manière. Cette implémentation est un atout très vendeur que je souhaite mettre en avant lors de ma recherche d'emploi.

Pour un aperçu de l'application finale : http://vimeo.com/10436953

Références

VII. Travaux cités

Harada, Takahiro. 2007. Sliced Data Structure for Particle-Based Simulations on GPUs. 2007. http://www.iii.u-tokyo.ac.jp/~yoichiro/report/report-pdf/harada/international/2007graphite.pdf.

Reynolds, Craig. 1987. Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics. [auteur du livre] SIGGRAPH. *SIGGRAPH '87 Conference Proceedings.* 1987, pp. 25-34. http://www.red3d.com/cwr/boids/.

Silva, Alessandro Ribeiro da, Lages, Wallace Santos et Chaimowicz, Luiz. 2008. Improving Boids Algorithm in GPU using Estimated Self Occlusion. 10 Novembre 2008. http://boid.alessandrosilva.com/sbgames08-boids.pdf.