Quantification à la volée de l'apparence sur GPU

Julien Hadim, Tamy Boubekeur, Xavier Granier, Christophe Schlick

To cite this version:

Julien Hadim, Tamy Boubekeur, Xavier Granier, Christophe Schlick. Quantification à la volée de l'apparence sur GPU. 19ièmes Journées de l'Association Française d'Informatique Graphique (AFIG), Nov 2006, Bordeaux, France. inria-00260949

HAL Id: inria-00260949

https://inria.hal.science/inria-00260949v1

Submitted on 6 Jun 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Journées de l'Association Francophone d'Informatique Graphique, Bordeaux, 2006

Quantification à la volée de l'apparence sur GPU

Julien Hadim, Tamy Boubekeur, Xavier Granier et Christophe Schlick

Projet IPARLA (LaBRI - INRIA Futurs) - CNRS - Université de Bordeaux

Abstract
In this paper, we describe a fast method for quantifying normals on GPU, in a 3D broadcast client-server context for visualization on heterogenous devices. Our method compresses on-the-fly meshes and point clouds, exploiting GPU features on server-side for providing them with quantified appearance properties — easier to store and transmit.

Dans cet article, nous décrivons une méthode rapide de quantification de normale et couleur sur GPU, dans un contexte client-serveur de diffusion de données 3D pour la visualisation sur terminaux hétérogènes. Notre méthode compresse les maillages et les nuages de points à la volée, en utilisant les spécificités du GPU côté serveur pour fournir des objets aux propriétés d'apparence quantifiées — plus léger à stocker et à transmettre.

1. Introduction

La pluralité des terminaux d'affichage 3D rend difficile la mise au point de bases de données 3D génériques, fournissant des flux de données adaptées à des clients aussi divers que les téléphones mobiles, les assistants personnels (PDA), les ordinateurs portables, les stations de travail ou bien encore les grappes de calcul. Habituellement, deux solutions sont possibles :

- stocker plusieurs fois un même objet à divers résolutions, et choisir la version à envoyer au client en fonction de ses capacités ; cette solution est simple et ne nécessite pas de calculs *en ligne*, mais nécessite le stockage/maintient de multiples versions d'un même objet ;
- convertir tous les objets de la base dans une représentation multirésolution, et filtrer un niveau de résolution à diffuser en fonction du client ; cette solution évite le stockage multiple, mais nécessite un prétraitement parfois complexe, et alourdit la charge du serveur lors de la diffusion (extraction du niveau de détail).

Une solution combinant les avantages des précédentes serait de ne stocker qu'une version de chaque objet — à pleine résolution — et de construire les niveaux de résolutions à la volée. Malheureusement, le coût calculatoire dans ce cas rendrait impraticable un nombre non trivial de requêtes simultanées sur un serveur de diffusion 3D, réduisant son utilisation à un faible nombre de clients.

En fait, en observant l'évolution de la puissance 3D des terminaux d'affichage, on remarque qu'ils atteignent pour la plupart aujourd'hui des performances raisonnables en affichage (de quelques dizaines de milliers à plusieurs millions de polygones). Par contre, la charge des réseaux et la latence reste un problème. Il s'agit donc moins d'un problème de complexité géométrique des données que d'un problème de taille des données à transmettre qui se pose. Ce qui nous amène à envisager la compression systématique des données à transmettre.

À nouveau, le problème du stockage multiple se pose. Hors, dans ce cas, il est possible d'utiliser un certain type de compression, particulièrement simple et effectif : la *quantification*. De toutes les techniques de compression, la quantification a plusieurs avantages :

- le gain est exactement déterminé à l'avance ;
- l'aspect systématique et mécanique de la quantification la rend parallélisable.

Considérant un maillage ou un nuage de points, la normale et la couleur associées aux échantillons géométriques représentent les deux tiers du poids de l'objet. Quantifier ces deux attributs peut donc amener un gain significatif sur la taille des fichiers. Malheureusement, même les quantification les plus simples peuvent nécessiter un certain temps de traitement (de l'ordre de quelques dizaines de secondes pour un objet de plusieurs millions d'échantillons), introduisant

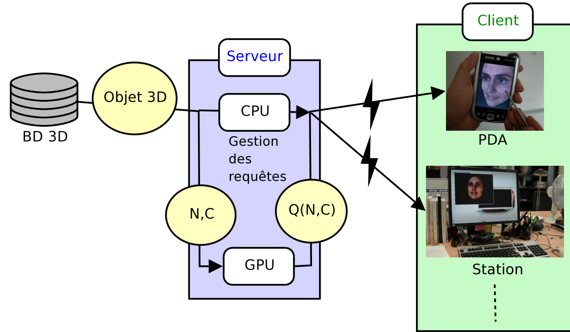


Figure 1: *Le contexte de notre système de quantification à la volée est une application client-serveur de visualisation 3D. La charge de traitement de quantification effectuée à la volée est déléguée au GPU.*

une forte latence dans l'accès aux données si cette quantification est effectuée *à la volée*.

Nous proposons une méthode de quantification de normales et de couleurs spécifique tirant parti du GPU pour réduire cette latence (voir Figure 1). Les propriétés de normale et de couleur sont quantifiées par le GPU du serveur, réduisant la charge CPU au traitement des entrées/sorties. Déplacer cette quantification sur le GPU fait apparaître deux avantages très nets :

- une accélération du traitement d'un minimum de 25%
- une diminution de la charge du CPU, augmentant le nombre de requêtes simultanées pouvant être prises en charge.

La suite de cet article est organisée comme suit : la section 2 récapitule les travaux antérieurs relatifs à notre méthode ; la section 3 détaille le principe de notre quantification basée GPU ; la section 4 détaille l'implémentation et la section 5 les performances obtenues expérimentalement ; enfin la section 6 conclut et propose divers possibilités pour améliorer le système.

2. Travaux précédents

Depuis les travaux fondateurs de Deering [Dee95], la compression géométrique est un domaine de recherche très actif. Cependant un large nombre de solutions développées dans le contexte géométrique ne sont pas adaptées aux normales car elles ne produisent pas un échantillonnage uniforme et isotrope dans l'espace des directions.

Une approche naturelle est de quantifier chaque coordonnée de normale sur un nombre réduit de bits [BPZ99]. Facile à implémenter, cette solution n'est ni uniforme ni isotrope. De plus, cette méthode ne préserve pas la norme des vecteurs quantifiés. Le nombre de bits peut être optimisé par *partitionnement* [KCK04], mais une table d'indexation dépendante de l'objet est nécessaire pour la décompression.

La quantification basée sur un octaèdre, proposée par Deering [Dee95], a été largement utilisée afin d'obtenir une compression plus uniforme des normales. Dans cette méthode, l'espace des directions est subdivisé en 8 sections, selon un octaèdre. Chaque section est ensuite subdivisée en 6 parties, et sur chaque sextant, les angles des directions sont encodés en utilisant une grille régulière. Deering affirme que, uniformément et isotropiquement distribuées, 100 000 directions (une densité angulaire de $0,01$) sont suffisantes pour une quantification sans artefacts visibles. Étant donné que la grille régulière ne fournit pas une telle distribution, 200 000 normales sont utilisées en pratique, via un encodage sur 18-bits, afin d'obtenir une quantification de haute qualité.

Taubin et al. [THLR98] utilise un quadtree régulier sur chaque face de l'octaèdre. L'encodage et le décodage sont alors récursifs, et la normale résultante doit être normalisée. L'avantage majeur de cette approche est une distribution plus uniforme et une meilleur exploitation des bits utilisés. Botsch et al. [BWK02] se restreignent à une plage de 13 bits (i.e., 8 192 directions) dans le cadre du rendu par points.

QSplat [RL00] introduit une segmentation de l'espace des directions basée sur le cube. Les normales quantifiées sont obtenues en échantillonnant une grille 52×52 sur chacune des six faces, combinée à une fonction de déformation afin d'échantillonner plus uniformément l'espace des directions. Une table d'indexation globale de 16 224 entrées est utilisée pour la décompression, la normale étant encodée sur 14 bits. Une méthode de streaming [RL01] a par la suite été proposée pour diffuser ces modèles. Cependant, cette quantification est effectuée via un lourd pré-traitement.

Notre méthode peut être vue comme une combinaison de la quantification basée sur l'octaèdre et de celle basée sur le cube. Notre approche s'inscrit dans le cadre des *méthodes hors-mémoire en flux*, précédemment exploitées dans le cadre des traitements hors-ligne [ILS05, IG03].

On notera que l'analyse locale d'un voisinage sur une *cartes de normales* [MAMS06] donne de meilleurs taux de compression à poids égal que les méthodes purement locales comme la notre. Dans cet article, nous ne considérons pas le cas des cartes de normales ou des textures, mais plutôt celui des maillages et nuages de points en haute résolution sans paramétrisation de surface, où les propriétés simples d'apparence (normale+couleur) sont encodées par sommet/point.

3. Quantification rapide

3.1. Description

Intuitivement, notre technique pour la quantification des normales combine celle basée sur l'octaèdre et celle basée sur le cube. Elle utilise un cube qui, combiné à une fonction de déformation, offre une distribution des normales plus uniforme. Elle prend aussi en compte toutes les symétries naturelles de l'espace des directions, à la manière de la quan-

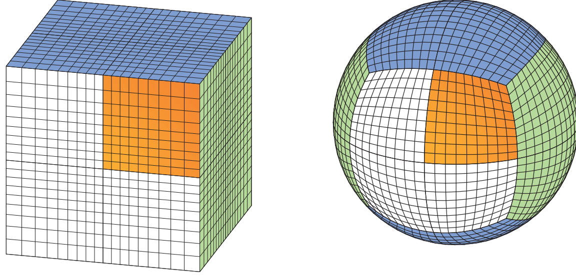


Figure 2: *Lignes déformées sur un cube unitaire par la fonction de déformation (gauche) puis projetées sur une sphère unitaire (droite), la partie orange correspond à la taille de la table d'indexation.*

tification basée sur l'octaèdre. La taille de notre table de décompression — utilisée lors de la reconstruction — est ainsi divisée par 24. Finalement, à l'instar de Qsplat [RL00], notre quantification de normales peut être combinée à une quantification de la couleur. Dans un premier temps, Nous avons simplement quantifié chaque composante RGB sur un nombre identique de bits, de manière à compresser l'apparence d'un sommet sur un mot de 32 bits. Ainsi lorsque nous encodons la normale sur 16 ou 17 bits, nous quantifions la couleur respectivement sur 16 ou 15 bits, tout en restant compatible avec le format interne des cartes graphiques.

3.2. Quantification

Puisqu'il n'existe pas de paramétrisation globale uniforme sur la surface d'une sphère, une table d'indexation est la méthode la plus efficace et la moins coûteuse pour reconstruire une normale en une représentation flottante en fonction de son code de quantification. Afin de réduire la taille non négligeable de la table d'indexation, le domaine sphérique peut être partitionné. En superposant un cube unitaire aligné sur les axes et une sphère unitaire, on obtient 6 sections sphériques identiques. De plus, chaque section peut être découpée en 4 parties : les symétries définies par les axes permettent de définir la notion de *symétrie horizontale* et de *symétrie verticale* dans le repère locale de la section. Notre table d'indexation peut donc tirer parti de ces symétries et être réduite à $1/24^{ème}$ de la sphère (zone orange dans la Figure 2-gauche). La table d'indexation contient les normales, dans leur représentation flottante et normalisée, correspondant à un quart de la face +X du cube, en référence au dénomination des faces du "cubemap" des cartes graphiques ($\pm X, \pm Y, \pm Z$).

Chaque section de la sphère associée à une face du cube est quasi-uniformément paramétrisée grâce à une fonction de déformation D_f sur la paramétrisation u, v de la face du cube associé :

$$D_f : \begin{array}{ccc} [-1,1]^2 & \rightarrow & [-1,1]^2 \\ (u,v) & \rightarrow & (f(u), f(v)) \end{array}$$

avec $f(t) = \tan(4t/\pi)$. L'encodage pour la face du cube et les symétries se font sur 5 bits. (i.e., les 24 différentes positions de la table d'indexation sur la sphère). Par simplicité la combinaison est faite par des masques:

- la face du cube est encodée sur trois bits ;
- les deux symétries axiales sur un bit chacune.

Une fois que l'entête est encodée, tous les bits restant sont utilisés pour définir un index linéaire dans la table d'indexation. Plus précisément, pour une résolution k^2, chaque couple (i, j) sur le quart de face correspond à l'index classique $i \times k + j$. Sur 16 bits, les 11 bits restant encodent 45^2 index (voir Figure 3), correspondant à 48 600 normales.

Pour pousseur plus loin la quantification de l'apparence, il est possible de combiner la couleur avec la normale associée à chaque point/sommet. La résolution de la quantification pour les normales peut alors atteindre 98 304 directions. L'encodage est constitué de deux mots de 16 bits (avec 5 bits par composante de la couleur, voir Figure 3). Cette représentation est donc compatible avec le format de texture présent sur la majorité des composants graphiques et notamment sur les assistants personnels ou téléphones mobiles, c'est à dire `RGB_5_6_5`. Dans un premier mot, nous combinons l'entête (en 5 bits), la première partie de l'index (en 6 bits) puis la composante rouge (c'est à dire i, en 5 bits). Dans le second, nous combinons la composante verte (en 5 bits), le reste de l'index (c'est à dire j, en 6 bits) et la composante bleu (en 5 bits).

Notre modèle de quantification est adapté au processeur graphique (GPU) puisque que le processus de quantification consiste en un unique et simple accès à une texture "cubemap" avec la normale en paramètres de texture. Toutes les valeurs quantifiées des directions sont donc pré-calculées et stockées dans une texture *cube-map* d'une résolution supérieure à celle de la quantification, ceci afin de limiter la quantification implicite induite par l'accès à la texture "cubemap".

3.3. Reconstruction

Le processus de *déquantification* (i.e., reconstruction d'une normale $n \in \mathbb{R}^3$) consiste au maximum à un accès à la table d'indexation suivi de trois inversions et deux permutations entre les coordonnées x, y, z de la normale (voir Algorithme 1). Premièrement l'index est décodé puis utilisé pour retrouver la représentation flottante de la normale dans

<table>
<tr><td>face</td><td>SV</td><td>SH</td><td colspan="2">index</td></tr>
<tr><td>face</td><td>SV</td><td>SH</td><td>index 1</td><td>rouge</td></tr>
<tr><td colspan="3">vert</td><td>index 2</td><td>bleu</td></tr>
<tr><td colspan="3">5bits</td><td>6bits</td><td>5bits</td></tr>
</table>

Figure 3: *En haut : normale encodée sur* 16 *bits. En bas : normale et couleur encodée sur* 32 *bits*

Algorithm 1 Déquantification

```
fonc Décodage(index_normale_quantifiée)
    Décoder index, face, les symétries SH et SV
    Retrouver x,y,z à partir xyzLookUp[index]
    Si (SV) alors z = -z
    Si (SH) alors y = -y
    Suivant (face)
        Si '±X' : return (±x, y, z)
        Si '±Y' : return (z, ±x, y)
        Si '±Z' : return (z, y, ±x)
```

la table d'indexation. Deuxièmement, la face et les deux symétries sont décodées puis la normale est transformée pour correspondre au quart de face correspondante. Pour atténuer le crénelage que la quantification peut introduire, on utilise une technique de *tremblement local* pour perturber la normale aléatoirement : notre représentation étant quasi-uniforme, le pas du bruit ajouté doit être maintenu inférieur au pas minimum de la quantification.

4. Implémentation GPU

Nous avons expérimenté notre compression d'apparence sur GPU dans le contexte de la compression d'attributs (normale et couleur) par sommets (resp. point) d'un maillage polygonale (resp. nuage de points non uniforme). L'encodage est conforme au format de texture `RGB_5_6_5` utilisé pour l'instant dans la texture cube pour l'encodage des normales. La conformité à ce format peut permettre aussi, dans le cas de maillage paramétrisé, de spécifier des attributs d'apparence dans des textures. L'algorithme de déquantification quant à lui nécessite des adaptations spécifiques pour être adapté sur GPU due aux limitations des unités de calculs pour sommets ou fragments de la carte graphique.

4.1. Encodage

La principale contribution de notre méthode est de permettre un encodage en flux à la volée (sans stocker à aucun moment la totalité de l'objet en mémoire). La boucle de traitement se scinde ainsi en 3 étapes :

1. Lecture d'un ensemble de taille fixe dans un tampon de sommets sur l'entrée du flux (fichier ou socket)
2. Rendu sur GPU effectuant la quantification sur le processeur de fragment
3. Écriture du résultat sur le flux de sortie.

Le processus de quantification étant ramené à un simple accès texture cube sur la carte graphique et cette opération s'effectuant dans l'unité de calcul par fragment, les données à encodée doivent donc être transmises via des textures au GPU. De même, le résultat de la quantification du tampon de normale est en pratique contenu dans le tampon de mémoire vidéo ("framebuffer") courant du GPU. Afin de mutualiser les représentation, l'entrée (c'est à dire la texture contenant les normales) et la sortie (le tampon vidéo contenant les normales quantifiées) du GPU sont instanciées via des *Pixel Buffer Objects* (PBO), directement dans la mémoire graphique.

La quantification se réduit alors au rendu d'un unique quadrilatère texturé dans un contexte graphique à la même résolution que la texture des données à encoder. De cette manière, nous nous assurons d'une bijection entre chaque texel de données (normale à quantifier) et chaque pixel en sortie (normale quantifiée). A chaque nouveau tampon de normales lu, on rend une image et le résultat est directement écrit sur le flux de sortie. Le système fonctionne donc à mémoire constante, ce qui permet de traiter plusieurs objets de grande taille en parallèle sur un même serveur : le goulot d'étranglement apparaît désormais sur les entrées/sorties, le CPU étant complètement libéré de toute tâche de quantification.

4.2. Optimisation des entrées/sorties

Afin de réduire les temps de transfert de données CPU/GPU, nous avons utilisé une fonctionnalité récente, les PBO, permettant des accès mémoires directes sur GPU depuis le CPU. Grâce à ces tampons mémoires, en les combinant et les alternant en entrées et en sorties, les transferts de données deviennent asynchrones, accélérant encore les transferts (voir Figure 4).

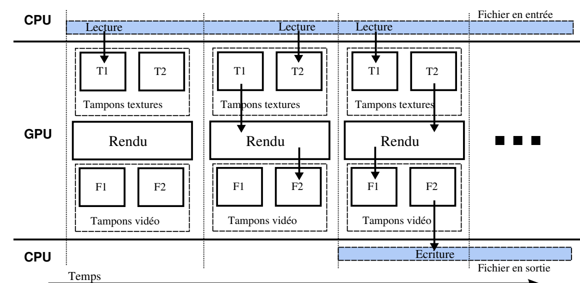


Figure 4: *Quantification asynchrone à la volée : en flux et hors-mémoire en utilisant les PBO. Dans un premier temps on lit sur le fichier les premières données que l'on stocke dans un tampon texture: $T1$. Dans un deuxième temps, on lit de nouvelles données stockées dans un deuxième tampon texture: $T2$ et on effectue le rendu avec le tampon de texture précedent $T1$ dans un tampon vidéo: $F2$. Dans un troisième temps, on lit des nouvelles données dans $T1$, on effectue le rendu de $T2$ dans un autre tampon vidéo: $F1$ et on écrit les données de $F2$ dans le fichier de sortie. On procède ainsi jusqu'à ce que tout le fichier soit quantifié.*

4.3. Décodage

Le décodage s'effectue côté client, soit sur le CPU dans le cas de clients à faible ressources (e.g., PDA sans accélérateur graphique) soit sur GPU si le client est équipé d'un processeur graphique.

Dans le cas du décodage sur CPU, nous avons directement implémenté l'algorithme décrit précédemment (voir Algorithme 1). Pour décoder les normales directement sur GPU dans le cas d'attributs par sommet et donc au niveau de l'unité de calcul par sommet, nous avons due adapter notre méthode de reconstruction. En attendant l'arrivée d'opérateurs binaires sur GPU [Bly06], nous utilisons la technique introduite par Purnomo et al. [PBCK05]. Les attributs pour l'apparence d'un sommet, encodés avec deux mots de 16 bits (voir Figure 3), sont transmis en tant que 2 mots de 32 bits de type flottant. Ainsi encodés, nous pouvons utiliser les fonctions de partie entière et fractionnelle de la carte graphique pour simuler l'action d'opérateurs binaires et ainsi décoder nos bits d'informations (voir Détails [PBCK05]).

Même notre table d'index réduite grâce aux symétries est encore trop grande pour tenir dans les registres du GPU. De plus, une fonctionnalité récente qui permettrait de stocker la table dans une texture au sein des unités de calcul par sommet n'est pas encore assez performante. Nous utilisons donc une table d'index de taille 8×8 et des interpolations bilinéaire. Pour un index donné d'une table complète (64×64 pour une normale sur 17 bits), les 4 plus proches normales dans la table 8×8 sont déterminées et reconstruites à l'aide de la table d'index réduite. La normale finale est obtenue par interpolation bilinéraire entre les 4 normales reconstruites puis normalisée. Pour décoder la face du cube et les symétries, nous utilisons une table contenant les 24 configurations possible encodées par l'entête de 5 bits. Chaque entrée de cette table est un vecteur indiquant la face et les deux symétries correspondantes.

Si la couleur est quantifiée, nous appliquons la méthode qui émule les masques de bits et procédons au calcul classique d'éclairage avec la normale et la couleur. Pour réduire les artefacts de quantification, du bruit peut être ajouté au niveau des coefficients d'interpolations pour perturber la normale reconstruite et atténuer le crénelage introduit.

5. Résultat

5.1. Performance de la compression à la volée

Si nos méthodes de compression et de décompression pour les normales et pour le couple normale/couleur ont toutes été implémentées sur carte graphique, seule la quantification à la volée des normales à été expérimenté sur GPU.

Nous avons comparé les temps de quantification de normales à la volée, pour des modèles de différentes tailles, sans et avec l'utilisation du GPU, puis en mode asynchrone avec le GPU (cf. Table 1). Nos mesures ont été effectuées sur un PC Pentium 4 3Ghz avec 1Go de RAM et une carte graphique NVIDIA Quadro 4400. L'utilisation du GPU améliore les temps de quantification d'un facteur de 25 à 35% de manière générale. Lorsque les transferts de données deviennent asynchrones les gains sont de 30 à 40%.

Modèles	# points (millions)	CPU (s)	GPU (s)	GPU asynchrone (s)
Face	0,04	0,21	0,06	0,36
Raptor	4,00	2,19	1,46	1,30
David	56,0	25,7	19,6	18,2

Table 1: *Comparaison des temps de quantification des normales pour des modèles à base de points. Notons que l'utilisation du mode asynchrone pour des modèles de petite taille augmente le temps de quantification due au temps d'initialisation de ce mode*

L'utilisation des GPUs apporte donc un gain significatif dans notre contexte puisque le temps de quantification est réduit, et donc le temps de charge du CPU.

5.2. Erreur

La figure 6 montre un objet de taille moyenne (8M polygones) rendu avec un matériau spéculaire, en présence de plusieurs sources de lumière. L'erreur RGB est données multipliée par 100. On observe une faible perte relative entre la version quantifiée et l'originale. En calculant l'erreur perceptuelle *Delta E* (CIE 1976) sur la même image, on obtient une erreur moyenne de $8,8\ 10^{-2}\%$ avec une erreur maximum de $9,7\%$ (erreur divisée par l'erreur maximum possible lorsque les composantes colorées varient entre 0 et 1 dans un espace RGB).

Nous avons aussi expérimenté la qualité de notre quantification combinée avec couleur et normale pour des modèles polygonaux texturés. Sur les résultats (cf. Figure 5), l'erreur moyenne est de 0.18% et celle maximum de 17%, et sont plus élevés que dans le cas précédent. Dans ce contexte où les normales et les couleurs étaient stockés dans deux textures en 24-bits, le taux de compression obtenue est de 25%. Si les erreurs sont perceptibles en comparaison avec l'image de référence, le résultat quantifié reste de très bonne qualité.

6. Conclusion et Travaux Futurs

Nous avons proposé un système de quantification pour les propriétés d'apparence visant les applications client-serveur de visualisation d'objet 3D.

Afin de réduire au maximum la charge CPU, l'essentiel de la charge de la quantification est construite pour fonctionner directement sur le GPU, via un flux de rendu horsligne. Notre méthode fonctionne à mémoire constante. Nous avons introduit également une combinaison efficace des quantifications rapides par octaèdre et cube. Notre solution améliore significativement les performances de quantification du serveur et autorise un plus grand nombre de requêtes simultanées pour la visualisation de bases de donnée 3D en ligne depuis des terminaux hétérogènes.

Nous prévoyons à l'avenir de déporter un maximum de

traitements géométriques (côté serveur) en flux sur le GPU, avec en particulier la simplification et la quantification de la géométrie, ainsi que la construction de niveaux de résolution à la volée.

De plus, afin de limiter le coût de décompression nécessaire sur le client pour l'interprétation des données, nous travaillons sur des solutions permettant d'effectuer la plupart des calculs (e.g., éclairage) dans l'espace des normales ainsi quantifié.

Remerciements : Nous remercions Mickaël Raynaud, ingénieur de développement au sein de l'équipe IPARLA INRIA-Futurs, pour l'adaptation de la visualisation des modèles quantifiés sur terminaux mobiles.

References

[Bly06] BLYTHE D.: The direct3d 10 system. In *Proc. ACM Siggraph* (2006).

[BPZ99] BAJAJ C. L., PASCUCCI V., ZHUANG G.: Single resolution compression of arbitrary triangular meshes with properties. In *Proc. IEEE Conference on Data Compression '99* (1999), p. 247.

[BWK02] BOTSCH M., WIRATANAYA A., KOBBELT L.: Efficient high quality rendering of point sampled geometry. In *Proc. Eurographics workshop on Rendering 2002* (2002), pp. 53–64.

[Dee95] DEERING M.: Geometry compression. In *Proc. ACM SIGGRAPH '95* (1995), pp. 13–20.

[IG03] ISENBURG M., GUMHOLD S.: Out-of-core compression for gigantic polygon meshes. In *Proceedings of SIGGRAPH'03* (2003).

[ILS05] ISENBURG M., LINDSTROM P., SNOEYINK J.: Streaming compression of triangle meshes. In *Proceedings of 3rd Symposium on Geometry Processing* (2005).

[KCK04] KIM D.-S., CHO Y., KIM H.: Normal vector compression of 3d mesh model based on clustering and relative indexing. *Future Gener. Comput. Syst. 20*, 8 (2004), 1241–1250.

[MAMS06] MUNKBERG J., AKENINE-MÖLLER T., STRÖM J.: High quality normal map compression. In *Proceedings of ACM Siggraph/Eurographics Graphics Hardware* (2006).

[PBCK05] PURNOMO B., BILODEAU J., COHEN J. D., KUMAR S.: Hardware-compatible vertex compression using quantization and simplification. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), pp. 53–61.

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *Proc. SIGGRAPH '00* (2000), pp. 343–352.

[RL01] RUSINKIEWICZ S., LEVOY M.: Streaming qsplat: A viewer for networked visualization of large, dense models. In *ACM I3D* (2001).

[THLR98] TAUBIN G., HORN W., LAZARUS F., ROSSIGNAC J.: Geometry coding and vrml. *Proceedings of the IEEE 86*, 6 (1998), 1228–1243.

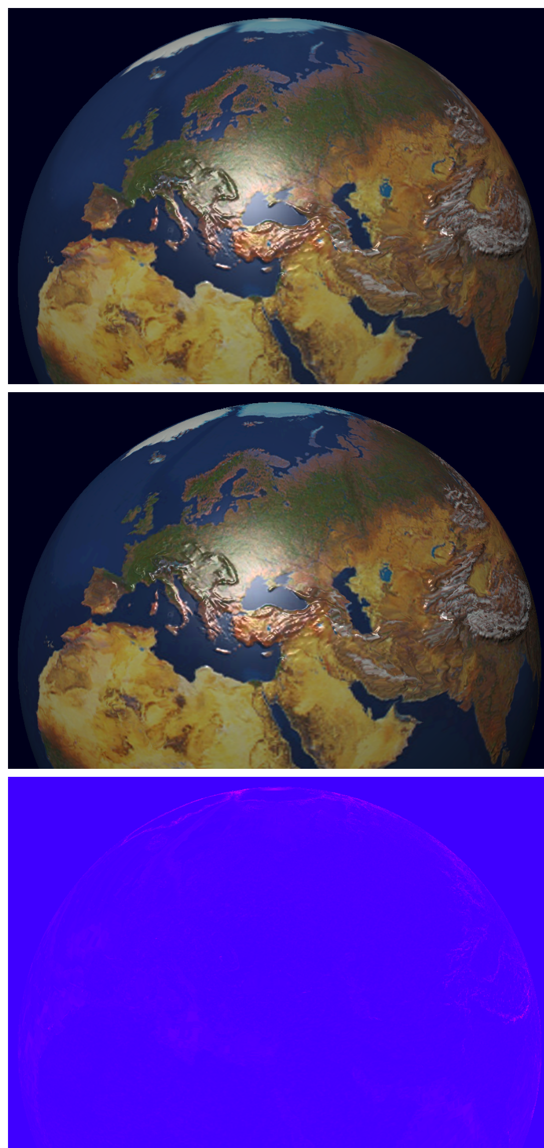
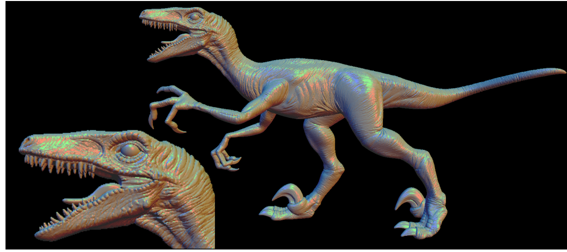
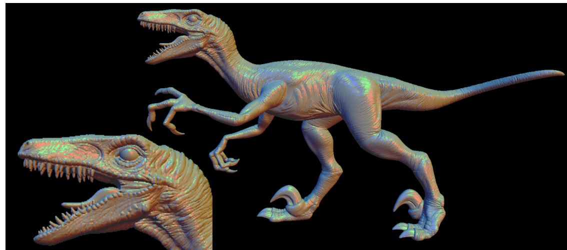


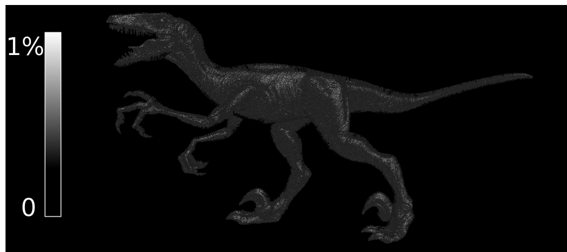
Figure 5: *Image du haut : programme de démonstration original de "bump mapping" par NVIDIA (couleur et normal sur 24 bits chacun). Image du milieu : même scène avec la carte de normales et la texture quantifiée avec notre approche (2 textures d'apparence en 16 bits). Image du bas : différence perceptuelle en fausse couleur (*Delta E - *CIE 1976) avec une erreur maximum de 17% (*$\Delta E = 83$*)*



(a) Surface originales - 288 bits/sommet



(b) Quantification normales couleurs sur GPU - 128 bits/sommet



(c) Erreur couleur - Image de différence x100

Figure 6: *Erreur observable de la quantification dans l'ombrage de Phong. Trois sources de lumières, surface spéculaire.*

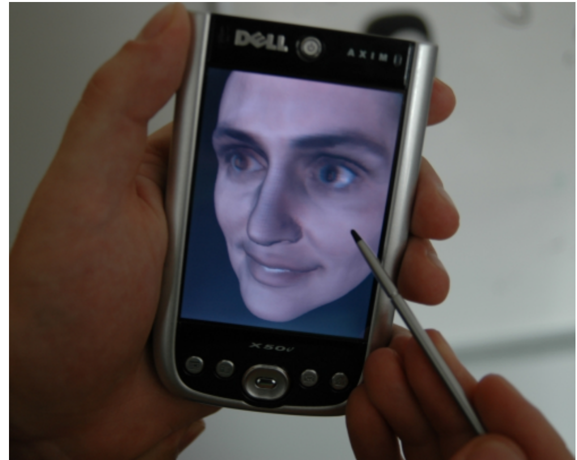


Figure 7: *La puissance des terminaux mobiles tels que les PDAs permet aujourd'hui de compter ces appareils parmi les clients potentiels d'une application de diffusion de contenu 3D. Ici, notre quantification est appliquée à un visage scanné, puis la déquantification et le rendu sont effectués sur un PDA avec OpenGL ES 1.1.*