Visualisation par surfels des textures volumiques

G. Guennebaud, M. Paulin
[guenneba|paulin]@irit.fr

IRIT-UPS 118 route de Narbonne 31062 Toulouse - Cedex 4

Résumé : *Dans cet article, nous présentons une méthode permettant la visualisation en temps réel de scènes complexes ayant un caractère répétitif. De telles scènes peuvent être avantageusement représentées par des textures volumiques. Malheureusement la technique de visualisation temps réel de celle-ci (adaptation de l'algorithme de Lacroute [LL94]) ne permet pas de gérer des motifs très précis (mémoire limitée des cartes graphiques). De plus, des artefacts apparaissent lors des changements de direction des tranches. Nous avons donc adapté à la visualisation des textures volumiques, une technique de rendu hybride entre le rendu à base d'images et le rendu à base de points, . Le motif de référence est alors représenté par un LDC Tree. Une technique permettant de visualiser une telle structure de données a déjà été proposée, mais la déformation des texels implique de revoir l'algorithme de projection ainsi que les différents tests de visibilités. L'intérêt d'une telle approche est de permettre la manipulation de motifs de références beaucoup plus précis, tout en accroissant la qualité du rendu.*

Mots-clés : Rendu à base d'images, Rendu à base de points, Textures volumiques, Temps réel.

1 Introduction

Quels que soient les outils utilisés lors de la modélisation d'un objet ou d'une scène, il est courant de passer par une représentation polygonale pour l'affichage. La principale raison est que ce type de représentation est directement exploitable et affichable par le matériel graphique. Par contre, le nombre de polygones générés est souvent très impressionnant pour des surfaces complexes (de l'ordre du million pour des arbres par exemple). A partir de cette remarque, il devient impossible de pouvoir visualiser en temps réel une forêt complète.

Un maillage de polygones est en fait une représentation très lourde à manipuler lorsque ceux-ci tentent d'approcher un objet complexe. En effet, la géométrie est donnée de manière beaucoup trop explicite et de plus, il existe un lien de connectivité beaucoup trop fort entre les primitives représentant l'objet. Cela rend la réduction du nombre de polygones très délicate dès que l'objet représenté devient trop complexe. Un algorithme de niveau de détails devient alors impraticable. Il devient donc intéressant de chercher une représentation alternative de ces objets complexes afin de rendre leur manipulation plus compacte et d'accélérer leur visualisation.

Quelques solutions ont déjà été proposées, avec en particulier le rendu à base d'images. L'objet est alors représenté par un ensemble d'images contenant au minimum une information colorimétrique mais aussi une information de profondeur et parfois même une information sur le matériau. A mi-chemin entre les représentations à base d'images et les représentations polygonales, on trouve ce que l'on appelle le rendu à base de points. Ici, l'objet est représenté par un ensemble d'échantillons ponctuels appartenant à sa surface ; on parle de surfels. On peut également citer les textures volumiques, qui permettent de représenter efficacement les scènes présentant un caractère répétitif, comme une forêt. Malheureusement la technique de visualisation temps réel de celle-ci présentent de nombreux inconvénients (voir section 3).

Afin de pallier à ces inconvénients, nous proposons une nouvelle manière de visualiser ces textures volumiques. Cela passe bien sûr par une nouvelle représentation des texels. Le choix s'est porté sur une représentation à base de points qui suscite un grand engouement depuis quelques années (voir section 2.1).

2 Travaux antérieurs

2.1 Rendu à base de points

La possibilité d'utiliser des particules pour rendre un objet solide a été initialement suggérée par M. Levoy et T. Whitted en 1985 dans [LW85] où ils présentent les problèmes fondamentaux comme la reconstruction de la

surface, la visibilité et le rendu de surfaces semi-transparentes. Mais ce n'est qu'en 1998 que J.P. Grossman et Dally reprirent cette idée [GD98] et formalisèrent pour la première fois l'utilisation de points comme primitive de rendu.

Les objets sont représentés par un ensemble d'échantillons ponctuels appartenant à leur surface. En fait, ces échantillons sont communément appelés surfels pour *"élément de surface"* (*"surface element"* en anglais). Ce terme a été introduit mathématiquement par Herman en 92 [Her92], mais Pfister et al. proposent une nouvelle définition plus adaptée à notre cas. D'après Pfister, un surfel est un n-tuple de dimension 0 avec des attributs de forme et de matière qui approxime localement la surface d'un objet. De plus, les surfels ne contiennent aucune connectivité explicite avec leurs voisins, ce qui en fait une représentation très souple à manipuler.

Pour ce qui est des structures de données, on peut citer la hiérarchie de sphères englobantes présentée avec le système QSplat dans [RL00] et le LDC Tree présenté avec les surfels dans [PZvBG00]. Ces deux structures possèdent deux caractéristiques fondamentales. La première est qu'elles sont multi-résolutions, ce qui permet d'ajuster le nombre de surfels à projeter en fonction de l'éloignement de la caméra. La deuxième est que les points sont stockés par blocs et de manière hiérarchique, ce qui rend ces tests de visibilité encore plus efficaces. Ces tests sont au nombre de trois et ont été proposés par Grossman [GD98] :

1. Un test classique de fenêtrage.
2. Un test basé sur les *"cônes de visibilité"* (équivalent de l'élimination des faces arrières).
3. Un test basé sur les *"masques de visibilités"* (gérant les problèmes d'auto occlusion).

Il existe deux manières d'aborder la reconstruction de la surface. Une première approche est de travailler dans l'espace image [GD98, PZvBG00]. Avec ce type d'algorithme, il est possible de prendre en compte des surfaces semi-transpentes (utilisation d'un A-buffer), par contre seule une implémentation logicielle est actuellement possible. Pour bénéficier d'une accélération par le matériel graphique il est nécessaire d'exprimer cette reconstruction dans l'espace objet [RL00, KV01]. Cependant, aucune des techniques précédentes ne supporte un anti-aliassage pour les modèles ayant une texture complexe. Pour répondre à ce manque, Pfister et al. reprennent le concept du filtrage pondéré elliptique de Heckbert [GH86] nommé EWA (Elliptical Weighted Average), et l'appliquent au splatting de surface en formulant les noyaux de reconstruction dans l'espace image [ZPvBG01]. Récemment Liu Ren et al. [RPZ02] parviennent à exprimer ces noyaux dans l'espace objet et tirent ainsi bénéfice d'une implémentation possible avec OpenGL.

2.2 Textures volumiques

Introduites par Kajiya et Kay en 1989 [KK89], puis développées par F. Neyret [Ney95], les textures volumiques utilisent 3 niveaux différents pour représenter l'information :

1. Les grandes variations, comme la surface d'une colline ou le dos d'un animal, sont codées par une description géométrique classique (mailles de polygones, carreaux de Bézier, surface NURBS, ...).
2. Le niveau de détail moyen, comme l'herbe ou les poils, qui sont concentrés au voisinage de la surface, est codé en utilisant un volume de référence stocké une seule fois et plaqué répétitivement, à la manière d'une texture 2D. Une instance de ce volume de référence est appelée texel.
3. Le niveau de détail fin, comme les microscopiques variations de chaque objet, sont codées par un modèle de réflexion stocké dans chaque voxel. Ce niveau correspond au niveau du pixel.

Il s'agit d'un modèle multi-échelle. Les textures volumiques peuvent être vue comme une extension des textures 2D traditionnelles. Au lieu de plaquer une image, forcément plane, on va plaquer une donnée volumique appelée texel. Ce plaquage est donc paramétré par les classiques coordonnées de texture 2D et par un vecteur de hauteur. Ce vecteur permet de déformer les texels en les coiffant par rapport à la surface par exemple (voir figure 1). Les texels forment une véritable couche sur la surface de la géométrie sous-jacente.

Au départ, la visualisation d'un texel passait par un algorithme de lancé de rayon. Le volume de référence étant modélisé par un octree, le rendu d'un texel ressemble beaucoup au rendu volumique. Mais, contrairement au rendu volumique, les texels ne contiennent pas une densité mais plutôt une probabilité d'occultation ainsi qu'un modèle de réflectance. Par la suite, la méthode de visualisation interactive de volume développée en 1994 par Lacroute et Levoy [LL94] a été adaptée aux textures volumiques [MN98]. Cette approche consiste à stocker le volume de référence par tranches. Trois séries de tranches sont extraites du volume dans trois directions orthogonales. Lors du rendu, une des piles de tranches est sélectionnée en fonction de la direction de visée. Les tranches sont ensuite

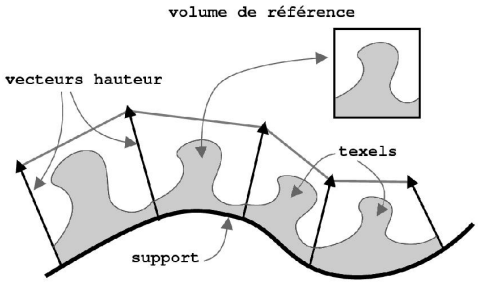


FIG. 1 – Spécification des textures volumiques

rendues par la carte graphique comme des polygones texturés. L'éclairage dynamique peut être pris en compte en stockant également des cartes de normales et en utilisant la même technique que pour le bump-mapping. L'utilisation du mipmapping permet de conserver le caractère multi-résolution des textures volumiques. Remarquons que les dernières cartes graphiques permettent l'utilisation de texture 3D. Il n'est donc plus nécessaire de stocker trois fois le volume de référence.

Un dernier point concerne l'animation des textures volumiques. En effet, il est possible d'animer les texels de trois manières différentes (et indépendantes) :

1. Déformer la surface sur laquelle se trouvent les texels.
2. Déformer l'orientation des vecteurs de hauteur se trouvant sur chaque sommet de la surface (par exemple pour simuler du vent dans l'herbe).
3. Animer le volume de référence à la manière d'un dessin animé (pré-calcul d'une série de volumes de référence).

Par contre, l'animation du volume de référence reste quand même très coûteuse en mémoire.

3 Une nouvelle représentation des textures volumiques

L'idée de proposer une nouvelle méthode de visualisation temps réel des textures volumiques vient du fait que la méthode initialement proposée présente quelques limites : artefacts lors des changements de tranches, impossibilité d'obtenir un ombrage correct, animation du texel de référence quasi impossible. Mais surtout, le volume de référence est stocké entièrement, et même plusieurs fois. La mémoire texture des cartes graphiques étant limitée, on est restreint à des volumes de 128^3 voir 256^3 (en compressant les données stockées).

Il nous faut donc trouver une nouvelle représentation plus compacte, tout en conservant l'aspect multi-résolution. Nous avons choisi une approche par surfels qui semble bien appropriée puisque seuls les points appartenant à la surface de l'objet représenté sont conservés. De plus, en utilisant une structure de données tel que le LDC Tree nous avons une représentation complètement multi-résolution de notre motif de référence. Le choix du LDC Tree est d'autant plus judicieux qu'il permet d'avoir une approche incrémentale de la projection des surfels, ce qui est impossible avec la hiérarchie de sphère englobante.

Nos techniques d'aquisition et de rendu d'un LDC Tree sont largement inspirées des travaux de Pfister et al. et de ceux de Grossman. Nous allons donc nous contenter de présenter ces deux phases dans les grandes lignes (sections 4 et 6) afin de nous concentrer sur l'adaptation aux textures volumiques (section 5).

4 Le LDC Tree et son acquisition

4.1 Contenants de base : les LDI

LDI est une abréviation pour « Layered Depth Image » [SGS98]. Comme son nom l'indique, il s'agit d'images stockées en couches, où chaque pixel contient une information de profondeur. Cette information de profondeur est relative au modèle de la caméra liée au plan image. Dans notre cas il s'agit d'une caméra à projection parallèle.

Un LDI est donc une matrice 2D (une image) où chaque pixel stocke la liste des intersections entre le rayon lui correspondant et la scène (figure 2). D'où la notion de couche. Dans notre cas, il s'agit d'une liste de surfels.

Les avantages de ce stockage sont multiples. Tout d'abord, certains attributs des surfels deviennent implicites, comme leurs coordonnées (x,y) et leur diamètre, ce qui entraine une certaine **compacité**. Un second point concerne l'**efficacité**. En effet, le stockage des surfels dans une grille régulière permet d'avoir une approche incrémentale de la projection [Gro98].

4.2 Pour un meilleur échantillonnage : le LDC

En fait, une seule LDI ne permet pas de représenter correctement tout un objet. En effet, les zones de la surface tangentes par rapport à la direction z de la LDI sont très mal échantillonnées. Une solution possible est d'utiliser trois LDI orthogonales entre elles. Cet arrangement est appelé un LDC, abréviation de « Layered Depth Cube », et est illustré figure 2. Un LDC est aussi appelé bloc.

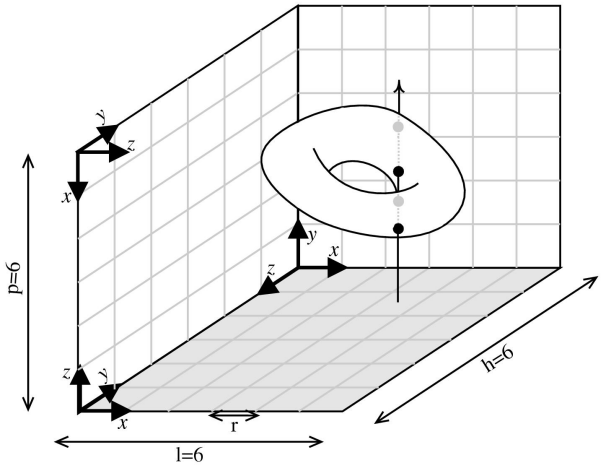


FIG. 2 – Un LDC composé de 3 LDI avec leur repère (i, j, k) respectif. Le rayon associé au pixel $(3, 4)$ de la LDI grisée intersecte 4 fois l'objet, ce pixel contient donc 4 surfels. Sont également mentionnées la largeur l, la hauteur h ainsi que la profondeur p du bloc. Une des trois LDI joue donc un rôle particulier : son repère est aussi celui du bloc.

4.3 Hiérarchique et multi-résolution : le LDC tree

Le LDC tree peut être vu comme une sorte d'octree ou chaque noeud est un LDC (ou plus généralement un bloc). Dans cette section nous allons voir comment construire une telle hiérarchie.

Pour la construction d'un LDC tree, nous partons d'un grand LDC contenant entièrement l'objet voulu. Ce LDC est subdivisé en un certain nombre de petits blocs de taille b^3 (on suppose, pour simplifier, que les blocs sont cubiques, donc $l = h = p = b$). Ces blocs forment les feuilles de l'octree. Le niveau supérieur est construit en fusionnant 8 blocs adjacents (fils) pour former un seul bloc (père) de même taille b^3 mais dont l'espace r entre les pixels est double. Pour la fusion des fils, notre approche est différente de celle de Pfister. Rappelons qu'ils se contentaient de recopier les adresses des listes de surfels d'un pixel sur deux. C'est pour cela qu'ils utilisent plusieurs niveaux de texture. Dans notre cas, un bloc temporaire de taille $(2b)^3$ est créé par concaténation des huit fils. Puis, la réduction de la résolution est réalisée, comme pour une image, en faisant la moyenne des pixels des LDI quatre par quatre. Mais ici les pixels contiennent une liste de surfels. Les surfels des quatre listes à fusionner sont d'abord mis en correspondance (en comparant leur valeur z) puis fusionnés en faisant les moyennes de leurs attributs (profondeur, couleur, normale). Les niveaux supérieurs de la hiérarchie sont construits ainsi de suite jusqu'à obtenir un seul bloc, la racine, représentant à lui seul tout l'objet mais à une très faible résolution.

5 Des surfels aux textures volumiques

Dans cette section nous allons voir comment adapter le rendu classique d'un LDC Tree lorsque celui-ci représente un texel. Pour cela nous proposons une nouvelle manière d'exprimer la déformation des texels (section 5.1) qui va nous permettre d'intégrer cette déformation au sein de notre propre algorithme de projection incrémentale (section 5.2). Enfin, nous aborderons le problème de la visibilité en essayant d'adapter les tests déjà proposés (section 5.3).

5.1 Déformation des texels

La forme et position d'un motif appliqué à la surface d'un objet est complètement définie par la donnée de la position des 4 coins du texel en contact avec la surface de l'objet et des 4 vecteurs hauteurs en ces points (voir figure 3).

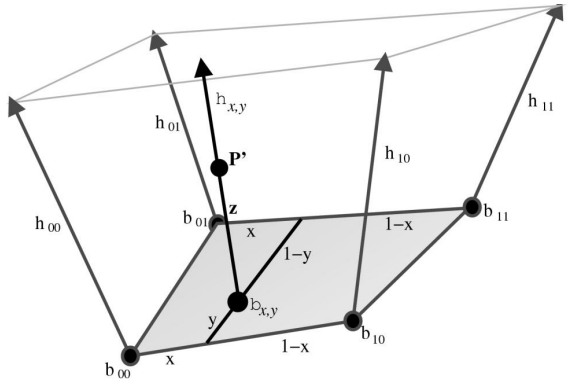


FIG. 3 – Paramétrisation de la déformation d'un texel par les quatres sommets $b_{i,j}$ et les vecteurs hauteurs $h_{i,j}$.

Cette déformation ne peut pas être exprimée à l'aide d'une matrice de transformation. En effet, on ne peut la décomposer en une succession de rotations, mises à l'échelle et translations. Nous allons donc voir comment elle peut être exprimée. Soit p un point du motif de référence de coordonnée (x, y, z) exprimées dans le repère local du texel. Nous supposons de plus que le motif est normalisé, c'est à dire de largeur 1, de hauteur 1 et de profondeur 1. Soit $\widetilde{p}$, l'image de p par la déformation. Le calcul de $\widetilde{p}$ peut être effectué de la manière suivante :

Calcul de $b_{x,y}$ (resp. $h_{x,y}$) résultat de l'interpolation bilinéaire des $b_{i,j}$ (resp. $h_{i,j}$), $(i,j) \in [0..1]^2$), par les coefficients x et y :

$$\begin{aligned} b_{x,y} &= lerp\left(y, lerp\left(x, b_{00}, b_{10}\right), lerp\left(x, b_{01}, b_{11}\right)\right) \\ h_{x,y} &= lerp\left(y, lerp\left(x, h_{00}, h_{10}\right), lerp\left(x, h_{01}, h_{11}\right)\right) \end{aligned} \tag{5.1}$$

Où $lerp$ est la fonction d'interpolation linéaire : $lerp(t, A, B) = (1-t)\,A + t\,B$. On a alors :

$$\widetilde{p} = b_{x,y} + z\,h_{x,y} \tag{5.2}$$

A ce stade, plusieurs remarques s'imposent. Tout d'abord, le calcul de $b_{x,y}$ et $h_{x,y}$ peut être réalisé par la géométrie support des texels, dans le cas de surface paramétrique notamment. Dans ce cas, le résultat d'un motif déformé ne correspond pas à la figure 3 et le motif épousera complètement la surface. D'autre part, dans le cadre de la figure 3, il est possible de considérer le carreau $(b_{00}, b_{10}, b_{01}, b_{11})$ comme un carreau de Bézier à partir duquel il est facile de calculer $b_{x,y}$ et $h_{x,y}$. Cela a pour conséquence de lisser la surface de l'objet. Dans la suite, nous allons rester dans le cadre de la figure 3, les deux possibilités précédentes pouvant être facilement adaptées par la suite.

5.2 Algorithme de projection

Commençons par remarquer que la forme d'un bloc au sein du motif déformé est similaire à celle du motif tout entier représenté figure 3. La position des quatres coins $b_{i,j}$ et les vecteurs hauteurs $h_{i,j}$ d'un bloc sont obtenus

facilement à partir de ses paramètres (position au sein du motif et dimensions) et des équations 5.2 et 5.1. Nous sommes donc en mesure de projeter les surfels d'un bloc : soit (x, y, z) les coordonnées du $k^{ième}$ surfel $S_{i,j,k}$ stocké en (i, j) dans la LDI. A partir de l'équation 5.2, on obtient $(\widetilde{x}, \widetilde{y}, \widetilde{z})$, coordonnées de $S_{i,j,k}$ dans le repère objet et après déformation. Il ne reste plus qu'à appliquer la transformation de modélisation et la projection en perspective conique pour obtenir les coordonnées (u, v) de $S_{i,j,k}$ dans l'espace image. Par contre, l'implémentation directe de cette méthode n'est pas vraiment envisageable, puisqu'elle demande un nombre bien trop important d'opérations par surfel à projeter.

Dans les sections précédentes, nous avons vu qu'il était possible de tirer bénéfice de la cohérence spatiale entre les surfels due à leur stockage dans une grille régulière. Nous allons donc tenter la même approche que Grossman, mais en tenant compte de la déformation du bloc. De plus, nous allons considérer uniquement le LDI des blocs dont la référence correspond au carreau $(b_{00}, b_{10}, b_{01}, b_{11})$. Pour les autres, il suffit de faire subir une rotation à la figure 3, le carreau $(b_{00}, b_{10}, b_{01}, b_{11})$ ne se retrouvant plus appliqué à la surface de l'objet.

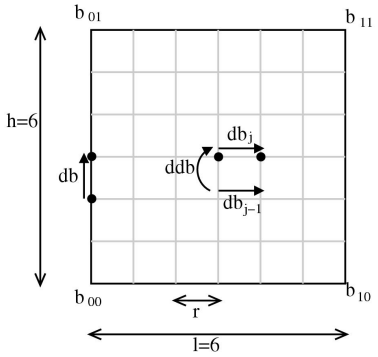


FIG. 4 – Illustration des différents incréments. Ce shéma s'applique aussi bien au vecteur b, qu'aux vecteurs h, q et r.

Afin de simplifier l'écriture des formules, nous allons nous placer dans une ligne j et considérer un seul surfel par "pixel" (i, j). Nous considérons donc le surfel S_i, de coordonnées (x_i, y_i, z_i) dans le bloc :

$$\begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} = o + r \begin{bmatrix} i \\ j \\ \delta_{i,j,k} \end{bmatrix} \tag{5.3}$$

Où, o est l'origine du bloc courant dans le repère du LDC Tree, r la distance entre deux pixels et $\delta_{i,j,k}$ la profondeur du $k^{ième}$ surfel stocké en (i, j). Il est donc intéressant d'insérer la translation par o et la mise à l'échelle par r dans la matrice de modélisation active ; soit M la matrice 4x4 résultante que l'on décompose en une matrice 3x3 orthogonale A (représentant une rotation et mise à l'échelle) et un vecteur de translation T. Pour simplifier l'écriture, posons : $b_i = b_{x_i,y_i}$ et $h_i = h_{x_i,y_i}$. Dans le repère de la caméra, on a :

$$\begin{bmatrix} x'_i \\ y'_j \\ z'_j \end{bmatrix} = A \begin{bmatrix} \widetilde{x}_i \\ \widetilde{y}_i \\ \widetilde{z}_i \end{bmatrix} + T = A\,(b_i + z_i\,h_i) + T \tag{5.4}$$

Soit (u_i, v_i) les coordonnées de S_i dans l'espace image :

$$u_i = \left\lfloor \frac{L}{2}\left(1 - \frac{x'_i}{z'_i}\right)\right\rfloor \qquad v_i = \left\lfloor \frac{L}{2}\left(1 - \frac{y'_i}{z'_i}\right)\right\rfloor \tag{5.5}$$

Où, L est la taille de l'image et $\lfloor . \rfloor$ renvoie la partie entière. En insérant 5.2 dans 5.5 et en posant :

$$q_i = A\,b_i + T \qquad \text{et} \qquad r_i = A\,h_i$$

On obtient finalement :

$$u_i = \left\lfloor \frac{L}{2}\left(1 - \frac{(p_i)_x + z_i\,(q_i)_x}{(p_i)_z + z_i\,(q_i)_z}\right)\right\rfloor \qquad v_i = \left\lfloor \frac{L}{2}\left(1 - \frac{(p_i)_y + z_i\,(q_i)_y}{(p_i)_z + z_i\,(q_i)_z}\right)\right\rfloor \tag{5.6}$$

Lors du parcours d'une ligne j (voir figure 4, on a donc :

$$q_i = q_{i-1} + \mathrm{d}q_j \qquad r_i = r_{i-1} + \mathrm{d}r_j \tag{5.7}$$

Où les deux incréments introduits sont égaux à :

$$\mathrm{d}q_j = A\,\mathrm{d}b_j \qquad \mathrm{d}r_j = A\,\mathrm{d}h_j$$

L'indice j des incréments est là pour signifier qu'il dépend de la ligne parcourue. Par la suite, on introduira donc des incréments de saut ligne pour ces incréments. $\mathrm{d}b_j$ (resp. $\mathrm{d}h_j$) est l'incrément permettant de passer de b_{i-1} à b_i (resp. de h_{i-1} à h_i). Eux aussi dépendent de la ligne parcourue et nous verrons qu'il n'est pas nécessaire de les calculer explicitement. Maintenant, voyons comment passer d'une ligne à la suivante. Tout d'abord, que se passe-t-il pour les vecteurs q et r :

$$q_{0,j} = q_{0,j-1} + A\,\mathrm{d}b \qquad r_{0,j} = r_{0,j-1} + A\,\mathrm{d}h \tag{5.8}$$

Où, $\mathrm{d}b$ (resp. $\mathrm{d}h$) est l'incrément permettant le calcul de $b_{x,y}$ (resp. $h_{x,y}$) lors du passage d'une ligne à la suivante avec $i = 0$, c'est à dire :

$$b_{x_0,y_{i-1}} = b_{x_0,y_i} + \mathrm{d}b \qquad h_{x_0,y_{i-1}} = h_{x_0,y_i} + \mathrm{d}h$$

Avec :

$$\mathrm{d}b = \frac{b_{01} - b_{00}}{h} \qquad \mathrm{d}h = \frac{h_{01} - h_{00}}{h}$$

Où, rappellons le, h est la hauteur du bloc. Il ne reste plus qu'à incrémenter les incréments $\mathrm{d}q_j$ et $\mathrm{d}r_j$:

$$\mathrm{d}q_j = \mathrm{d}q_{j-1} + A\,\mathrm{dd}b \qquad \mathrm{d}r_j = \mathrm{d}r_{j-1} + A\,\mathrm{dd}h \tag{5.9}$$

Ici, $\mathrm{dd}b$ (resp. $\mathrm{dd}h$) est l'incrément de l'incrément $\mathrm{d}b_j$ (resp. $\mathrm{d}h_j$). Pour illustrer tous ces incréments on peut se référer à la figure 4. Ces derniers sont obtenus par :

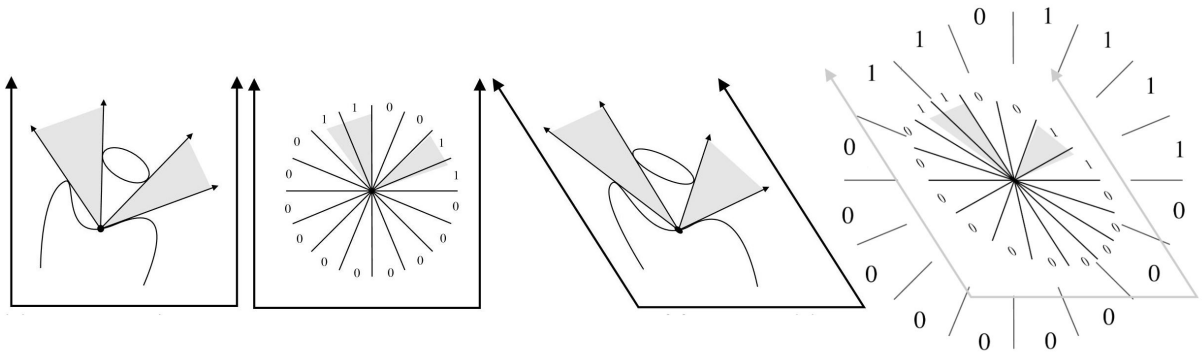
$$\mathrm{dd}b = \frac{b_{11} - b_{01} - b_{10} + b_{00}}{l\,h} \qquad \mathrm{dd}h = \frac{h_{11} - h_{01} - h_{10} + h_{00}}{l\,h}$$

Pour résumer, après initialisation des différents constantes et variables, le rendu d'un bloc s'effectue très simplement et à peu de frais. Un incrément selon i est réalisé par les équations 5.7, (2 additions vectorielles). La projection d'un surfel, equations 5.6, nécessite 2 multiplications vectorielles, 1 inversion et 2 additions vectoriels. Le passage d'une ligne à suivante est réalisé par les equations 5.8 et 5.9 pour un total de 4 additions vectorielles. Bien sùr, la prise en compte de la déformation ajoute un coût de calcul supplémentaire mais qui, au final, reste tout à fait raisonnable, d'autant plus qu'il s'agit de calcul vectoriel et donc directement optimisable par l'utilisation des instructions flottantes SIMD des derniers processeurs PC.

5.3 Visibilité

Nous venons de voir que la prise en compte de la déformation du LDC Tree nécessite de revoir à la base la projection des surfels. Maintenant qu'en est-il des tests de visibilité ?
Pour le fenêtrage, pas de problème, il suffit de calculer la boite englobante du bloc une fois déformé.
Pour ce qui est des masques et cônes de visibilité, les choses se compliquent. Ici nous allons nous contenter de présenter le problème pour les masques de visibilité, le cas des cônes de visibité étant semblable. Une description détaillé de leur fonctionnement peut être trouvé dans [Gro98], mais rapellons tout de même qu'un masque de visibilité est un masque de n bits (typiquement $n = 128$) où chaque bit correspond à un triangle résultant de la subdivision régulière de la sphère en n triangles. Un masque est calculé pour chaque bloc de tel sorte que $k^{ième}$ bit vaut 1 si et seulement si le bloc est visible (à partir d'un point de vue hors de l'enveloppe convexe de l'objet) d'une direction correspondant au $k^{ième}$ triangle. Au moment du rendu, un masque est calculé pour le volume de visualisation. Il suffit alors de réaliser un ET logique entre les deux masques pour savoir si le bloc est visible ou non.

La difficulté est qu'il est tout à fait possible que, pour une position et orientation de la caméra données, une partie du motif soit caché par lui-même et devienne tout à fait visible suite à la déformation. Cette partie, équivalente



(a) (b) (c) (d)

FIG. 5 – Illustration de la déformation des masques de visibilité (en 2D). (a) motif non déformé avec en grisé les directions à partir desquelles le bloc matérialisé par un point est visible. (b) masque de visibilité associé au bloc. (c) déformation du masque de visibilité et reprojection sur la partition de l'espace des directions. (d) une fois le motif déformé, les directions valides pour le bloc correspondent bien au masque précédemment calculé. Masque avant la transformation : 0011 0000 0000 0011 Masque après la transformation : 1111 0000 0000 0110

à un bloc, risque alors d'être éliminée par le test des masques de visibilité. Cela vient du fait que l'espace des directions est aussi perturbé par la déformation. Il faut donc faire subir une transformation au masque avant le test. Cette transformation peut être calculée de la manière suivante. Tout d'abord, la pseudo-sphère représentant les 128 partitions de l'espace des directions est positionnée au centre du bloc. Des 128 triangles approximant la sphère, seuls ceux correspondant aux bits du masque égaux à 1 sont conservés. Puis, la déformation (équations 5.2 et 5.1) est appliquée aux sommets des triangles qui sont ensuite reprojetés sur la pseudo-sphère afin d'en déduire le nouveau masque (figure 5).

Si en théorie tout cela semble fonctionner, le passage à la pratique reste problématique car la démarche qui vient d'être exposée est beaucoup trop lourde en terme de coût de calcul. Un test de visibilité doit être extrêmement rapide à exécuter pour être valide, sinon il ne fait que ralentir le processus de rendu. L'idée pourrait être de prendre en compte la cohérence temporelle de la déformation des texels. En effet, il est raisonnable de faire l'hypothèse que la forme d'un texel varie lentement au cours du temps. Il est alors possible de stocker, pour chaque bloc de chaque texel, le nouveau masque de visibilité et de les mettre à jour seulement lorsque cela est nécessaire. On se retrouve alors avec un problème de coût mémoire dès que le nombre de texels est grand. Rappelons que dans la pratique seul un texel servant de référence est stocké puis instancié et déformé lors du rendu.

Une implémentation efficace des masques de visibilité prenant en compte la déformation du motif serait d'autant plus souhaitable que les textures volumiques permettent d'en exprimer toute la puissance. En effet, prenons l'exemple d'une forêt modélisée avec l'aide des textures volumiques. Il est alors astucieux de construire un motif représentant plusieurs arbres. Par exemple, un au centre et un deuxième réparti dans les quatre coins. En calculant les masques de visibilité sur un tel motif, ceux-ci sont alors capables de gérer les interactions avec les arbres voisins et pas seulement pour l'arbre lui-même.

6 Rendu

6.1 Calcul de l'éclairage

Le calcul correct de l'éclairage nécessite tout d'abord d'appliquer la déformation également aux normales. Exprimer quelle est la transformation que subit la normale d'un surfel quelconque du texel est loin d'être évident. Nous prosons donc une astuce qui consiste à prendre un point situé sur la normale du surfel courant et à une distance ϵ (ϵ étant suffisement petit) de celui-ci. Soit p les coordonnée du surfels courant, $\vec{N}$ sa normale et p_ϵ le point correspondant. On a :

$$p_\epsilon = \epsilon \vec{N} + p \qquad \text{d'ou :} \qquad \widetilde{p_\epsilon} = b_{\epsilon N_x + p_x, \epsilon N_y + p_y} + (\epsilon N_z + p_z)\ h_{\epsilon N_x + p_x, \epsilon N_y + p_y} \tag{6.1}$$

Le calcul direct de $\widetilde{p_\epsilon}$ avec les équations 5.1 et 5.2 est possible mais couteux. Cela dit, comme pour l'algorithme de projection, il est possible d'avoir une approche incrémentale de ce calcul :
En remarqant que :

$$lerp(t+\epsilon, A, B) = lerp(t, A, B) + \epsilon\,(B - A) \text{ et } lerp(t, A+a, B+b) = lerp(t, A, B) + lerp(t, a, b) \quad (6.2)$$

On arrive à :

$$\widetilde{p_\epsilon} = \widetilde{p} + R_b + \epsilon\, N_z\, h_{p_x, p_y} + (p_z + \epsilon\, N_z)\; R_h \quad (6.3)$$

Avec :

$$R_b = \epsilon\, N_y\, lerp\,(p_x, (b_{01} - b_{00}), (b_{11} - b_{10}))\, \epsilon\, N_x\, lerp\,(p_y, (b_{10} - b_{00}), (b_{11} - b_{01})) + \epsilon^2\, N_x\, N_y\; (b_{11} + b_{00} - b_{01} - b_{10}) \quad (6.4)$$

R_h étant calculé de la même manière. Le vecteur normal $\widetilde{n}$ non normalisé est alors égal à :

$$\widetilde{\vec{n}} = \widetilde{p} - \widetilde{p_\epsilon} = R_b + \epsilon\, N_z\, h_{p_x, p_y} + (p_z + \epsilon\, N_z)\; R_h \quad (6.5)$$

Au premier abord cette expression parait beaucoup plus compliquée que le calcul direct de $\widetilde{p_\epsilon}$ avec les équations 5.1 et 5.2. Mais en fait, de nombreux termes sont des constantes qui peuvent être précalculées et les interpolations linéaires apparaissant dans les expressions de R_b et R_h peuvent être calculées de manière incrémentale. De plus, le calcul de h_{p_x, p_y} est déjà réalisé lors de la projection incrémentale du surfel (section 5.2) et les $3^ième$ terme en ϵ^2 dans les expressions de R_b et R_h sont négligeables. Finalement, le nombre de multiplication a été divisé par 2 par rapport au calcul direct de $\widetilde{p_\epsilon}$. Il reste le problème du choix de la valeur à donner à ϵ. Dans la pratique, prendre ϵ égale à la distance entre deux pixels du bloc courant semble être un bon choix.

Le calcul de l'éclairage est alors effectué une fois que tous les surfels ont été projetés, mais avant la reconstruction comme dans [GD98]. En fait, n'importe quel modèle d'illuminaton peut être utilisé, tout dépend de la manière dont les matériaux sont représentés. Pour faire simple et efficace, nous utilisons un modèle de Phong. Pour ce qui est des ombres portées, celle-ci peuvent être calculées très simplement par la technique des *"shadows maps"* [WTRDHS87].

6.2 Reconstruction de la surface

Pour ce qui est de la reconstruction de la surface, nous avons simplement adapté la méthode de Grossman à notre problème. Si cette méthode est loin d'être la meilleure en terme de qualité, elle a comme avantages d'être simple et très rapide. Ici, les surfels sont projetés sur un seul pixel de l'image. La détection des trous est alors réalisée par une hiérarchie de Z-buffer à résolution décroissante. Lors de la projection d'un bloc, le z-buffer ayant une résolution suffisement faible pour qu'il n'y ait pas de trou est activé et utilisé conjointement avec le z-buffer qui est à la résolution de l'image. Une fois que tous les surfels sont projetés, cette hiérarchie de z-buffer va permettre de filtrer les pixels situés en avant plan de ceux situés en arrière plan (en comparant la profondeur du pixel avec la profondeur correspondante stocké par les z-buffer de résolution inférieure). Le résultat est un poids compris entre 0 et 1 qui est affecté à chaque pixel. Ce sont ces poids qui vont permetre la reconstruction des *« trous »* par interpolation, celle-ci étant réalisée par un algorithme dit "pull-push".

6.3 Intégration avec une scène OpenGL

Puisqu'il ne semble pas raisonnable de modéliser une scène complète uniquement avec une représentation ponctuelle (que ce soit par l'intérmédiaire des textures volumiques ou non), et que l'algorithme de rendu présenté est indépendant de toute bibliothèque graphique, il serait intéressant de pouvoir intégrer nos objets dans une scène rendue par OpenGL. La solution est de récuperer le tampon de profondeur après le rendu OpenGL et avant la projection de nos surfels. Il serait également possible de récuper en même temps le tampon chromatique, mais il est plus efficace de mettre à jour une texture OpenGL à partir de l'image résultant de la reconstruction, puis de l'afficher en rendant un polygone recouvrant tout l'écran. Lors de ce dernier rendu et selon les capacités de la carte graphique, il est possible de faire réaliser par cette dernière de nombreuses opérations couteuses :

– Calcul de l'éclairage dans le cas où nos objets sont diffus et que la source de lumière est située à l'infini (cas d'une forêt en plein jour) : utilisation d'une deuxième texture représentant la carte des normales (reconstruite en même temps que la carte colorimétrique) et des *pixels shaders*).
– Normalisation des normales par une texture cubique et l'utilisation des *"textures shaders"*.
– Gérer la visibilité entre nos objets et les objets rendu avec OpenGL en utilisatant les *"textures shaders"* et une autre texture représentant notre carte de profondeur. Il n'est alors plus nécessaire de récupérer le tampon de profondeur calculé par OpenGL (opération coûteuse).
– Calcul de l'éclairage avec des matériaux non nécessairement diffus et des sources ponctuels grace aux *"fragments shaders"* disponible sur la futur NV30.

7 Résultats

Les tests ont été réalisés sur un Atlhon 1Ghz disposant de 256Mo de mémoire et d'une carte graphique GeForceII MX. Pour ce qui est de l'implémentation, notons que seule la partie concernant la projection incrémentale d'une LDI (et la transformation et normalisation des normales) a été optimisée par l'utilisation du jeu d'instructions 3DNow !. L'algorithme de reconstruction traitant des données vectorielles (couleurs RGBA et normales), on peut estimer que l'utilisation du jeu d'instructions SSE diminurait les temps de reconstruction par 4. Les images de la figures 6 ont été obtenues à partir d'un motif contenant un seul arbre et d'une résolution de 512^3 (les feuilles du LDC Tree sont au nombre de 64^3 et contiennent trois LDI d'une résolution de 8^2). Cet arbre a été habillé avec la méthode présentée par Maritaud dans [MDG00] et est représenté par 1.2 millions de surfels. Au niveau du coût de stockage, ce motif nécessite à peu près 30Mo d'espace disque (2 millions de surfels et 40 milles noeuds pour le LDC Tree entier). La scène complète (2000 arbres) nécessitent de 0.3s à 1s (selon le point de vue) de temps de calcul réparti de la manière suivante :

– Recupération du tampon de profondeur : 0.05s
– Reconstruction : de 0 à 0.6s selon la distance entre les surfels dans l'espace image. :
  – 70% pour la phase de recherche des trous et de calcul des poids.
  – 30% pour l'algorithme pull-push
– Rendu des LDC Tree :
  – 10% pour le parcours de la hiérarchie incluant les tests de visibilité et l'initialisation de la projection incrémentale d'une LDI.
  – 90% pour la projection des surfels
– Calcul de l'illumination et rendu OpenGL : 0.024s

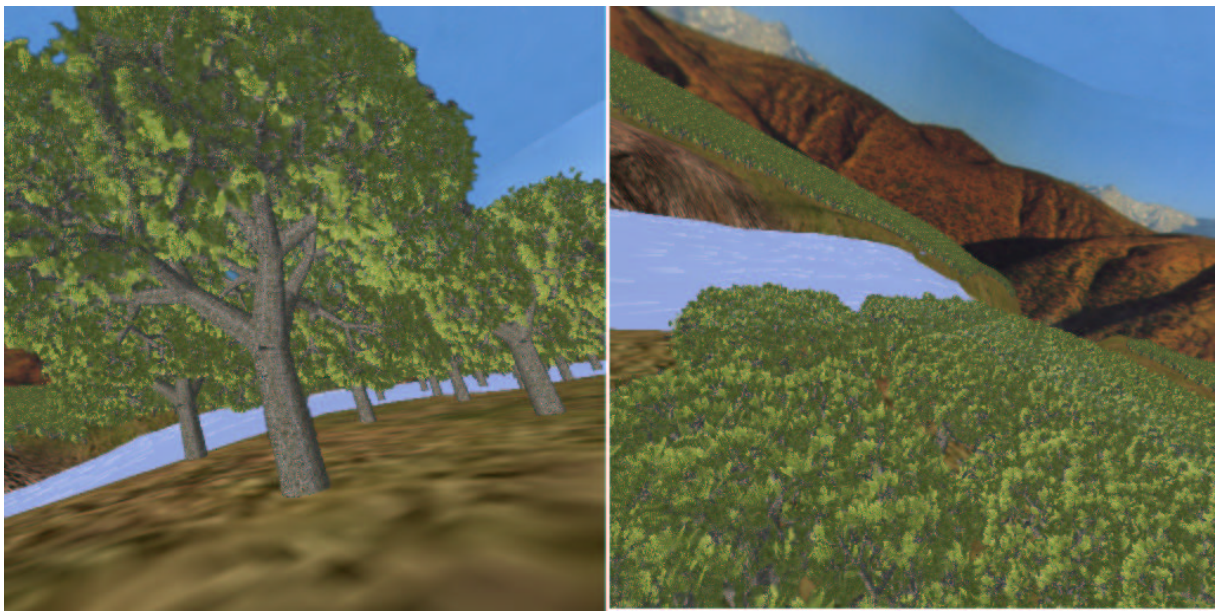


FIG. 6 – Images types

8 Conclusion

Le principal intérêt de cet article est de montrer qu'une représentation par surfels est particulierement bien adaptée à la visualisation des textures volumiques. Cependant, il reste encore de nombreux points à améliorer. Le premier concerne les tests de visibilité. Comme nous l'avons vu, il semble très difficile d'adapter de manière efficace les tests déjà proposés par Grossman. Il serait donc interressant de proposer de nouveaux tests qui soient plus adaptés aux objets subissant une déformation comme les textures volumiques ou bien les objets animés par squelette... Un second point est que l'algorithme de reconstruction utilisé ici est d'assez mauvaise qualité. Son gros avantage est qu'il n'y a aucune rastérisation des surfels projetés, ce qui permet une implémentation logicielle très efficace. Actuellement, le meilleur algorithme de reconstruction est celui proposé par Pfister et al. qui s'appuie sur le filtrage EWA [ZPvBG01] (c'est le seul à proposer un filtrage anisotrope). De plus, une implémentation utilisant le matériel graphique a déjà été proposée [RPZ02]. L'avenir passe donc sans doute par l'utilisation de cette technique de reconstruction. A priori, cela ne semble pas trop difficile puisque la déformation des texels peut facilement être implémentée au niveau des *vertex programmes* au prix d'une trentaine d'instructions supplémentaires.

Références

[GD98] J. P. Grossman and W. J. Dally. Point sample rendering. *Rendering Techniques 98*, pages 181–192, June 1998.

[GH86] N. Greene and P. Heckbert. Creating raster omnimax images from multiple perspective views using the ellipticalweighted average filter. *IEEE Computer Graphics & Applications*, 6(6) :21–27, June 1986.

[Gro98] J. P. Grossman. Point sample rendering. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, August 1998.

[Her92] G. T. Herman. Discrete multidimensional jordan surfaces. *CVGIP : Graphical Modeling and Image Processing*, 54(6) :507–515, November 1992.

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In *Proceedings of SIGRAPH 89*, volume 23, pages 271–280, August 1989.

[KV01] Aravind Kalaih and Amitabh Varshney. Differential point rendering. In *Proceedings of 12th Eurographics Workshop on Rendering*, pages 139–150, June 2001.

[LL94] Philipe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of SIGRAPH'94*, pages 451–458, July 1994.

[LW85] Marc Levoy and Turner Whitted. The use of points as display primitive. Technical Report TR 85-022, 1985.

[MDG00] K. Maritaud, J.M. Dischler, and D. Ghazanferpour. Rendu réaliste d'arbres à courte distance. In *AFIG'00*, December 2000.

[MN98] Alexandre Meyer and Fabrice Neyret. Textures volumiques interactives. In *AFIG'98*, pages 261–270, Dec 1998.

[Ney95] Fabrice Neyret. A general and multiscale model for volumetric textures. In Wayne A. Davis and Przemyslaw Prusinkiewicz, editors, *Graphics Interface '95*, pages 83–91, May 1995.

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels : Surface elements as rendering primitives. In *Proceedings of SIGGRAPH 2000*, pages 335–342, July 2000.

[RL00] S. Rusinkiewicz and M. Levoy. Qsplat : A multiresolution point rendering system for large meshes. In *Proceedings of SIGGRAPH'2000*, pages 343–352, July 2000.

[RPZ02] L. Ren, H. Pfister, and M. Zwicker. Object space ewa surface splatting : A hardware accelerated approach to high quality point rendering. In *Proceedings of Eurographics 2002.*, 2002.

[SGS98] Jonathan Shade, Steven J. Gortler, and Richar Szeliski. Layered depth image. In *Proceedings of SIGRAPH 98*, pages 231–242, July 1998.

[WTRDHS87] Robert L. Cook William T. Reeves David H. Salesin. Rendering antialiased shadows with depth maps. *Computer Graphics (SIGGRAPH 87 Proceedings)*, 21(4) :283–291, July 1987.

[ZPvBG01] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proceedings of SIGGRAPH 2001*, August 2001.