Benoît Bossavit
Stéphane Romero-Romero
Frédérick Roy
Romain Vergne

Mémoire

Rendu 3D temps-réel de grands objets

Vendredi 07 Avril 2006

Chargé de TD : Pascal Desbarats
Client : Tamy Boubekeur

Université Bordeaux I

Résumé

Les applications actuelles utilisent des objets 3D de plus en plus grands. Bien que la technologie évolue sans cesse, la manipulation de ces objets, en particulier leur rendu temps-réel, reste un problème. L'objectif de ce projet est de mettre en place un moteur multirésolution capable d'afficher des objets de plusieurs millions de polygones. Dans un premier temps, nous implémenterons l'algorithme développé dans l'article [DVS03]. Celui-ci présente une structure de données optimisée pour le GPU permettant le rendu adaptatif de grands objets sous forme de points. Nous pourrons ensuite combiner cette technique avec les méthodes de rendu réaliste, comme l'*antialiasing* ou le *Phong Shading*. Leurs implémentations sur les dernières générations de carte 3D sont décrites dans l'article [BHZK05].

Mots-Clés : rendu temps-réel, adaptatif, multirésolution, GPU, splatting, structure hiérarchique

Abstract

Today, softwares use larger and larger 3D objects. Althought technology is evoluting, objects manipulation is still a problem, especially the realtime rendering. The project's aim is to set up a multiresolution rendering system, which can display objects with several millions of polygons. First, we will implement the algorithm developped in the article [DVS03]. This presents a data structure optimized for the GPU, allowing the adaptative rendering of large point-based objects. In a second time, we will mix that with realistic rendering methods, like *antialiasing* or *Phong Shading*. Theirs implementations on latest generation of 3D graphics cards are described in [BHZK05].

Keywords : realtime rendering, adaptative, multiresolution, GPU, splatting, hierarchical structure

Table des matières

Chapitre 1

Introduction au domaine

1.1 Synthèse du domaine

Le rendu en trois dimensions (3D) est une partie indispensable du domaine plus global qu'est le graphisme 3D. Dans la chaîne de traitement graphique, c'est la dernière étape majeure : elle met la touche finale aux modèles et à l'animation.

Les principaux problèmes à résoudre sont la qualité du rendu, avec par exemple le réalisme du traitement de la lumière (ombrage, caustique, reflets, etc), ou encore le crénelage des scènes. Un autre problème est une conséquence du précédent : tous les traitements deviennent de plus en plus lourds, ce qui a un impact sur la technologie à employer, et plus important, sur les différentes techniques pour réduire la complexité et diminuer l'espace mémoire utilisé. Ainsi le traitement de grands objets en 3D requiert obligatoirement une maîtrise de différentes techniques pour régler ces problèmes.

Deux méthodes de rendu sont possibles. La plus courante est celle du rendu par polygones. Celle-ci est la méthode optimisée dans la technologie actuelle (GPU). La seconde méthode est celle du rendu par points. La méthode des polygones, par exemple, permet de calculer plus facilement les normales. Mais la méthode par points, sans contrainte, permet de manipuler plus librement les structures de données.

Enfin, il faut remarquer que le rendu varie selon son utilité :

- soit il a besoin de lourds traitements, auquel cas on parle de rendu pré-calculé : le calcul de la scène peut durer plusieurs heures ;
- soit l'interactivité est primordiale. On parle alors de rendu temps-réel.

Le rendu temps-réel, qui sera partie intégrante au projet, pose la question de la dualité entre la rapidité et la qualité du rendu. Comment allier l'un sans désavantager l'autre ?

1.2 Définitions

1.2.1 Grands objets

L'affichage de grands objets est un problème complexe car le nombre élevé de points ou de polygones qu'ils utilisent nécessite une place mémoire importante. Il existe deux manières de charger un objet : la méthode *in-core* charge un objet complètement dans la mémoire de travail avant de l'utiliser. La méthode *out-core*, au contraire, ne fait qu'un chargement partiel et récupère les éléments utiles au fur et à mesure sur le disque physique.

Pour les manipuler, il est nécessaire d'utiliser une structure de données spécifique suivant la méthode de chargement de l'objet ou le travail que l'on veut effectuer. En effet, certaines applications souhaiteront privilégier la recherche d'un élément dans la structure (par exemple un arbre), alors que d'autres préfèreront un parcours rapide de l'ensemble des données (une liste).

1.2.2 Level Of Detail

Un algorithme LOD (acronyme de *Level Of Detail*) permet d'adapter le niveau de détail d'un objet selon la façon dont il est perçu par l'observateur. En effet, un modèle lointain nécessite moins de détail, donc pourra être rendu avec moins de polygones ou de points, que le même modèle se trouvant proche de l'observateur. De la même manière, la géométrie d'un élément voilé par une nappe de brouillard ou représenté dans un environnement nocturne peut être simplifiée. L'emploi de cette technique permet d'alléger considérablement le travail du GPU, et donc de pouvoir augmenter le nombre d'images par seconde, donnée cruciale en temps-réel.

1.2.3 Octree

Un octree (oct + tree) est une structure de données hiérarchique. Chaque noeud de cet arbre possède 8 fils. Cette structure est utilisée principalement pour partitionner l'espace 3D, en vue de faire du LOD. Chaque noeud est représenté par un cube, élément qui peut être partitionné par 8 autres cubes (fils). Ainsi, selon le niveau de précision désiré, le moteur de rendu pourra simplifier grandement le nombre de détails à afficher.

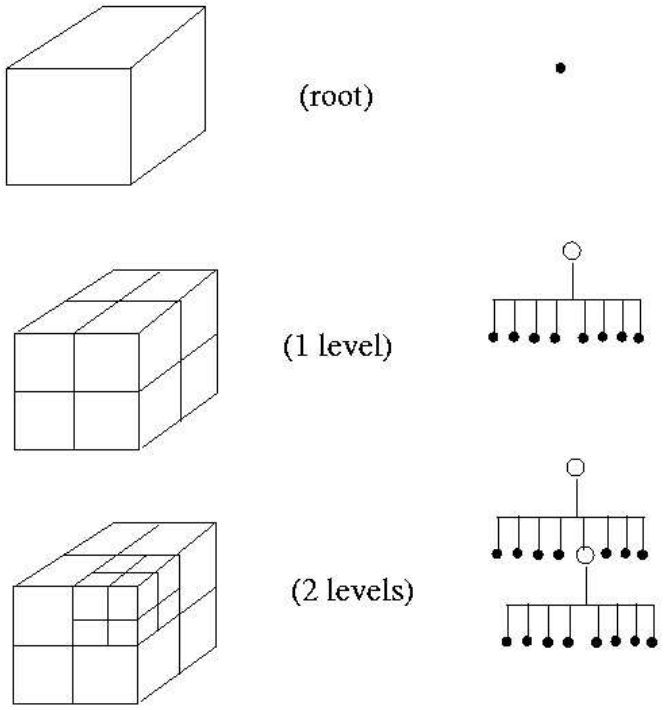


FIG. 1.1 – Octree[1]

1.2.4 Arbre séquentiel de points

Marc Levoy et Szymon Rusinkiewicz mettent en place en 2000 un système de représentation de grands objets, *QSplat*. Ce logiciel utilise une hiérarchie multirésolution pour effectuer un rendu par point.

L'arbre séquentiel de points est une structure spécialisée pour un rendu adaptatif multirésolution. Elle est définie dans l'article [DVS03] et se base essentiellement sur *QSplat*, une autre structure proposée dans l'article [RL00], mais en l'optimisant pour le GPU.

Dans un premier temps, les objets sont chargés dans un arbre binaire dans lequel chaque noeud représente un point de l'objet (caractérisé par un centre, un rayon, un cône de normale et éventuellement une couleur). Ces paramètres sont des moyennes de valeurs plus précises que l'on peut trouver dans des noeuds plus bas dans la hiérarchie. Les feuilles correspondent donc aux sommets des triangles, auxquelles un rayon aura été attribué auparavant.

Une fois cet arbre construit, il est réduit à une liste (la structure finale) qui permettra un rendu plus efficace car elle est très optimisée pour le GPU. Les

[1] Image tirée du site
http ://hpcc.engin.umich.edu/CFD/users/charlton/Thesis/html/node29.html

éléments qui sont contenus dans cette liste sont les mêmes que ceux de l'arbre, mis à part qu'ils contiennent deux valeurs de plus, pour éviter d'afficher à la fois un noeud père et un noeud fils.

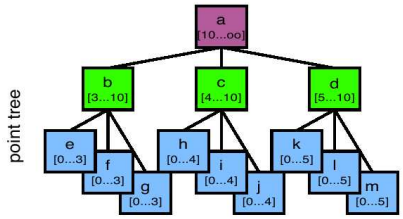


FIG. 1.2 – Arbre de points (utilisé dans QSplat)[1]

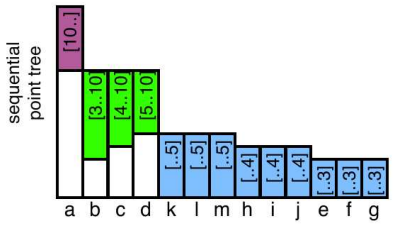


FIG. 1.3 – Arbre séquentiel de points[1]

1.2.5 Shaders

Plusieurs étapes sont nécessaires lors du processus de traitement graphique comme le montre la figure 1.4. A l'origine, les objets sont stockés sous forme de données géométriques que l'on charge en mémoire. Ils sont caractérisés par des informations sur leurs sommets (position, couleur, texture, ...) et d'autres concernant leurs primitives (point, triangle, rectangle ...).

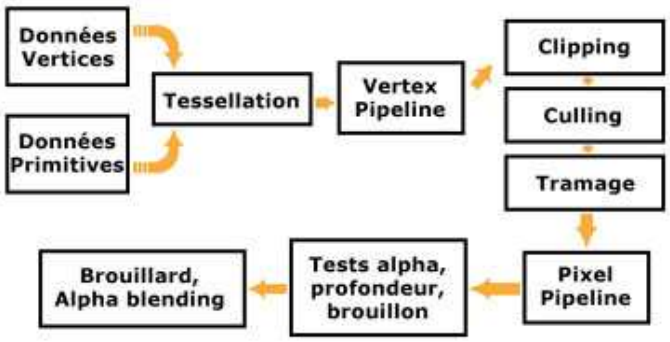


FIG. 1.4 – Pipeline d'une unité de traitement graphique[2]

La *tessellation*, qui est la première étape du rendu, consiste à rajouter des sommets dans l'objet afin d'augmenter leur nombre et ainsi d'affiner la surface dont ils font partie.

[1] Dachsbacher, C., Vogelgsang,C. and Stamminger, M..«Conversion of a point tree into a sequential point tree». *Sequential Point Trees*

[2] Image tirée du tutorial GLSL sur le site http ://www.lighthouse3d.com/opengl/glsl/

Le *vertex pipeline* est un élément-clé du GPU. Il permet d'effectuer les transformations élémentaires sur les coordonnées des sommets (translation, rotation et redimensionnement) qui sont choisies par l'utilisateur avec les commandes adéquates suivant le langage utilisé.

Une fois ces modifications effectuées, il est nécessaire de faire quelques opérations consistant à préparer les nouvelles données. Le *culling* consiste à enlever les primitives qui ne se verront pas à l'écran étant donné qu'elles se trouvent derrière d'autres objets. Le *clipping* sert aussi à minimiser le nombre de primitives à afficher. Cette technique permet de n'afficher que les sommets qui sont à l'intérieur du volume de vision. Enfin, le tramage permet de créer les pixels à partir des sommets et de leur forme géométrique.

Le rôle du *pixel pipeline* est de créer un pixel complet à partir d'un pixel et de quelques informations qui l'accompagnent comme les coordonnées de textures.

L'étape suivante est le test alpha. Elle consiste à créer des effets de transparence dans l'image, mais ralentie la fréquence de rafraîchissement car plus de pixels doivent alors être affichés. Le test de profondeur sert à n'afficher que les pixels visibles depuis le point de vue choisi par l'utilisateur. C'est un tampon spécial : le *depth buffer*, sauvegardant la profondeur de chaque pixel, qui est utilisé pour cette action. Le test brouillon se sert du *stencil buffer* pour effectuer d'autres effets comme l'ombrage ou les contours.

Enfin, le brouillard et le mélange alpha permettent respectivement de rajouter un brouillard et de rajouter de la transparence dans la scène.

Toutes ces étapes permettent d'afficher des objets géométriques dans une fenêtre de visualisation.

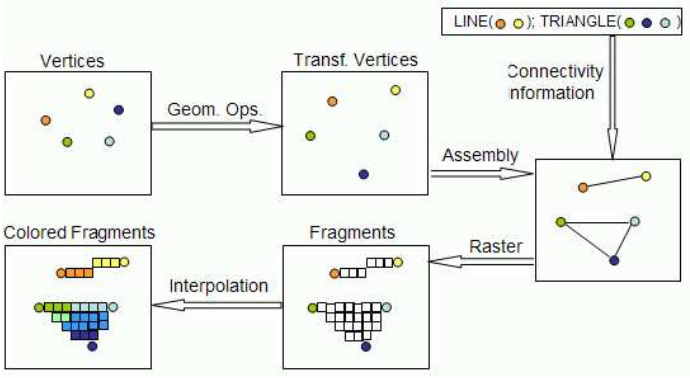


FIG. 1.5 – Visualisation des étapes de rendu[1]

Depuis l'apparition des cartes Nvidia Geforce3 et ATI Radeon8500 autour des années 2000, deux de ces processus sont devenus programmables pour l'utilisateur : le *vertex* et le *pixel pipeline*. C'est ce qu'on appelle les *shaders*. Leur

[1]Image tirée du tutorial GLSL sur le site http ://www.lighthouse3d.com/opengl/glsl/

programmation permet de créer de nouveaux effets ou bien même de définir des calculs de luminosités différents de ceux qu'offrent une simple bibliothèque 3D.

Plus précisémment, le *Vertex Shader* permet de modifier les propriétés des sommets, de calculer l'éclairage et de générer des coordonnées de texture. Le *pixel shader* (apparu un peu plus tard lors de l'arrivée des Nvidia GeforceFX), qui se trouve plus loin dans le pipeline, va agir sur les propriétés interpolées calculées précédemmment. De nombreuses autres opérations sont possibles, et permettent de faire des effets visuels surprenants.

1.2.6 Splatting

Le *splatting* (en français "éclaboussure") est la projection d'un point de l'espace objet (3D) sur l'écran (espace 2D). Le principe d'un algorithme implémentant cette technique est «d'envoyer» (virtuellement) le point sur l'écran afin qu'il laisse une «tache» sur celui-ci. En répétant le processus sur tous les voxels (unité définissant un point 3D) d'un objet, on obtient la projection de celui-ci sur l'écran. Les principaux problèmes rencontrés par cette technique sont l'apparition d'*aliasing* (crénelage) et de flou sur l'image finale.

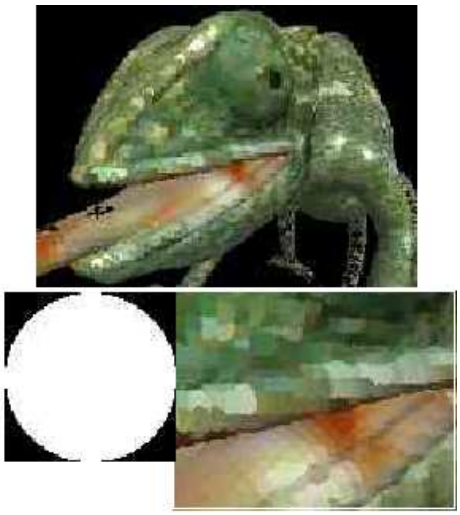


FIG. 1.6 – Splatting[1]

[1] Image tirée de la présentation technique *Deferred Shading* de Guennebaud, G., Barth, L. et Paulin, M.

1.2.7 Rendu Temps-réel

FIG. 1.7 – Film *Final Fantasy : The Spirit Within* © Sony Pictures (Précalculé)

FIG. 1.8 – Jeu *Half-Life 2* © Valve (Temps-réel)

De nos jours, de nombreuses applications ont besoin de calculer des scènes en 3D dont le temps de réponse doit être immédiat, contrairement aux rendus précalculés. Par exemple, les jeux vidéo, par essence, sont caractérisés par une interactivité avec l'utilisateur, alors que les films en images de synthèse n'en ont pas.

1.2.8 Rendu réaliste

De nombreuses techniques ont été développées pour améliorer la qualité du rendu. Nous retiendrons principalement :

Antialiasing

L'*aliasing* est un effet d'escalier plus ou moins marqué, visible sur certaines lignes droites. Cet effet non-désirable est dû au système de pixelisation utilisé par les écrans. En effet, lors du rendu, certains polygones, ou certaines textures ne sont pas correctes, voire disparaissent. De nombreux algorithmes, plus ou moins compliqués, ont été développés pour résoudre ce problème. On retrouve deux techniques, la première appelée préfiltrage (*prefiltering*) qui consiste à traiter des pixels dans une zone et choisir judicieusement la couleur : celle-ci rend alors l'objet plus épais, mais le crénelage disparaît. Cette technique est lourde en calcul et donc assez peu utilisée.

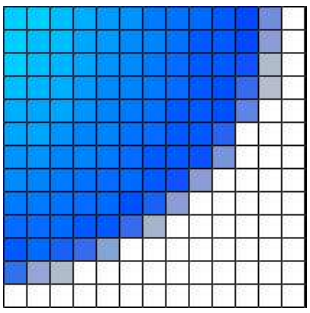


FIG. 1.9 – Représentation du préfiltrage[1]

La deuxième technique est appelée postfiltrage. Cette méthode se base sur les fréquences des courbes. Ces méthodes s'effectuent en trois étapes : on calcule la fréquence de la courbe, on applique un filtre passe-bas et on recalcule la courbe.

Aujourd'hui, l'algorithme de référence est l'algorithme EWA (acronyme de Elliptical Weighted Average). En 2002, M.Zwicker, H.Pfister, J.van Baar et M.Gross décrivent une méthode de splatting de haute qualité, EWA Splatting. Par exemple, le EWA Splatting (application de EWA pour le rendu par points) est une technique de filtrage consistant à reconstruire le signal (domaine continu) à partir d'échantillons (domaine discret), puis à appliquer un filtre passe-bas éliminant les parties du signal entraînant l'*aliasing*, avant de rééchantillonner.

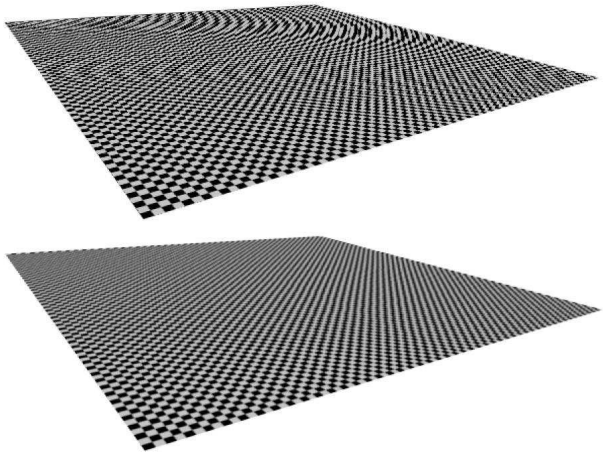


FIG. 1.10 – Antialiasing[2]

[1] Images tirées du site http ://web.cs.wpi.edu/ matt/courses/cs563/talks/antialiasing/methods.html

[2] Botsch, M., Hornung, A., Zwicker, M. and Kobbelt, L..Figure 3. *High Quality Surface Splatting on Today's GPUs*

Interpolation de Gouraud

Cette technique, expliquée en 1971 par Henri Gouraud dans son article [Gou71], a pour but de simuler les effets de la lumière diffuse sur les objets d'une scène.

L'algorithme consiste à interpoler linéairement les couleurs d'un polygone de la manière suivante : dans un premier temps, les normales sont calculées pour chacun des sommets. Pour cela, il faut faire la moyenne de chacune des normales adjacentes au sommet traité (figure de gauche 1.11).

Grâce à la direction de la lumière et à la normale calculée précédemment, il est alors possible de calculer une couleur pour le sommet par rapport à l'angle formé entre ces deux vecteurs. Une interpolation est alors effectuée le long des segments entre les sommets puis à l'intérieur de la facette (figure de droite 1.11).

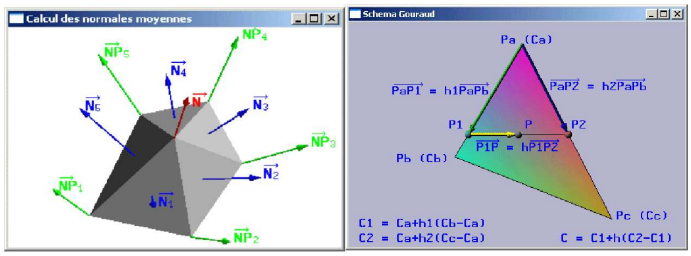


FIG. 1.11 – Calcul de la moyenne des normales pour chaque sommet (à gauche), interpolation de Gouraud (à droite)[1]

Il est facile d'implémenter cet algorithme à l'aide de la programmation GPU. En effet, il suffit de calculer les couleurs dans le Vertex Shader, et celles-ci sont alors automatiquement interpolées lors du passage dans le pixel shader. Cette méthode est donc rapide, mais elle produit cependant des effets visuels indésirables. Elle constitue tout de même un bon compromis entre la qualité du rendu et la vitesse de calcul.

[1] Images tirées du site de l'université de Franche-Comté http ://raphaello.univ-fcomte.fr/IG/Physique/Physique.htm

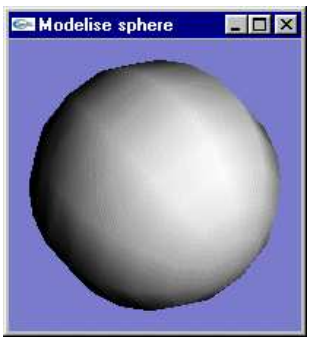


FIG. 1.12 – Ombrage de Gouraud[1]

Cette technique (comme l'interpolation de Phong) peut être appliquée également au rendu par point.

Interpolation de Phong

Le *Phong Shading*, édité dans [Pho75] par Bui Tuong Phong en 1975 est une technique d'éclairage donnant de meilleurs résultats que celle de Gouraud. La principale différence est que l'interpolation ne se fait plus sur les couleurs mais sur les normales. De plus, elle prend en compte trois paramètres lumineux : la spécularité, l'intensité diffuse et l'intensité ambiante. Cela permet d'éviter certaines limites dûes au modèle de Gouraud comme la réflexion spéculaire. Le rendu est donc beaucoup plus réaliste, mais il est aussi bien plus lent car les calculs sont plus nombreux et se font sur des réels.

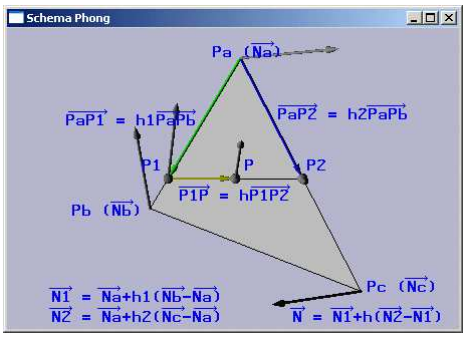


FIG. 1.13 – Interpolation de Phong[1]

L'intensité en un point P se calcule de la manière suivante :

$$I = I_a + I_d + I_s$$

[1]Images tirées du site de l'université de Franche-Comté http ://raphaello.univ-fcomte.fr/IG/Physique/Physique.htm

$$I_a = k_a I_{a0}$$

I_a représente l'intensité dûe à la lumière ambiante. $k_a \in [0,1]$ est le coefficient de la lumière réfléchie et I_{a0} est l'intensité de la lumière ambiante.

$$I_d = k_d cos\theta I_{source}$$

I_d est l'intensité dûe à la lumière diffuse. k_d est le coefficient de réflexion diffuse de la surface et θ est l'angle entre la direction du rayon lumineux et la normale à la surface.

$$I_s = k_s cos^n(\alpha) I_{source}$$

Enfin, I_s correspond à l'intensité dûe à la lumière spéculaire (elle apparaît dans les zones de forte intensité et sur les surfaces brillantes). k_s est le coefficient spéculaire et n contrôle la rugosité de la surface (celle-ci devient de plus en plus lisse lorsque n tend vers l'infini). On obtient donc la formule de l'intensité globale en un point :

$$I = k_a I_{a0} + k_d cos\theta I_{source} + k_s cos^n(\alpha) I_{source}$$

Il est aussi possible de rajouter un terme $I_\alpha = k_\alpha I_{\alpha 0}$ de façon à gérer la transparence.

De plus, s'il existe plusieurs sources de lumière dans la scène, alors I_d et I_s correspondent à la somme de leurs intensités correspondantes pour chacune de ces sources.

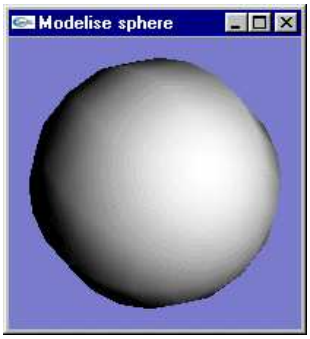


FIG. 1.14 – Ombrage de Phong[1]

[1] Images tirées du site de l'université de Franche-Comté http ://raphaello.univ-fcomte.fr/IG/Physique/Physique.htm

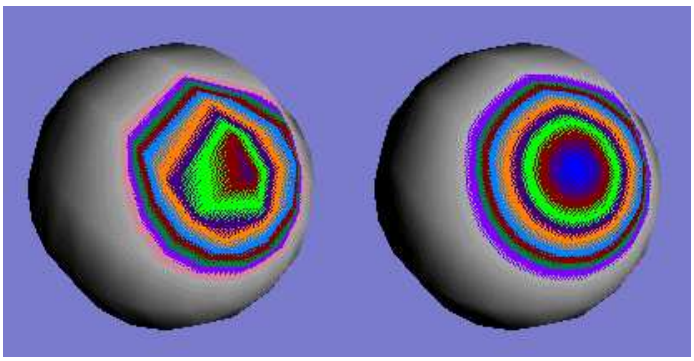


FIG. 1.15 – Comparatif (à gauche : Interpolation de Gouraud ; à droite : Interpolation de Phong) [1]

1.3 Exemples

De nombreuses applications de rendu 3D manipulent de grands objets. Leur nombre tend d'ailleurs à augmenter. Plus le nombre de polygones ou de points est grand, plus la scène est susceptible d'être grande, précise et/ou réaliste (suivant ce que l'on veut représenter). De plus, certaines techniques d'acquisition comme les scanners 3D entraînent nécessairement la manipulation d'un grand nombre de points. Les applications concernant la géographie ou le relief des terrains (la cartographie et la topographie) gèrent aussi d'impressionantes quantités de polygones. Beaucoup d'autres programmes manipulent de tels objets ; parmi eux on peut citer la conception assistée par ordinateur (C.A.O), la réalité virtuelle qui tente de reconstruire des villes entières en 3D, ou encore les applications concernant les représentations de monuments archéologiques.

1.4 Logiciels existants

Il existe très peu de logiciels (disponibles) employant la méthode de rendu par points. Les plus connus sont *QSplat*, *PointShop3D* et ses nombreux *plugins*. Enfin il existe une autre application moins connue, *Surfel Viewer*.

QSplat

QSplat est un logiciel de rendu temps-réel implémentant la structure de données définie dans l'article [RL00]. Il s'agit donc d'un logiciel multirésolution effectuant un rendu par points et dont les détails s'affinent lorsque l'objet ne bouge pas. Cet article datant de 2000, les auteurs n'ont pu faire qu'un rendu logiciel

[1] Images tirées du site de l'université de Franche-Comté http ://raphaello.univ-fcomte.fr/IG/Physique/Physique.htm

(*software*). Selon ceux-ci, sur une machine relativement faible (Intel Pentium II 366 Mhz, 128 Mo de mémoire vive et pas de carte graphique accélératice 3D), *QSplat* peut rendre environ 60 000 splats par image (résolution de 500x500), à raison de 5 images par seconde. On a donc 300 000 splats par seconde.

(Disponible sur http ://graphics.stanford.edu/software/qsplat/)

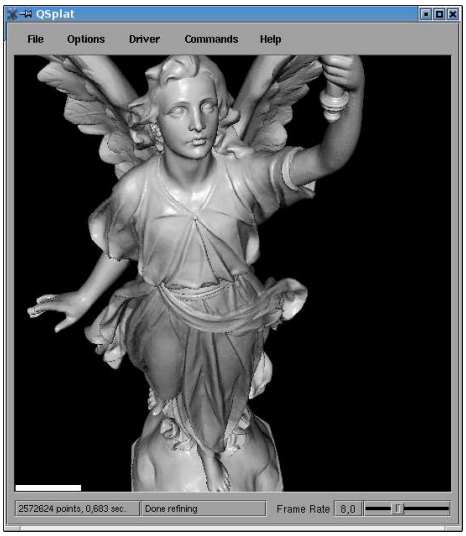


FIG. 1.16 – Copie d'écran du logiciel *QSplat*

Pointshop3D

PointShop3D est un logiciel spécialisé dans le rendu par points dont la réalisation est décrite dans [ZPKG02]. Il existe de nombreuses extensions (*plugins*) dont un moteur de rendu qui se sert du GPU (le moteur par défaut est un rendu logiciel). L'écriture de plugins est très simple. Pour les performances, avec le moteur de rendu logiciel, l'application réussit à rendre 500 000 splats «antialisés» par seconde, sur un Intel Pentium IV 2GHz (avec une image de 512x512). Parmi les extensions proposées, quelques-unes ont retenu notre attention :

- **OpenGLRenderer** ;
- **SurfelRenderer** ;
- **EWAGLRenderer** ;
- **HGLSplatRenderer** ;

Ils permettent de changer la méthode de rendu ; les deux derniers utilisent les shaders des cartes graphiques.

(Disponible sur http ://graphics.ethz.ch/pointshop3D/)

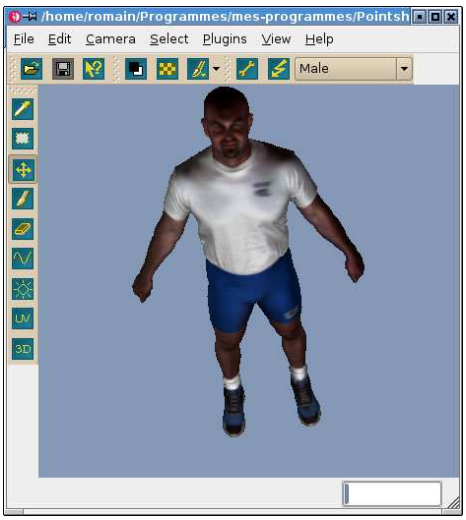


FIG. 1.17 – Copie d'écran du logiciel *Pointshop3D*

Surfel Viewer

Ce logiciel a été développé par Bart Adams, pour les besoins de ses recherches. Bien que relativement simple, il peut rendre les objets de 6 manières différentes (selon des points, des disques, ...). Mais il ne prend en entrée que des fichiers de format spécifique.

(Disponible sur http ://www.cs.kuleuven.ac.be/ barta/sview/sview.html)

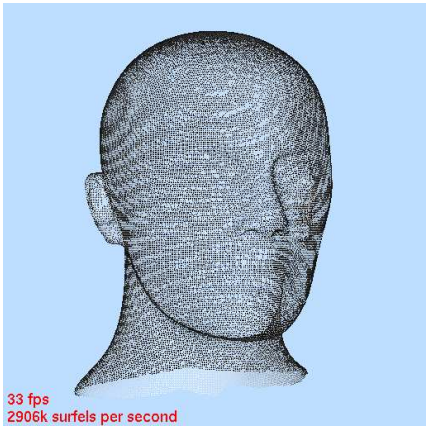


FIG. 1.18 – Rendu simple avec *GL_POINT*

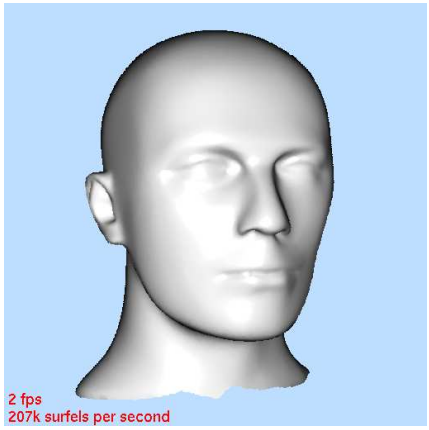


FIG. 1.19 – Rendu avec splatting et filtrage

Chapitre 2

Introduction au projet

Notre projet a pour but de rendre en temps-réel, de manière efficace, des objets 3D constitués de plusieurs millions de polygones. Nous allons pour ce faire, mettre en place un moteur multirésolution, implémentant les techniques décrites dans les rapports de recherche [DVS03] et [BHZK05].

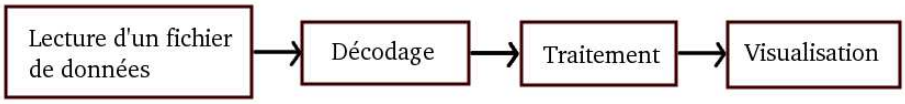


Fig. 2.1 – Déroulement des tâches

Notre premier objectif est le chargement de grands objets et leur rendu sous forme de points. Pour cela, nous stockerons l'objet en mémoire à l'aide de la structure de données explicitée dans [DVS03]. Celle-ci permet d'adapter dynamiquement le niveau de détails de l'objet en fonction de la position de l'observateur, optimisant ainsi le temps de rendu. De plus, la quasi-totalité des opérations est effectuée par le GPU, libérant de ce fait le CPU d'une charge de travail importante. Cette technique est basée sur une représentation du modèle sous forme d'un arbre de points, proche de celui utilisé dans [RL00]. Chaque noeud, correspondant à une partie de l'objet, est caractérisé par un point central P, une normale moyenne et le diamètre D de la sphère autour de P englobant la partie de l'objet représentée. Une feuille comprend réellement un ensemble de points de l'objet, alors que les valeurs des noeuds intérieurs sont obtenues par approximation des valeurs de ces fils. Pour chaque noeud, l'erreur engendrée par cette approximation (appelée erreur géométrique) est évaluée et incluse dans celui-ci. Ainsi, plus on remonte dans l'arbre, plus l'erreur est importante. L'algorithme de rendu est alors relativement simple : il suffit, pour chaque ramification de l'arbre, de la parcourir jusqu'à trouver un noeud (que l'on rendra à l'écran) tel que son erreur divisée par la distance entre le noeud et l'observateur soit inférieure à un certain seuil fixé par l'utilisateur. Pour que le traitement par

le GPU soit efficace, l'idée d'arbre est implémentée effectivement par une liste, permettant un parcours séquentiel.

Nous nous attacherons parallèlement à implémenter les techniques de rendu offrant aujourd'hui les meilleurs résultats en terme de qualité. Celles-ci sont étudiées dans [BHZK05] sur les dernières générations de cartes graphiques. Ce rapport décrit une méthode obtenant des résultats comparables au rendu de surfaces de points EWA original décrit dans [ZPvBG02], qui est actuellement la référence dans le domaine. De plus, par rapport aux précédentes implémentations GPU de qualité comparable, elle permet le traitement d'un nombre bien plus élevé de points par seconde. Elle consiste à réaliser tout d'abord une interpolation de Phong à l'aide d'un algorithme en trois étapes (*Deferred Shading*). En premier lieu, la projection des points de l'objet 3D sur l'écran est calculée afin de remplir le z-buffer (*Visibility Pass*). Ensuite, pour chaque pixel, la valeur du vecteur normal lui correspondant ainsi que sa couleur sont stockées séparément dans d'autres tampons fournis par la carte graphique (*Attribute Pass*). Enfin, ces tampons et le z-buffer sont utilisés comme textures pour être normalisés et enfin pouvoir rendre l'image finale (*Shading Pass*). Pour supprimer l'*aliasing* pouvant encore apparaître dans le cas d'une forte réduction, un filtre particulier, approximation du filtre passe-bas EWA, est combiné à l'interpolation.

Nous combinerons enfin ces deux techniques dans une application alliant rapidité et qualité de rendu.

Chapitre 3

Besoins Non-Fonctionnels

3.1 Qualités globales

Le temps de réponse est l'élément primordial de ce projet : le but étant d'avoir un rendu temps-réel, il faut veiller à ce que les opérations soient effectuées presque instantanément, c'est-à-dire espérer avoir un temps de réponse inférieur à $\frac{1}{25}^{ieme}$ de seconde (pour avoir 25 images/seconde), ce chiffre dépendant de la fréquence de rafraîchissement choisie. On voit alors apparaître la principale contrainte de ce projet, le compromis entre la rapidité de l'exécution et la qualité du rendu.

L'application assurera une certaine robustesse. Lors de la construction de l'arbre séquentiel de points, plusieurs taux d'erreurs seront calculés pour chaque noeud. Ceci permet de bien définir les points corrects à afficher à l'écran. Les techniques que nous utiliserons ont déja été prouvées, ce qui garantira la fiabilité de l'application.

La sûreté de fonctionnement n'est pas une priorité. Ceci est renforcé par le fait que le programme n'aura que des fonctions de visualisation.

Le programme est conçu principalement pour Linux, mais grâce à des bibliothèques standards (OpenGL, QT, ...), il pourra être facilement portable sur d'autres plate-formes.

3.2 Tests de validation

3.2.1 Tests du *Sequential Point Trees*

Construction de l'arbre

Nous allons tester si l'arbre construit par l'application représente bien l'objet à visualiser. Pour cela, nous vérifierons tout d'abord que le nombre de feuilles correspond bien au nombre de polygones du modèle. Puis, nous nous assurerons que chaque noeud a un niveau d'erreur concordant avec sa position dans l'arbre, c'est-à-dire que cette erreur soit supérieure ou égale à celle de ces fils.

Conversion de l'arbre en liste

Lors de la séquentialisation de notre structure, nous pourrons tester les correspondances entre notre arbre et la liste passée au GPU pour vérifier qu'elle soit correcte. Dans un premier temps, nous vérifirons que le nombre de noeuds dans l'arbre (feuilles comprises) est égal au nombre d'éléments dans notre liste.

En outre, lorsqu'un noeud sera sélectionné pour être affiché dans la liste, il est nécessaire que l'ensemble des noeuds correspondant à ses fils dans l'arbre ne le soit pas. Pour cela, nous nous servirons de l'intervalle qui définit les seuils dans l'arbre. Il suffira de faire un simple parcours de l'arbre, et de vérifier pour chacun des noeuds si l'intervalle correspondant est plus grand que celui de leurs fils dans la liste.

Ainsi, durant le parcours de la liste par le GPU, nous serons sûr que plusieurs noeuds d'une même branche de l'arbre ne seront pas affichés en même temps.

Résultats attendus

Tous les tests seront effectués sur une machine composée d'une carte graphique Nvidia 6600 GT, d'un processeur Intel Pentium IV 2,8GHz et de 1024Mo de mémoire vive.

A la fin de cette partie, l'application devra charger un fichier dans l'arbre séquentiel de données et le convertir en une liste en moins d'une minute.

Le modèle de référence sera *lucy.ply*. C'est un objet composé de 14 millions de sommets et 28 millions de polygones. Celui-ci devra être affiché à 35 fps (*frame per second*).

3.2.2 Tests du rendu

Cette étape est validée si sur la machine de test, notre plugin *PointShop3D3D* permet de rendre 3 millions de splats par seconde avec une résolution de 512 x 512 pixels.

3.2.3 Tests d'intégration

La première chose à faire est de s'assurer que tous les tests réalisés pour les étapes précédentes réussissent toujours (exception faite des tests de performance) Nous devons vérifier ensuite que toute les fonctionnalités de l'interface soient opérationnelles. L'interêt principal de cette étape est d'observer les performances obtenues par la combinaison des algorithmes de sélection et de rendu implémentés précédemment. C'est pour cela que nous ne pouvons fixer un nombre minimal de splats par seconde à afficher.

Chapitre 4

Besoins Fonctionnels

Ce projet consistera à implémenter une application permettant de charger et de visualiser des grands objets, constitués de plusieurs millions de points et/ou polygones. Il se décompose en trois parties :

- chargement d'un grand objet au format *ply* dans une structure adaptée,
- mise en place d'un rendu réaliste,
- combinaison des deux méthodes précédentes pour obtenir l'application finale.

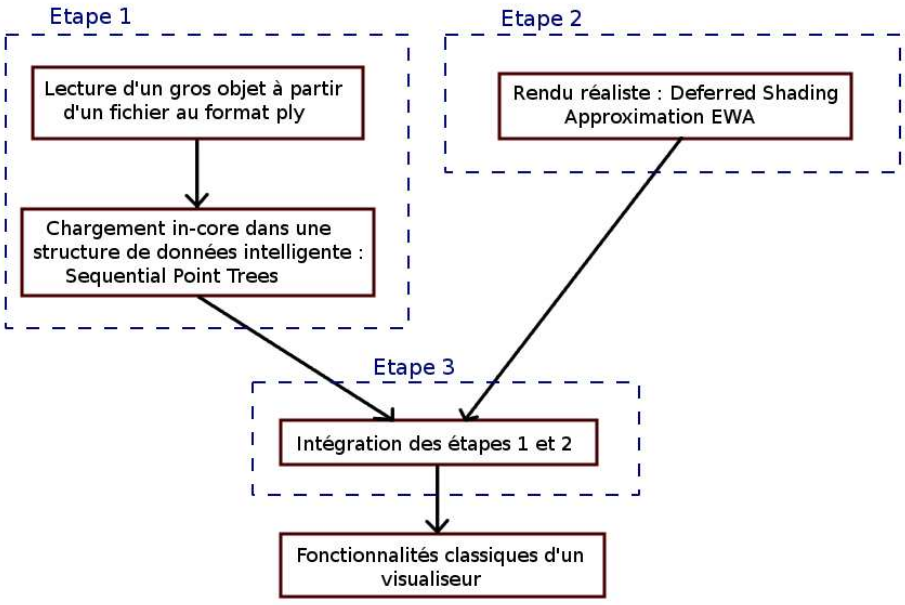


FIG. 4.1 – Organisation des fonctionnalités

La troisième étape n'est pas prioritaire car le temps imparti ne nous le permettra peut être pas.

4.1 Lecture d'un grand objet à partir d'un fichier au format *ply*

Un fichier de type *ply (Polygon File Format)* contient :
- le nombre de sommets,
- le nombre de facettes,
- les coordonnées spatiales de chaque point,
- pour chacune des facettes, le nombre de sommets les constituant ainsi que l'indice de ceux-ci.

Il est possible de rajouter des couleurs à chaque facette, ainsi que d'autres propriétés. Enfin, il existe deux formes de fichier *ply* : ASCII ou binaire.

L'application devra être capable de lire ce type de fichier (sous ces deux formes) pour que l'on puisse stocker :
- soit des couples (point,normale),
- soit des n-uplets (point,normale,couleur+propriétés).

4.2 Chargement *in-core* dans une structure de données intelligente : *Sequential Point Trees*

Le moteur de l'application devra être adaptatif et multirésolution. Pour cela, on utilisera une structure de données pour stocker l'objet en mémoire, présentée dans [DVS03]. Cette structure sera un arbre dans lequel chaque noeud correspondra à une partie de l'objet. Les feuilles représenteront les parties les plus précises de l'objet et les noeuds internes seront obtenus par approximation de leurs fils (ce qui génèrera une certaine erreur). Le parcours de l'arbre permettra de sélectionner les points à afficher en fonction de la distance du point de vue et du taux d'erreur. Ainsi, plus l'observateur sera proche de l'objet, plus les détails apparaîtront. On obtiendra ainsi un rendu multirésolution. Le moteur sera aussi adaptatif. En effet, une géométrie complexe d'une zone de l'objet entraîne une approximation importante, et donc une représentation avec un plus grand nombre de points.

Cet arbre sera converti en liste pour pouvoir être traité séquentiellement par le GPU. L'objet étant chargé intégralement en mémoire, cette méthode est *in-core*.

Nous pourrons dans un second temps optimiser le rendu en représentant des zones de l'objet par des polygones à la place des points lorsque cette solution sera la plus rapide. On aboutira donc à un moteur hybride point/polygone.

4.3 Rendu réaliste : *Deferred Shading* et approximation EWA

Nous devrons implémenter les techniques de rendu réaliste explicitées dans [BHZK05], en exploitant les capacités offertes par les cartes graphiques actuelles. Cette méthode fournit un «*per-pixel Phong shading*» ainsi qu'une approximation EWA permettant de réaliser un *antialiasing* haute-qualité.

Comme l'implémentation permet une séparation claire entre la rasterisation (conversion d'une image vectorielle en une matrice de pixels) et le *shading*, il sera possible de mettre en place d'autres types de rendu comme les rendus Non-Photo Réaliste (NPR).

4.4 Fonctionnalités classiques d'un visualiseur

L'application devra être simple d'utilisation avec les options proposées par un navigateur classique d'objet 3D :

- déplacement autour de l'objet,
- fonctions de zoom,
- déplacement de la source lumineuse.

La fréquence de rafraîchissement pourra également être réglée. L'application permettra aussi de changer les couleurs de l'objet selon deux critères :

- couleurs réelles de l'objet,
- couleurs différentes selon le niveau de détail.

Si le moteur hybride est implémenté, nous pourrons aussi afficher les points et les polygones selon des couleurs différentes.

4.5 Prototype Papier

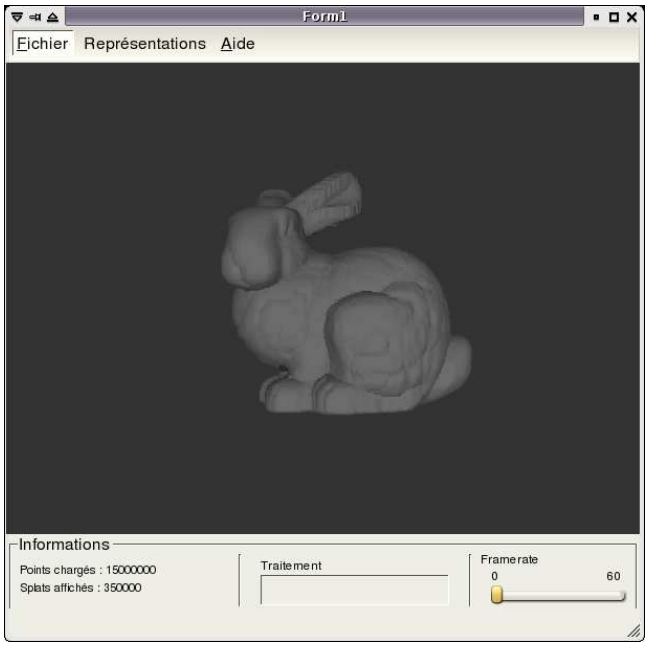


FIG. 4.2 – Interface du programme

4.5.1 Ouverture du fichier

Le modèle contenu dans le fichier *ply* doit être ouvert par l'intermédiaire du choix *Charger* qui se trouve dans le 1^{er} menu.

4.5.2 Déplacement de la caméra & Zoom

L'utilisateur peut déplacer la caméra dans la scène 3D grâce aux flèches directionnelles du clavier et la faire pivoter en maintenant le clic gauche de la souris enfoncé. On peut se rapprocher (resp. s'éloigner) de l'objet en utilisant la roulette de la souris vers le bas (resp. vers le haut).

4.5.3 Déplacement de la lumière

La source lumineuse peut être déplacée avec le clic droit de la souris (en la faisant glisser ou en cliquant simplement).

4.5.4 Visualisation

Diverses formes de représentations sont permises en choisissant dans le menu *Représentations* ou dans la barre d'outils :

- par défaut : les couleurs représentées sont les couleurs réelles du modèle ;
- en niveau de détail : les couleurs symbolisent le niveau de détail de l'objet ;
- hybride : l'affichage dissocie les points et les polygones à l'aide de deux couleurs différentes.

4.5.5 Fréquence de rafraîchissement

La fréquence de rafraîchissement (*framerate*) peut être modifiée avec la glissière (*slider*), en bas à droite de l'interface. Celle-ci pourra varier de 0 fps jusqu'au maximum de 60 fps.

4.5.6 Informations

Diverses informations sont affichées dans la barre du bas :

- le nombre de points du modèle (contenus dans le fichier),
- le nombre de splats actuellement affichés à l'écran,
- une barre de progression évaluant la durée du chargement du fichier lors de l'ouverture.

4.5.7 Autres

Beaucoup de raccourcis-clavier sont implémentés par défaut par QGLViewer ; ceux-ci peuvent être consultés en sélectionnant le menu *Aide*.

Chapitre 5

Tests préparatoires

5.1 Les différents tests effectués

5.1.1 Chargement de fichiers ply

Fichier	Nombre de points chargés	Temps de chargement (s)
bun_zipper.ply	69451	0.27
dragon_vrip.ply	871414	2.85
happy_vrip.ply	1087716	3.46

FIG. 5.1 – test effectué sous Linux avec une machine équipée d'un processeur Intel Pentium IV 3.2GHz, de 1Go de RAM et d'une Nvidia 6800 GS

Le prototype permettant de charger des gros fichiers dans une structure simple (telle qu'une liste), nous permet d'estimer facilement le temps qui nous sera nécessaire dans l'application finale.

5.1.2 Comparaison des bibiothèques *QGLViewer/GLUT*

L'application devra être facile d'utilisation pour toute personne habituée à manipuler des interfaces graphiques. Pour cela, nous allons employer *QGLViewer*, une librairie libre écrite en C++ basée sur la bibliothèque QT, utilisée notamment pour l'environnement graphique KDE. En effet, *QGLViewer* permet la création rapide d'un visualiseur de scènes 3D OpenGL, avec la plupart des fonctionnalités désirées par défaut. Nous avons comparé les performances obtenues entre le visualiseur de base *QGLViewer* et un autre proposant les mêmes services, implémenté avec la librairie *GLUT*.

Nombres de polygones	GLUT (fps)	QGLViewer (fps).
160 000	3.3	25
320 000	1.6	15

FIG. 5.2 – test effectué sous Linux avec une machine équipée d'un processeur Intel Pentium IV 3.2GHz, de 1024Mo de RAM et d'une Nvidia 6800 GS.

Ces tests confirment le choix de *QGLViewer*.

5.1.3 Chargement *in-core*

Nous avons voulu tester l'aspect *in-core* du projet. Pour cela nous avons implémenté naïvement un arbre à quatre fils, un *quadtree*. Nous avons construit un premier arbre sans aucune optimisation, dans lequel les feuilles et les noeuds sont implémentés de la manière suivante :

- un tableau de 3 flottants correspondant au centre du point,
- un flottant pour la taille du rayon,
- un tableau de 3 flottants correspondant au vecteur normal,
- un tableau de taille 4, pointant vers ses fils.

En remplissant cet arbre avec un million de feuilles (ce qui correspond à un objet d'un million de points), on obtient un arbre de 1,4 millions de noeuds, ce qui occupe une place de 60Mo en mémoire avec une moyenne de 44 octets par noeud.

Nous avons ensuite légèrement optimisé l'arbre en distinguant les feuilles, ne possédant pas de pointeurs, des noeuds. Ce qui nous donne pour la même taille, une occupation de la mémoire de 44Mo soit une moyenne de 32 octets par noeud. D'autres optimisations seront effectuées notamment sur l'implémentation des noeuds.

Le parcours total de l'arbre est effectué en 45ms (test effectué sous Linux avec une machine équipée d'un processeur Intel Celeron 2.4GHz, de 256Mo de RAM et d'une Nvidia Geforce4 MX). On remarque que nous n'utiliserons que certaines branches de l'arbre lors du parcours. Des optimisations sont donc possibles.

5.1.4 Splatting

Nous avons aussi voulu savoir combien de splats par seconde nous étions en mesure d'afficher à l'aide d'un moteur 3D simple. Pour cela, nous avons développé un prototype capable de charger un objet composé de plusieurs dizaines de milliers de polygones et d'afficher un point (*GL_POINTS*) pour chacune des faces.

Fichier	Nombre de splats/frame	framerate (fps)
bun_zipper.ply	69451	50
dragon_vrip_res2.ply	202520	17
dragon_vrip.ply	871414	4.1
happy_vrip.ply	1087716	3.3

FIG. 5.3 – test effectué sous Linux avec une machine équipée d'un processeur Intel Pentium 4 3.2GHz, de 1024Mo de RAM et d'une Nvidia 6800 GS

5.2 Risques et alternatives

D'après les résultats des tests préparatoires, les risques principaux sont liés à la mémoire utilisée pour charger de tels objets, et au grand nombre de points à afficher en un laps de temps très court.

Nous devons pouvoir gérer 14 000 000 de points dans notre structure. Or, le test effectué précédemment sur le *quadtree* montre que nous ne pourrons pas charger plus de 5 000 000 de points à l'aide d'une structure naîve. Nous devrons donc mettre en place certaines optimisations pour gagner de la place. Pour cela, nous pourrons prendre comme exemple la structure de l'arbre implémentée dans *QSplat* et faire des restrictions sur la taille de chaque noeud. Nous pourrons aussi nous servir de documents comme [BWK02] qui permet d'optimiser efficacement le codage de points.

L'autre risque concerne l'affichage des surfaces de points en temps limité. Le test de splatting permet d'afficher 2 000 000 de points par seconde en utilisant des *GL_POINTS*. Pour arriver à 3 000 000, il nous faudra utiliser des structures plus efficaces proposées par la librairie OpenGL, telle que les *glList* ou les *Vertex Buffer Object (VBO)*.

5.3 Choix des outils

L'ensemble du projet sera développé à l'aide de la librairie graphique OpenGL. Cette dernière est libre, portable et performante grâce à l'intégration directe de ses fonctionnalités au sein des cartes graphiques. Nous avons vu précédemment que les services de visualisation seront implémentés par QGLViewer, non seulement pour sa simplicité d'utilisation grâce à son large choix d'options mais aussi pour son efficacité. Nous programmerons avec le langage C++. C'est un langage compilable, ce qui le rend rapide lors de son éxécution, aspect primordial du projet. De plus, QT nécessite l'usage de ce langage.

Pour réaliser la seconde partie du projet (obtenir un rendu réaliste de la scène), nous utiliserons le logiciel *PointShop3D*. Celui-ci permet d'insérer des plugins, nous pourrons donc programmer indépendamment la structure utilisée pour stocker l'objet.

Remarque préliminaire

Notre projet est divisé en deux parties distinctes :

- **LargeObjectsViewer**, qui implémente le chargement *in-core* d'un grand objet dans l'arbre séquentiel de points avec un rendu basique ;
- **HQSSRenderer**, le plugin de *PointShop3D* implémentant les techniques de rendu réaliste

Tout au long de l'analyse, nous présenterons donc chacune de ces parties séparément.

Chapitre 6

Déroulement du programme

6.1 LargeObjectsViewer

6.1.1 Interface graphique

Lors de l'exécution du programme, une fenêtre utilisant la bibliothèque graphique **Qt4** de **TrollTech** se lance.

L'application étant graphique, la majeure partie de la place est consacrée à la visualisation. Cependant, on peut y voir le nombre de points totaux de l'objet à charger et le nombre d'images par seconde. Enfin, on pourra régler le niveau de détails lors de la sélection des points par l'intermédiaire d'une glissière.

6.1.2 Ouverture de fichier et traitement

Après avoir choisi un fichier dans le menu approprié (Fichier⟶Ouvrir), le *GUI* lance un thread effectuant la construction du SPT. En effet, lors de sa mise en route, le thread lance la fonction **build()**, fonction principale de l'interface **SurfelsStructureInterface**. Celle-ci effectue dans l'ordre (pour un arbre séquentiel de points) :

- la lecture du fichier *ply*, grâce à l'interface **FileLoaderInterface** et son implémentation, **PlyFormatFile** ;
- la construction de l'arbre de points (de style *QSplat*) ;
- le calcul des erreurs géométriques de chaque noeud/feuille dans l'arbre ;
- sa séquentialisation dans un tableau ;
- enfin le tri de ce dernier.

Les données nouvellement créées sont passées au *widget* **Viewer**. Les différentes étapes sont tracées dans la console où le programme a été lancé.

```
Loading : /.../ply/dragon.ply
Number of vertices=437645
Number of faces=871414
Computing normals... Done.
Computing splat sizes... Done.
Building tree...Done.
Calculate lower bound error...Done
Calculate upper bound error...Done
Sequentialization...Done
Sort SPT...Done
Put SPT into Memory...Done
```

6.1.3 Rendu

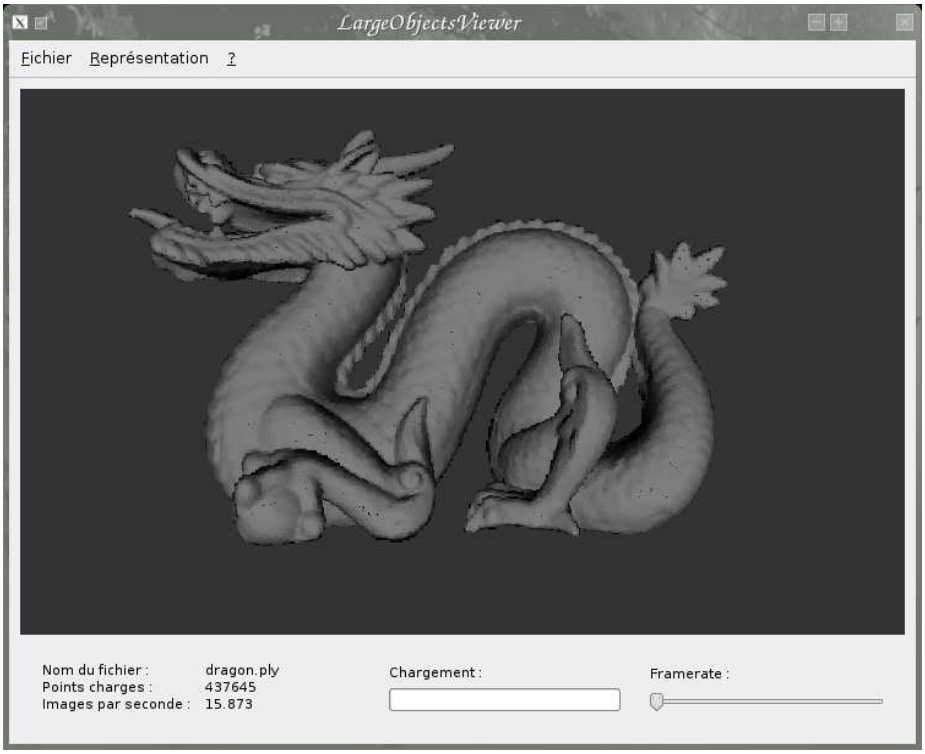


FIG. 6.1 – Rendu de l'objet *dragon*

Dès que le *widget* s'occupant de la visualisation reçoit la structure, celui-ci appelle à chaque modification de l'affichage (déplacement de la fenêtre ou de l'objet entre autres) la fonction **drawSelectedSurfelsList()**, pour envoyer au *Vertex Shader* les points devant être utilisés. Dans le cas du *Sequential Point*

Trees, celle-ci effectue auparavant une préselection selon l'erreur maximale par rapport à la distance de vue (distance entre la caméra et l'objet). Dans le *Vertex Shader*, une deuxième sélection est faite en comparant les deux erreurs calculées par rapport à la distance de vue. Enfin les points restants sont projetés à l'écran («splattés»), selon une couleur dépendant du mode de représentation actuel :

- soit en gris (les couleurs n'étant pas gérées) ;
- soit les couleurs selon le niveau des points dans la hiérarchie calculée précédemment.

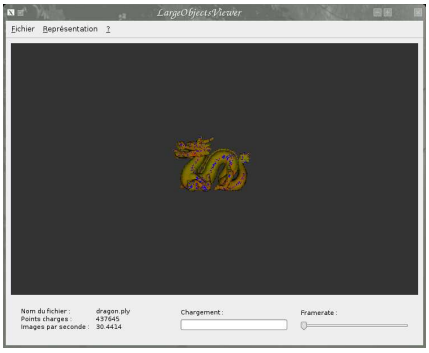


FIG. 6.2 – LOD avec l'objet éloigné

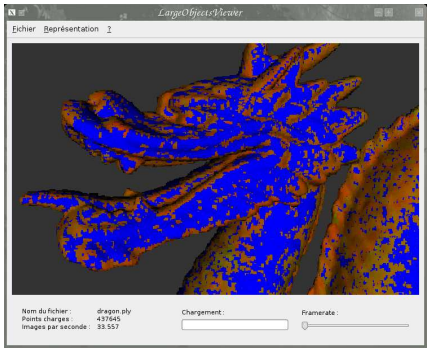


FIG. 6.3 – LOD avec l'objet proche

6.2 HQSSRenderer

PointShop3D charge un fichier objet de type *sfl*. L'utilisateur choisit alors notre plugin dans « Edit⟶Settings⟶Renderer General ». Cet objet est alors représenté à l'écran avec notre rendu réaliste, basé sur le principe du *deferred Shading*. Avant d'obtenir un résultat concret, trois étapes préliminaires sont nécessaires.

La première étape permet de remplir le tampon de profondeur. En effet, le *surfel* (le point 3D représenté par une position, une normale et deux vecteurs tangents dont la norme est la longueur du rayon) doit être projeté à l'écran afin d'obtenir un *splat*. On peut calculer la profondeur de celui-ci, et ainsi garder seulement les parties visibles de l'objet. Voici un aperçu de cette étape nommée *Visibility Pass*.

FIG. 6.4 – *Visibility Pass*

Cette première étape est indispensable car elle permet un gain de temps considérable. Elle réduit le nombre de calculs pour les étapes qui suivent.

Le second traitement effectué est nommé *Attribute Pass*. Son rôle est de remplir deux textures contenant d'une part les couleurs respectives à chaque pixel, et, d'autre part, les normales. Pour obtenir un résultat satisfaisant, chaque pixel est pondéré par une fonction gaussienne afin d'effectuer un fondu entre chaque *splat*. Les couleurs entre plusieurs *splats* superposés sont mélangées. Sans cela, le dernier *splat* traité couvre ceux qui sont en dessous de lui. Voici le contenu de chacune de ces deux textures.

FIG. 6.5 – *AttributeColorPass*

FIG. 6.6 – *AttributeNormalPass*

Une dernière étape est nécessaire afin d'obtenir un rendu réaliste. Il faut désormais gérer les reflets lumineux sur l'objet en question. Le rôle de cette étape est d'obtenir une lumière diffuse et spéculaire. La lumière est réfléchie en fonction de l'inclinaison de l'objet puis forme des zones plus ou moins illuminées. La sortie de cette étape, *Shading Pass*, est le résultat du rendu réaliste. Voici l'aperçu de l'objet après les différents traitements effectués.

FIG. 6.7 – *ShadingPass*

Il est aussi possible de modifier quelques données pour personnaliser son rendu. Plusieurs onglets sont disponibles dans « Edit⟶Settings », et offre la possibilité de choisir une proportion pour la taille des splats, l'intervalle de choix pour la fonction gaussienne, ainsi que les paramètres du frustum.

Chapitre 7

Architecture

7.1 Diagramme de classes de l'application

(Voir Figure 7.1)
L'architecture de notre application se décompose en cinq modules (ou paquetages) : **Utils**, **FileLoader**, **Structure**, **Viewer** et **GUI**.

Le paquetage **Utils** contient un ensemble de classes et de structures représentant des données (vertex, face, surfel, ...) utilisées par les autres paquetages. **Vector3D** est une structure dont les attributs sont de type spécifiable lors de la déclaration d'un **Vector3D**. **Face**, **Vertex**, **Normal** et **Color** sont en fait des **Vector3D** avec un type spécifié. Les classes **FileUtils** et **ColorUtils** contiennent des méthodes statiques pour manipuler respectivement des fichiers et des couleurs.

Le paquetage **FileLoader** permet de lire un fichier décrivant un objet 3D et d'en extraire ses vertices et ses faces. Il se compose d'une interface, **FileLoaderInterface**, fournissant les méthodes publiques pour lire le fichier passé en paramètre et pour obtenir les vertices lus ainsi que les faces. Cette interface est implémentée par la classe **PlyFormatFile** qui charge les fichiers *ply*, binaire ou ASCII. Elle utilise pour cela les méthodes fournies par la librairie **ply**. Cette classe utilise également les services de la classe **FileUtils** afin de lever une exception si le fichier objet à charger n'existe pas. Grâce à l'interface, le chargement peut être étendu à d'autres types de fichier objet : il suffit d'écrire une nouvelle classe implémentant **FileLoaderInterface**.

Le module **Structure** s'occupe de la représentation de l'objet sous forme d'un arbre séquentiel de points (SPT), et de la sélection dans celui-ci des surfels à afficher à chaque rendu. Un objet SPT utilise les services fournis par **FileLoaderInterface** pour récupérer les vertices et faces du fichier objet afin de

remplir sa structure. Il utilise aussi les méthodes de **FileUtils** pour lire les fichiers contenant les shaders et pour propager l'exception engendrée lors de la tentative d'ouverture d'un fichier objet au niveau de la classe **PlyFormatFile**, ainsi que celles de **ColorUtils** pour attribuer à chaque surfel une couleur en fonction de sa position dans l'arbre. Cette classe implémente l'interface **SurfelsStructureInterface**, manipulée par le visualiseur. Ainsi, notre arbre séquentiel de points peut être remplacé par toute autre structure fournissant les mêmes services. La sélection dans **SurfelsStructureInterface** est en partie réalisée par le GPU.

Le paquetage **Viewer** permet de visualiser l'objet. Il est constitué de la classe **Viewer** héritant de **QGLViewer**, classe que nous avons reprise car elle fournit de base une grande partie des fonctionnalitées attendues.

Le module **GUI** contient toutes les classes utiles à l'interface graphique : **LargeObjectsApplication** est l'application principale, et **StructureThread** gère en parallèle la construction de la structure de données représentant l'objet.

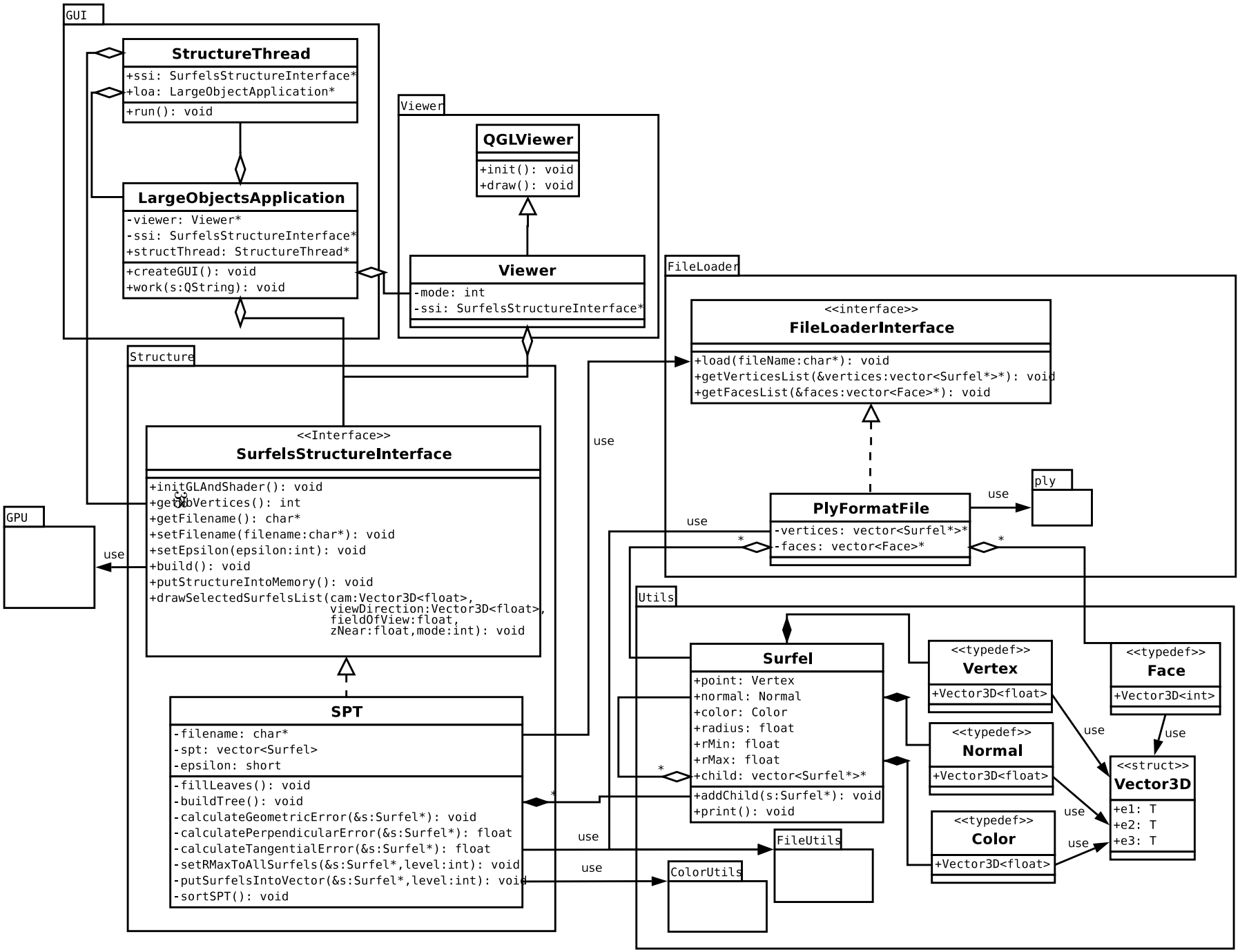


Fig. 7.1 – Diagramme de classes de l'application

7.2 Étapes du projet

Dans cette section nous traiterons une à une les trois étapes qui composent ce projet.

7.2.1 Lecture et chargement du fichier *ply* dans la structure

Cette partie consiste à implémenter les 5 modules **Utils**, **FileLoader**, **Structure**, **Viewer** et **GUI**.

(Voir Figure 7.2)
L'utilisateur ouvre un fichier de modèle *ply* par l'intermédiaire de l'interface graphique **LargeObjectsApplication**. Celle-ci crée une structure adaptée **SurfelsStructureInterface**, qui sera dans ce projet un arbre séquentiel de points **SPT**. L'interface lance ensuite la construction du SPT. Celui-ci utilise les méthodes fournies par **FileLoaderInterface** pour obtenir les vertices et les faces de l'objet représenté par le *ply*. Puis, à partir de ces données, on réalise les étapes décrites dans l'article [DVS03].

(Voir Figure 7.3)
Après que l'utilisateur ait modifié l'affichage de l'objet (rotation, déplacement, luminosité, ...), **LargeObjectsApplication** envoie un signal à la classe **Viewer**, chargée de la visualisation. Celle-ci demande à la structure **SurfelsStructureInterface** les surfels à rendre. Cette dernière va tout d'abord «élaguer» grossièrement la liste des surfels. Puis, cette sélection est affinée au niveau du GPU, dans le *Vertex Shader*. Enfin, la liste selectionnée est envoyée au module chargé du rendu qui s'occupe de l'affichage.

7.2.2 Rendu réaliste de l'objet

Comme nous l'avions précisé dans notre mémoire intermédiaire, la deuxième partie de notre projet consiste à réaliser un plugin du logiciel *Pointshop3D*. Cela nous permettra de mettre en oeuvre le rendu réaliste (*Deferred Shading* et approximation EWA) car tous les objets de la scène sont disponibles et modifiables facilement à l'aide de l'API de *Pointshop3D*.

(Voir Figure 7.4)
Ce logiciel possède une interface **RendererInterface** qui permet de lui ajouter une technique de rendu. Il suffit alors d'implémenter celle-ci pour créer notre propre rendu. C'est la classe **HQSSRenderer** qui fait ce travail. L'API concernant l'interface est riche : elle rend possible la manipulation des surfels, l'association des shaders ainsi que le rendu *multipass*. C'est la raison pour laquelle *Pointshop3D* nous a paru être un bon outil pour travailler sans que nous ayons à nous préoccuper du chargement ou de la création de surfels.

HQSSRenderer possède une instance de **HQSSRendererWidget** et une autre de **HQSSRendererConfiguration**. La première hérite de **QGLWidget** et contiendra toutes les fonctions nécessaires à l'affichage (la modification et manipulation des surfels) s'effectuent par le logiciel *PointShop3D*. La différence se situe au niveau des *shaders*. La seconde implémente l'interface **RendererConfigurationInterface**. Elle contient, comme son nom l'indique, toutes les méthodes concernant la configuration du rendu.

Ce module pourra dès lors être compilé séparément, puis associé au logiciel en le plaçant dans un répertoire spécifique.

(Voir Figure 7.5) Le diagramme de séquence ci-dessous décrit les étapes successives qui interviennent lors du *Deferred Shading*. Lorsque la scène doit être redessinée, le logiciel *Pointshop3D* envoie un signal au plugin de rendu qui se charge d'effectuer ce travail. Dans un premier temps, les surfels sont envoyés au GPU de manière à gagner du temps en évitant trop d'aller-retours entre le GPU et le CPU. Le rendu se fait clairement en trois étapes, ce qui permet de supprimer certains calculs trop lents et inutiles.

Le premier passage consiste seulement à remplir le *depth buffer*. Le travail du *Vertex Shader* est de calculer la taille du splat à rastériser. Le *Fragment Shader* s'occupe alors de remplir le *depth buffer*, mais en ayant auparavant légèrement éloigné l'objet du point de vue de façon à garantir une bonne accumulation des propriétés matérielles de l'objet lors de l'étape suivante.

Durant le second passage, l'approximation EWA est prise en compte pour calculer la taille des splats, et ainsi éviter l'aliasing. C'est dans le *Fragment Shader* que les couleurs et les normales sont accumulées. Les pixels qui sont trop éloignés ou cachés ne sont pas pris en compte dans ces calculs de *blending* car leur position est testée à l'aide du *depth buffer* du passage précédent.

Lors du dernier passage, nous avons alors en notre possession trois textures : la première contient les coordonnées de profondeur, et les deux autres possèdent les informations relatives aux couleurs et aux normales de l'objet. Il suffit alors de dessiner un rectangle de la taille de l'écran sur lequel on plaque et combine les textures pour obtenir l'objet final. C'est ici que les calculs concernant la lumière (ou autre shading), qui prennent énormément de temps, sont effectués. Cette méthode garantit qu'un seul calcul par pixel est fait, au lieu d'une dizaine pour un simple rendu.

7.2.3 Combinaison de la structure et du rendu réaliste

Le but de cette partie est tout d'abord d'implémenter le module **Viewer** final, auquel on a intégré le rendu réaliste développé dans le plugin *Pointshop3D*. **Viewer** sera ensuite inclus dans la classe **LargeObjectsApplication**, qui utilisera le module **Structure** pour sélectionner les points à afficher.

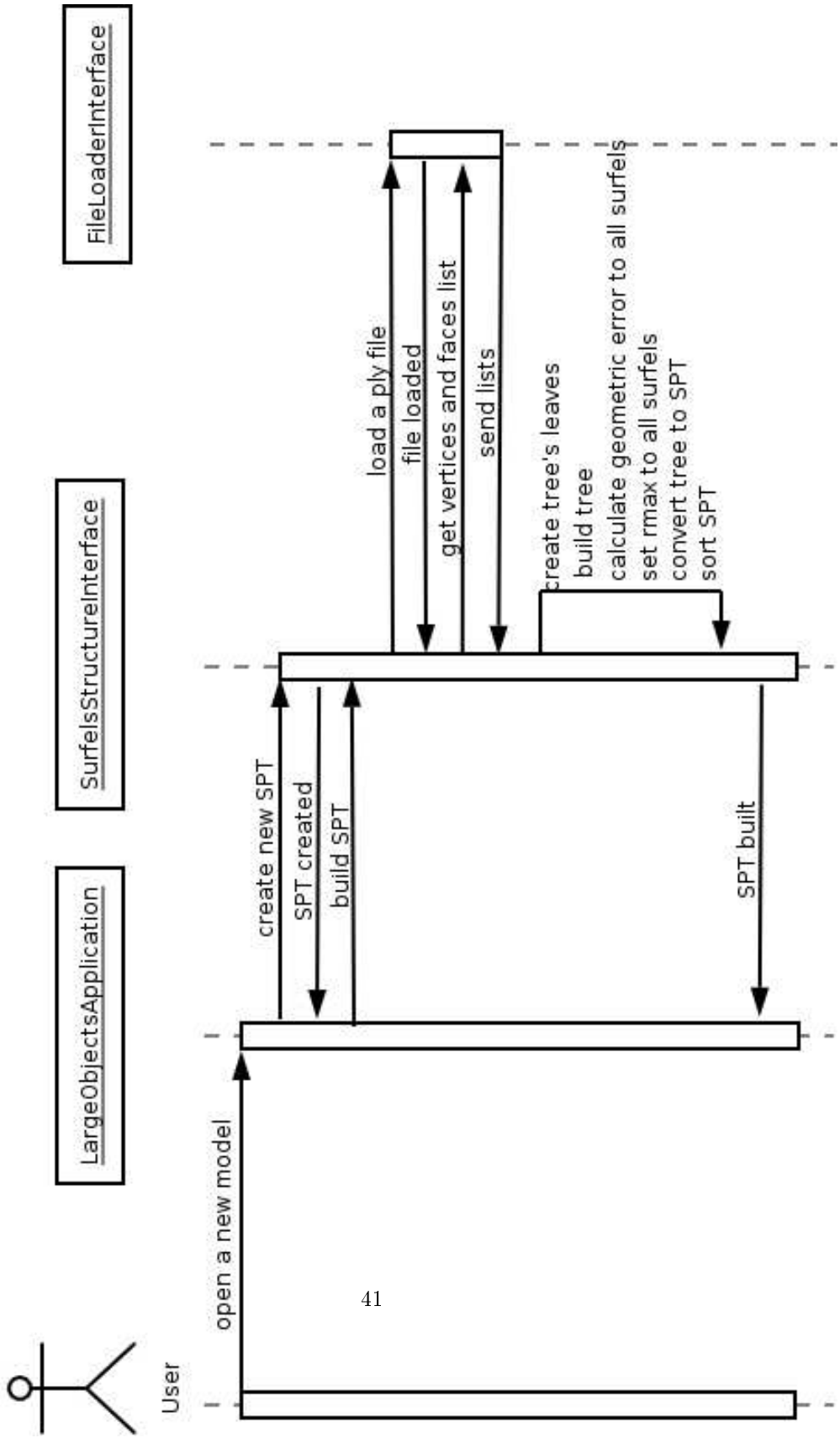


FIG. 7.2 – Diagramme de séquence : chargement d'un fichier *ply*

FIG. 7.3 – Diagramme de séquence : rendu basique

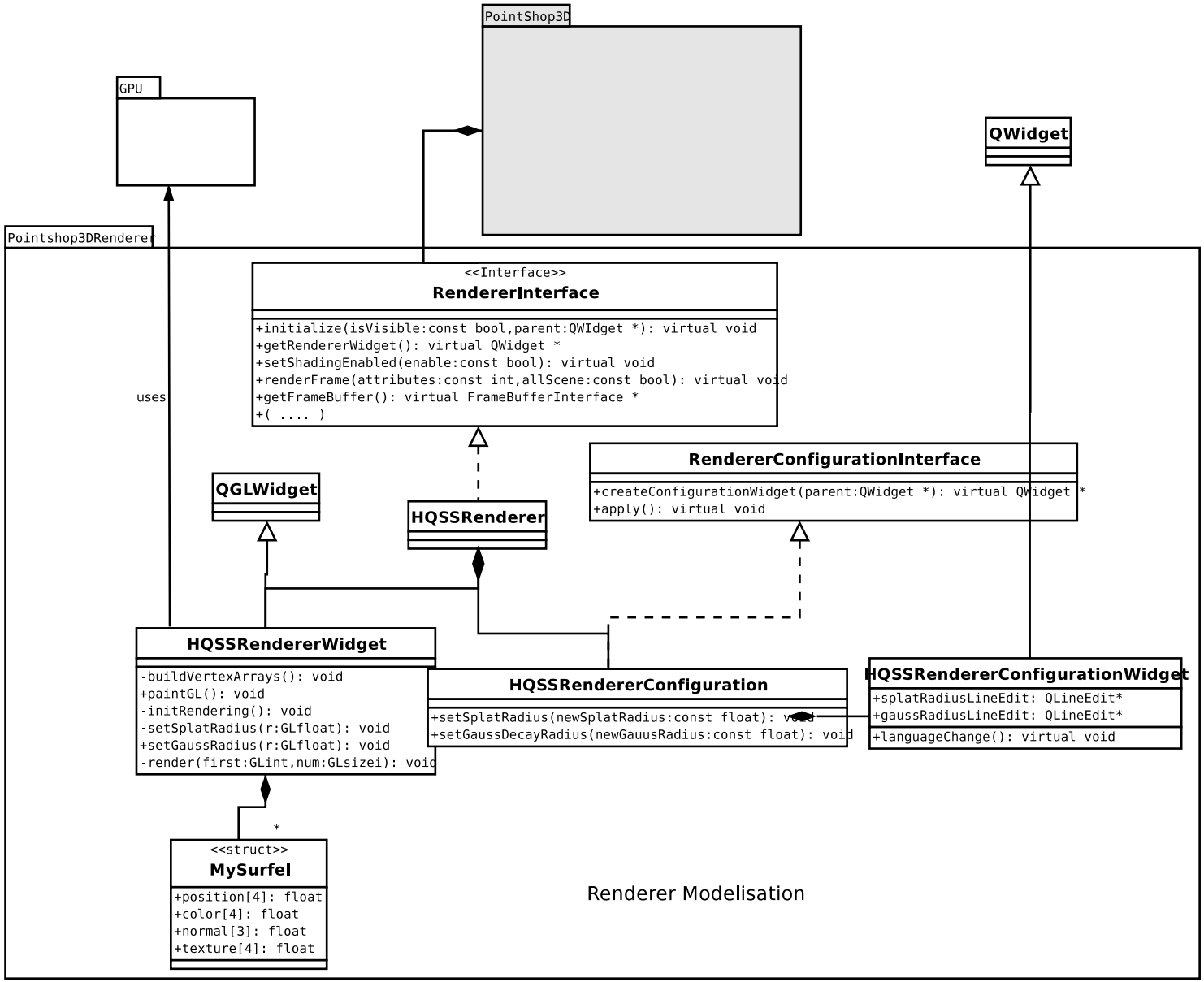


FIG. 7.4 – Diagramme de classe : plugin Pointshop3D de rendu réaliste de l'objet

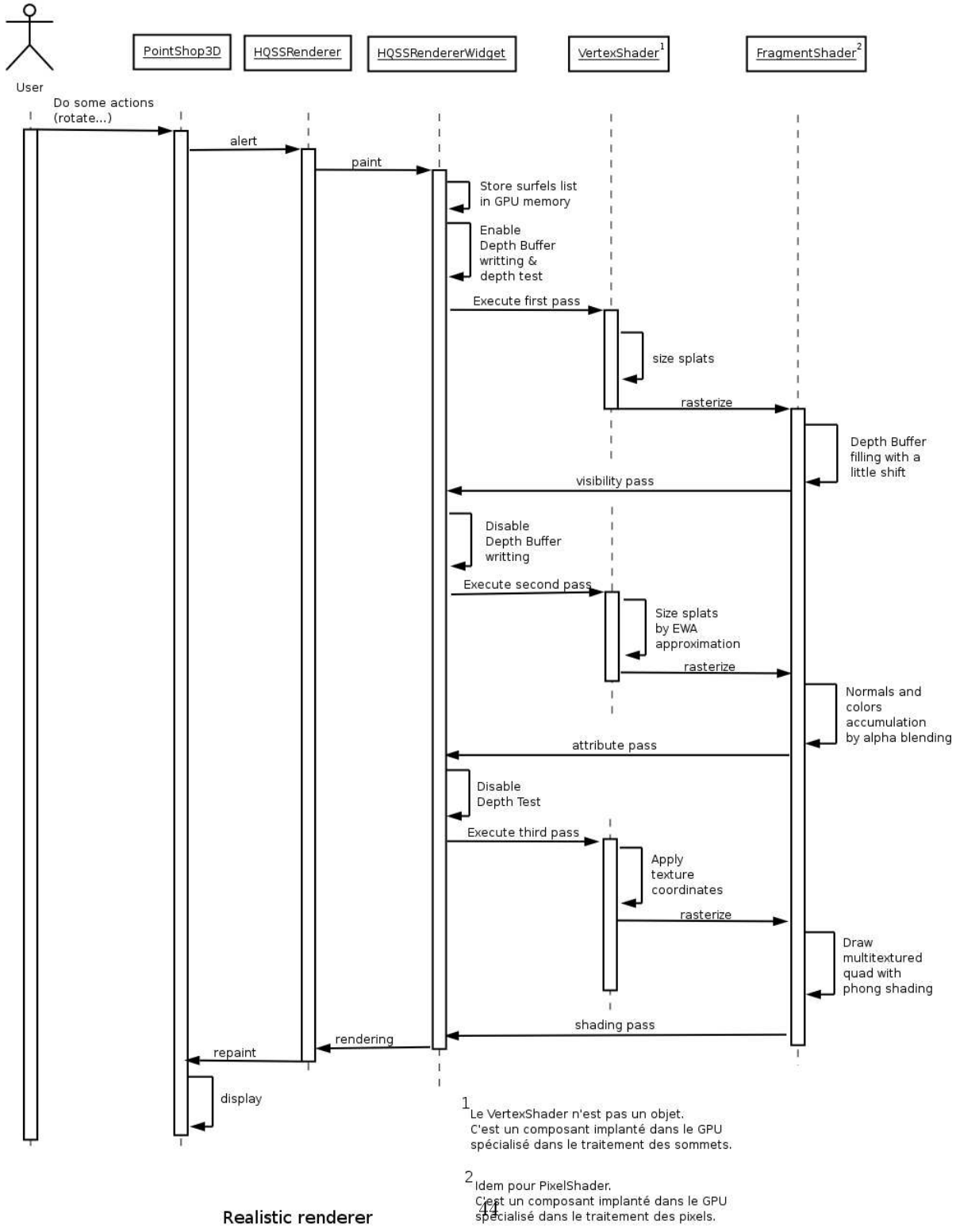


FIG. 7.5 – Diagramme de séquence : plugin Pointshop3D de rendu réaliste de l'objet

Chapitre 8

Algorithmes, structures de données et complexité

8.1 LargeObjectsViewer

L'arbre séquentiel de points que nous avons réalisé est une implémentation de la structure de données explicitée dans l'article [DVS03]. Cette implémentation se décompose en plusieurs étapes :

- construction d'un arbre (quadtree) pour représenter en mémoire les surfels composant l'objet ;
- pour chaque noeud de cet arbre, calcul de son erreur géométrique ;
- séquentialisation de l'arbre : on transforme l'arbre en une liste, plus facilement traitable par le GPU ;
- à chaque rendu, sélection des surfels de la liste à afficher.

Nous allons dans la suite de cette section décrire en détail les algorithmes et les structures de données utilisés dans chacune de ces étapes.

Pour le calcul de la complexité, posons fa le nombre de faces de l'objet, ve son nombre de vertices, et su le nombre de surfels.

8.1.1 Construction de l'arbre

Une grande partie du code concernant l'élaboration de cet arbre a été reprise à partir de celui de *QSplat*, puis adaptée à notre situation. Nous créons d'abord les surfels feuilles de l'arbre à partir de la liste des vertices et des faces de l'objet, puis nous calculons le reste de l'arborescence à partir de ces feuilles.

Création des surfels feuilles

Tout d'abord, nous instancions un objet **PlyFormatFile** afin d'obtenir les vertices et les faces du modèle. Cette instance stocke celles-ci dans deux ta-

bleaux, un pour les faces et un pour les vertices, qui sont en fait des pointeurs vers respectivement un **std : :vector** de **Face** et un **std : :vector** de pointeur de **Surfel**. Le stockage de pointeurs de **Surfel** et non de **Surfel** sera justifié plus avant dans l'explication. Nous récupérons ensuite les pointeurs vers ces listes. Nous créons un surfel feuille par vertex, mais la connaissance des faces est nécessaire pour calculer la normale et la taille des surfels.

La première étape de la création des surfels feuilles est le calcul de leur normale. Pour cela, on parcourt chaque face et on calcule sa normale (qui est le résultat du produit vectoriel entre deux vecteurs directeurs des côtés de la face). Cette valeur de normale est ajoutée à celle des surfels sommets de la face (qui est initialisée avant la boucle à 0). Ainsi, à la fin de la boucle, chaque surfel a sa normale positionnée, correspondant à la moyenne des normales des faces auxquelles elle appartient.

La deuxième étape consiste à calculer le rayon des surfels feuilles. On commence pour cela par initialiser ce rayon à 0 pour toutes les feuilles. Puis on parcourt chaque face. Pour chacune d'entre elles, on calcule un rayon r tel que deux surfels d'un tel rayon placés à deux des sommets (quelconques) de la face se touchent. Les surfels sommets de la face reçoivent comme valeur de rayon le maximum entre r et leur valeur courante. Ceci est une méthode conservatrice : elle peut engendrer des surfels trop larges, mais elle garantit qu'à la fin de la boucle il n'y ait aucun trou entre les surfels.

Remarque : nous avions dans un premier temps créé chaque surfel feuille à partir des faces, le surfel étant le cercle circonscrit de la face, mais nous avons abandonné cette idée pour deux raisons :

- il y a en moyenne deux fois plus de faces dans l'objet que de vertices, ce qui implique un arbre plus de deux fois plus gros : la mémoire étant une donnée critique dans notre application, cette alternative est catastrophique ;
- le calcul du cercle circonscrit est trop coûteux en terme de temps : il augmente énormément la durée de construction de la structure, ne nous permettant ainsi plus d'assurer les temps de chargement annoncés.

Construction du reste de l'arbre

Chaque surfel, pointeur vers une instance de la classe **Surfel**, contient, outre les attributs permettant de stocker les valeurs de son centre, de son rayon et de sa normale, un pointeur vers un **std : :vector** de pointeurs d'objets **Surfel**, qui seront ses fils dans l'arbre. Ainsi, lorsque l'arbre est créé, la classe n'a besoin de stocker que le pointeur vers la racine de l'arbre, car tout le reste de l'arborescence peut être retrouvé à partir de ce dernier. La construction de l'arbre n'est donc qu'une manipulation de pointeur, il n'y a pas de duplication ou de recopie inutile de données. En ce qui concerne le stockage des fils pour chaque surfel, nous avons choisi un pointeur vers un **std : :vector** plutôt qu'un **std : :vector** car cela

nous permet, pour les feuilles qui n'ont par définition pas de fils, de ne pas allouer ce tableau, ce qui économise quelques octets précieux.

Cette partie de l'algorithme est récursive. Nous considérons tout d'abord le tableau entier de feuilles, que nous divisons en deux parties. Pour cela, nous calculons la boîte englobant ces surfels, puis nous coupons cette boîte en deux au niveau du milieu du côté le plus long de celle-ci. Nous trions alors le tableau de telle façon que les surfels présents dans une même moitié de la boîte se trouve dans la même partie de tableau. Puis nous réexécutons l'algorithme récursivement sur chaque partie du tableau jusqu'à ce que la partie traitée contienne quatre surfels ou moins (car notre arbre est un quadtree). Dans ce cas, nous créons le noeud père de ces surfels de la manière suivante :

- son centre est l'équibarycentre des centres de ses fils ;
- sa normale est la moyenne des normales de ses fils ;
- son rayon est tel qu'il englobe ses surfels fils.

Puis, nous remontons récursivement pour construire le niveau supérieur de l'arborescence, et ainsi de suite. A la fin de l'algorithme, nous obtenons donc bien l'arbre souhaité.

Complexité

La lecture du fichier objet n'est pas considérée comme une étape de l'implémentation de la structure de données, et ne sera donc pas traitée.

- pour calculer les normales, on parcourt chaque face : la complexité est donc en $\mathrm{O}(fa)$;
- pour le calcul des rayons, on initialise d'abord chaque surfel feuille, puis on parcourt chaque face : la complexité est en $\mathrm{O}(ve + fa)$, soit en $\mathrm{O}(fa)$ (car il y a plus de faces que de vertices) ;
- lors de la construction de l'arbre, pour assembler quatre noeuds, il faut $log(su)$ opérations. Celles-ci sont réalisées «hauteur de l'arbre» fois, soit $log(su)$ fois. Au total, la complexité de la construction du reste de l'arbre est de $\mathrm{O}((log(su))^2)$.

La complexité totale de construction de l'arbre est donc en $\mathrm{O}(fa)$.

8.1.2 Calcul de l'erreur géométrique pour chaque surfel

Une fois l'arbre créé, nous calculons l'erreur géométrique de chacun de ses surfels, en faisant un parcours en profondeur de l'arbre. L'erreur géométrique e_g est composée de l'erreur perpendiculaire e_p et de l'erreur tangentielle e_t, tel que $e_g = \sqrt{e_p^2 + e_t^2}$. Nous allons étudier en détail comment ces deux erreurs sont calculées.

Calcul de l'erreur perpendiculaire

L'erreur perpendiculaire est la distance minimum entre deux plans parallèles au surfel, qui couvrent tous ses surfels fils. Elle est représentée ainsi :

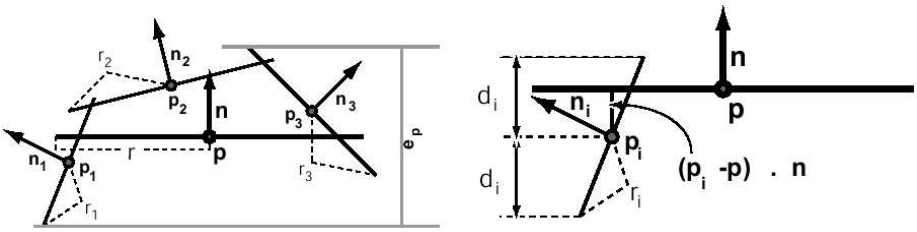


FIG. 8.1 – Erreur perpendiculaire

Pour calculer cette erreur, nous nous contentons d'appliquer la formule donnée dans l'article :

$$e_p = max[((p_i - p).n) + d_i] - min[((p_i - p).n) - d_i]$$

avec : $d_i = r_i\sqrt{1-(n_i.n)^2}$

p (resp. p_i) les coordonnées du centre du surfel traité s (resp. de son surfel fils i)

n (resp. n_i) les coordonnées de la normale de s (resp. de son surfel fils i)

Calcul de l'erreur tangentielle

L'erreur tangentielle mesure la zone inutile couverte par un surfel, c'est-à-dire celle qui ne correspond pas à un de ses surfels fils. Elle est représentée de la manière suivante :

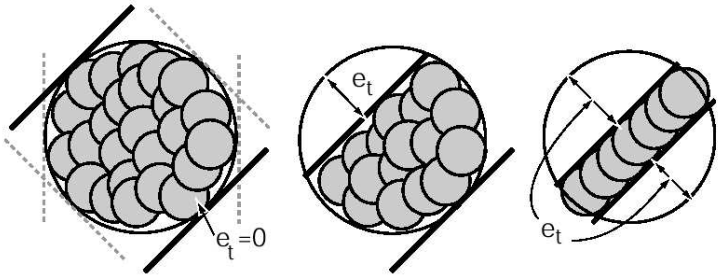


FIG. 8.2 – Erreur tangentielle

Pour la calculer, nous commençons par obtenir les vecteurs $\vec{n}$, $\vec{u}$ et $\vec{v}$, base d'un repère 3D orthogonal ayant pour origine le centre O du surfel.

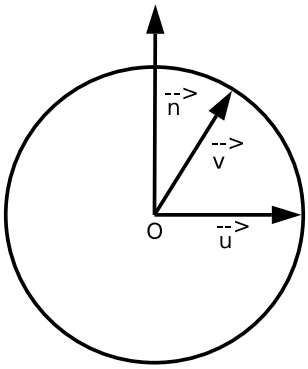


FIG. 8.3 – Repère (O, $\vec{u}$, $\vec{v}$, $\vec{n}$)

Le vecteur $\vec{n}$ est la normale du surfel, et les vecteurs $\vec{u}$ et $\vec{v}$ sont obtenus à partir de lui. Nous représentons ensuite les surfels fils du surfel traité dans le repère (O, $\vec{u}$, $\vec{v}$, $\vec{n}$), en calculant la matrice de changement de repère à l'aide des services fournis par la librairie **gsl**, puis nous projetons les surfels fils sur le plan défini par $\vec{u}$ et $\vec{v}$ (simplement en mettant à 0 la troisième coordonnée de ces surfels). On regarde alors sur quel axe du repère 2D (O, $\vec{u}$, $\vec{v}$) la distance entre les deux surfels fils extrémaux est minimale : l'erreur tangentielle est égale à la différence entre le diamètre du surfel traité et cette distance.

Remarque : pour que l'algorithme de sélection puisse fonctionner, il faut que l'erreur d'un surfel soit supérieure ou égale aux erreurs de ses fils dans l'arbre. Or, dans ce calcul de l'erreur géométrique, rien ne nous assure que cette condition soit respectée, et il se trouve effectivement qu'elle ne l'est pas forcément. C'est pour cela qu'à l'erreur calculée, nous ajoutons l'erreur maximale des fils du surfel traité.

Complexité

- pour calculer l'erreur perpendiculaire d'un surfel, il faut parcourir ses fils (qui sont au maximum au nombre de 4), soit une complexité en O(1) ;
- lors du calcul de l'erreur tangentielle d'un surfel, on détermine une matrice de changement de repère. Comme cette matrice est de taille bornée (3x3), la complexité est en O(1). On applique ensuite cette matrice sur les fils du surfel, soit là encore une complexité en O(1) ;
- le parcours en profondeur de l'arbre, pour appliquer l'erreur à tous les surfels, est de complexité O(su).

La complexité totale du calcul de l'erreur est donc en O(su).

8.1.3 Séquentialisation de l'arbre

A cette étape de la construction du SPT, nous avons un arbre comparable à celui de *QSplat*. L'algorithme de sélection dans celui-ci consiste à parcourir chaque ramification de l'arbre jusqu'à trouver un surfel (que l'on pourra rendre à l'écran) tel que son erreur divisée par la distance r entre le surfel et l'observateur soit inférieure à un certain seuil ϵ fixé par l'utilisateur. Or, cette procédure est récursive et ne convient pas à un traitement par le GPU. C'est pour cela qu'il faut réarranger cette arbre en une liste et remplacer le test récursif par une boucle dans la liste.

Simplification du test

Tout d'abord, à la place de l'erreur géométrique e_g pour chaque surfel, nous stockons $r_{min} = e_g/\epsilon$, ce qui simplifie le test précédent à $r > r_{min}$.

Transformation du test récursif en un test itératif

En réorganisant l'arbre en une liste, nous perdons la notion de hiérarchie nécessaire au test actuel. Nous avons donc besoin d'un test supplémentaire qui vérifie pour chaque surfel qu'aucun de ses ancêtres n'a déjà été sélectionné. Pour cela, nous ajoutons à chaque surfel l'attribut r_{max}, qui correspond au r_{min} de son parent direct (cela en réalisant un parcours en profondeur de l'arbre). Ainsi, le test de sélection d'un surfel devient $r \in [r_{min}, r_{max}]$.

Réarrangement de l'arbre en une liste

Notre liste est un **std : :vector** d'objets **Surfel**. Nous stockons ici des instances de classe et non des pointeurs vers celles-ci car pour la suite, non seulement les pointeurs ne nous sont d'aucune utilité et auraient donc gaspillés de la mémoire, mais en plus cette organisation permet d'utiliser simplement les *Vertex Buffer Objects*, comme nous le verrons.

Pour remplir la liste, il suffit de parcourir l'arbre en profondeur et d'ajouter chaque surfel à la liste (en n'oubliant pas de supprimer de l'arbre au fur et à mesure chaque surfel traité).

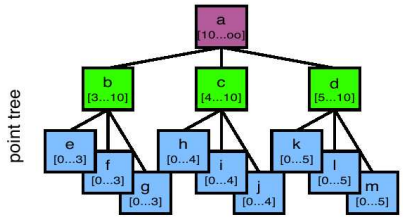


FIG. 8.4 – Arbre de points (utilisé dans QSplat)[1]

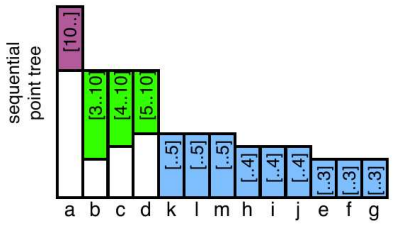


FIG. 8.5 – Arbre séquentiel de points[1]

Complexité

- pour ajouter l'attribut r_{max} à tous les surfels, on réalise un parcours en profondeur de l'arbre : la complexité est en $O(su)$;
- le réarrangement de l'arbre en une liste est également un parcours en profondeur : la complexité est en $O(su)$;

La complexité totale de la séquentialisation est donc en $O(su)$.

8.1.4 Sélection des surfels à afficher

La sélection dans le SPT est réalisée en partie au niveau du CPU, en partie au niveau du GPU : le CPU va d'abord effectuer une présélection qui va être transmise au GPU, ce dernier réalisant la sélection fine.

Etapes préliminaires

Avant de pouvoir effectuer la sélection, il nous faut trier le SPT par ordre décroissant des r_{max}. Pour cela, nous utilisons la fonction **sort()** fournie par la *STL*, qui est de complexité $O(su * log(su))$, soit la meilleure complexité «dans le pire des cas» connue.

Nous plaçons ensuite le contenu du SPT en mémoire video, à l'aide d'un *Vertex Buffer Object* (*VBO*). Ce mécanisme récent permet de stocker directement certaines données de dessin en mémoire vidéo, ce qui minimise les transferts CPU $\longleftrightarrow$ GPU. Nous positionnons ensuite des pointeurs sur ce VBO, afin d'indiquer au GPU où trouver les données qui l'intéressent. Nous positionnons donc le pointeur sur les coordonnées des centres des surfels (**glVertexPointer()**), sur leur normale (**glNormalPointer()**) et sur leur couleur (**glColorPointer()**). Un dernier pointeur peut être placé, celui sur les coordonnées de texture. Comme nous ne gérons pas de texture, nous positionnons ce pointeur sur le rayon (radius), r_{min} et r_{max}, afin que ceux-ci soient disponibles au niveau du GPU pour la sélection fine.

[1] Dachsbacher, C., Vogelgsang,C. and Stamminger, M..«Conversion of a point tree into a sequential point tree». *Sequential Point Trees*

Remarque :
- le fait d'avoir choisi une telle architecture pour le SPT nous permet de pouvoir l'utiliser directement avec le *VBO*, sans avoir à construire une structure intermédiaire ;
- dans un premier temps nous ne stockions en mémoire vidéo que la partie du SPT présélectionnée, et donc nous réalisions ce stockage à chaque rendu. Cette méthode permettait d'économiser de la mémoire vidéo mais le temps de transfert entre la mémoire vive et la mémoire vidéo étant très important avec nos objets, le rendu devenait extrêmement lent.

Présélection par le CPU

A chaque rendu, le CPU commence par calculer une borne inférieure pour r (appelée $min(r)$), qui est la distance entre l'observateur et la boîte englobante de l'objet (qui est obtenue plus tôt dans l'algorithme, après avoir construit les feuilles de l'arbre). Puis, il cherche dans le SPT (précédemment trié par r_{max}) le premier surfel S tel que $r_{max} \leq min(r)$. Cette recherche est effectuée par la fonction **upper_bound()** fournie par la *STL*, qui effectue une recherche dichotomique. Le début de la liste jusqu'à ce surfel S est alors envoyé au GPU par l'intermédiaire de la fonction **glDrawArrays()**, en signalant que le contenu du VBO doit être traité par le pipeline graphique jusqu'à S.

Sélection fine par le GPU

Cette sélection est réalisée au niveau du *Vertex Shader*. Pour chaque surfel préselectionné, celui-ci commence par calculer r, à partir de la position de la caméra passée au *Vertex Shader* par l'intermédiaire d'une variable uniforme. Puis il réalise le test de sélection : $r \in [r_{min}, r_{max}]$, à partir des r_{min} et r_{max} récupérés dans les coordonnées de la texture. Les surfels réussissant ce test seront rendus, alors que les autres seront éliminés (en positionnant la coordonnée homogène w de ceux-ci à 0, ce qui a pour effet de les déplacer à l'infini).

Complexité

Le coût des étapes préliminaires est le coût du tri, soit une complexité en $O(su * log(su))$. Pour chaque rendu :
- la présélection est réalisée à l'aide d'une recherche dichotomique, d'une complexité en $O(log(su))$.
- nous pouvons donner une borne supérieure pour le nombre de surfels envoyés au GPU ($O(su)$, mais ensuite certains surfels sont rejetés, dépendant de la position de l'objet par rapport à l'observateur. Nous ne pouvons donc pas déterminer exactement la complexité de la sélection par le GPU.

La complexité totale de la sélection est donc en $O(log(su))$.

8.2 HQSSRenderer

8.2.1 Communication avec le GPU

Mécanisme de communication

Nous utilisons une structure de données spécifique pour communiquer avec le processeur graphique : les VBO. Il s'agit d'une extension OpenGL permettant un mécanisme rapide de transfert de données entre le CPU et le GPU. Au niveau CPU, nous disposons d'un tableau dont chaque case contient les attributs caractéristiques d'un surfel. Les VBO permettent d'envoyer ce tableau directement au GPU, en précisant seulement où chercher les éléments à l'aide de pointeurs. Cette méthode est actuellement la meilleure, et dépasse de loin les performances obtenues par les moyens traditionnels comme l'utilisation des **glBegin()-glEnd()** entre lesquels il faut définir chaque sommet, ou même des **glArrayElement()** offerts par OpenGL.

Transfert des données

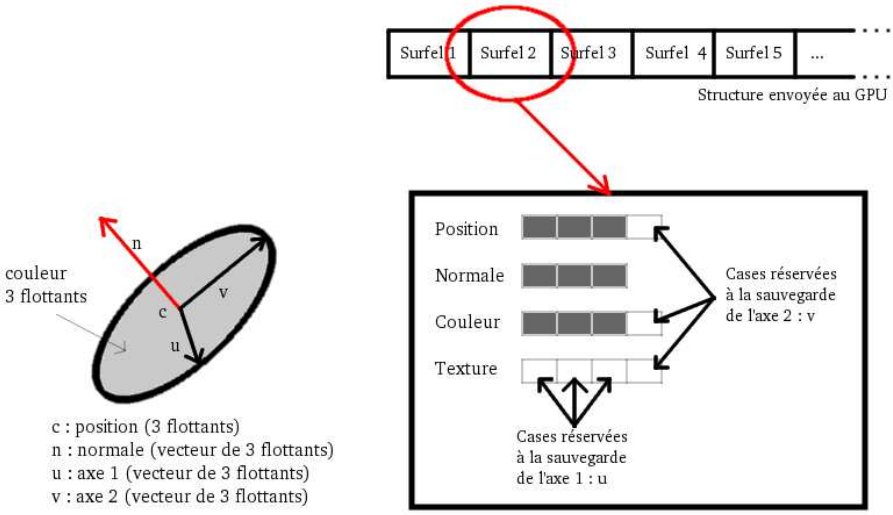


FIG. 8.6 – A gauche : représentation d'un surfel, à droite : structure de données et sauvegarde des surfels

En utilisant une telle méthode, il n'est possible de transférer que des données spécifiques d'OpenGL vers le GPU. En effet, nous ne pouvons faire passer que les données concernant la position, la couleur, la normale et la texture d'un sommet avec les VBOs. Or, nous avons besoin de récupérer d'autres éléments tels que les axes des surfels de façon à ce que nous puissions calculer leur taille et leur forme dans les shaders (figure 8.6 à gauche). Nous avons donc recours

à une astuce : nous introduisons nos données dans les structures d'OpenGL. Cette technique permet de conserver la rapidité du transfert, tout en ayant la possibilité de récupérer les éléments indispensables à l'affichage des surfels. La figure 8.6 à droite montre comment sont stockés les éléments d'un surfel dans la structure de données passée à OpenGL.

8.2.2 Affichage d'un surfel

Une fois que nous avons récupéré la structure de chaque point dans le shader, le travail consiste à les afficher avec leur bonne taille pour éviter les trous dans l'objet, mais aussi sous forme d'ellipse car ils se trouvent dans un environnement 3D.

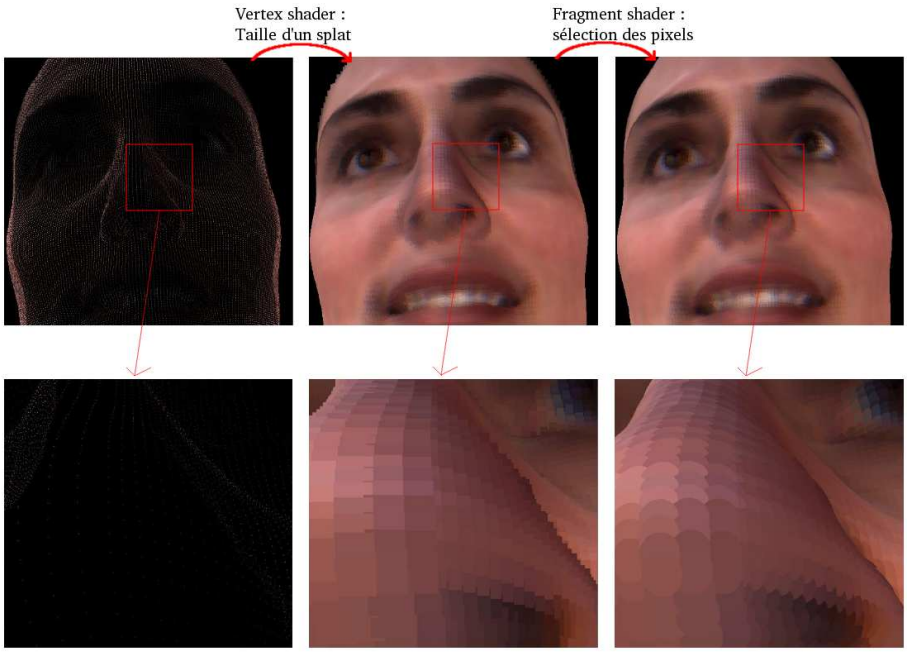


FIG. 8.7 – A gauche : affichage simple des points reçus avec leur couleur, au centre : affichage du carré de taille $size * size$, à droite : affichage des ellipses correspondant aux surfels projetés dans un environnement 3D

Taille d'un point

Grâce à la manipulation de la structure de données décrite précédemment, nous récupérons les attributs caractéristiques d'un surfel dans le *Vertex Shader* (position, normale, couleur, axe 1 et 2). Son travail est de calculer la taille du carré à rastériser correspondant au surfel (voir figure 8.7 au centre). La formule est donnée dans l'article [BSK04] et n'est en fait qu'une estimation de la valeur

exacte de manière à simplifier les calculs.

$$size = 2r.\frac{n}{z_{eye}}.\frac{h_{vp}}{t-b}$$

r est le rayon du surfel, z_{eye} la valeur de profondeur de son centre en coordonnées de vue, h_{vp} la hauteur du *viewport*, et les variables n, t, b correspondent respectivement aux paramètres du volume de vision. n est donc la distance du plan proche, et $t - b$ sa hauteur. Mis à part le rayon qui est déjà présent dans les shaders, ces variables ne changent pas durant toute la durée d'un affichage et sont donc très facilement transférables vers le GPU. Il existe des méthodes OpenGL spécifiques pour cela.

Création de l'ellipse

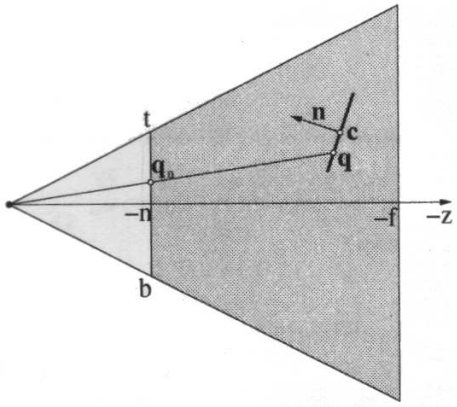


FIG. 8.8 – Calcul du point q correspondant au pixel donné en projetant le point $\mathbf{q}_n$ sur le plan proche et en déterminant son intersection avec le plan du splat dans l'espace de vue[2]

Une fois que la taille des carrés est calculée, il convient de vérifier dans le *Fragment Shader* si chacun des pixels est à l'intérieur d'un splat ou non. En effet, les affichages se font dans un environnement 3D et correspondent donc à des ellipses (voir figure 8.7 à droite). La méthode consiste tout d'abord à calculer le point $\mathbf{q}_n$ qui est la position du pixel sur le plan proche (voir figure 8.8). La formule, donnée dans [BSK04], correspond à l'inversion de la transformation du *viewport* :

$$\mathbf{q}_n = \begin{pmatrix} x.\frac{w_n}{w_{vp}} - \frac{w_n}{2} \\ y.\frac{h_n}{h_{vp}} - \frac{h_n}{2} \\ -n \end{pmatrix}$$

[2]Image tirée de l'article [BSK04]

Les paramètres x et y sont les coordonnées du pixel courant, n la distance du plan proche, et $w_{\{n,vp\}}$ et $h_{\{n,vp\}}$ sont respectivement la largeur et la hauteur du plan proche et du *viewport*.

La projection $\mathbf{q}$ de ce point sur le splat en coordonnées de vue peut alors être calculée de la manière suivante :

$$\mathbf{q} = \mathbf{q}_n \cdot \frac{\mathbf{c}^T \mathbf{n}}{\mathbf{q}_n^T \mathbf{n}}$$

c et n sont le centre et la normale du splat en coordonnées de vue.

Il devient alors possible de déterminer si le point $\mathbf{q}$ est à l'intérieur du splat en calculant ses paramètres (u, v), et en vérifiant que la somme de leurs carrés est inférieure ou égale à 1. Un pixel doit donc être affiché si son point $\mathbf{q}$ associé satisfait la condition suivante :

$$u^2 + v^2 = \left(\mathbf{u}_j^T (\mathbf{q} - \mathbf{c}_j)\right)^2 + \left(\mathbf{v}_j^T (\mathbf{q} - \mathbf{c}_j)\right)^2 \leq 1$$

c_j, u_j et v_j correspondent au centre et aux deux axes du surfel en coordonnées de vue. Ces paramètres sont calculés dans le *Vertex Shader*, et sont ensuite envoyés au *Fragment Shader*. Si un pixel est accepté, il est affiché. Sinon, il est rejeté et ne sera donc pas pris en compte dans la suite du pipeline graphique. Nous obtenons ainsi des ellipses, de couleurs et de tailles spécifiques, comme le montre la figure 8.7 à droite.

8.2.3 Deferred Shading

Le *Deferred Shading* est une technique de rendu efficace car les calculs prenant du temps comme ceux de la lumière ne sont effectués qu'une seule fois pour chaque pixel. La méthode consiste à faire trois passages dans le GPU, chacun ayant une spécificité pour afficher la scène. Le schéma 8.9 les illustre.

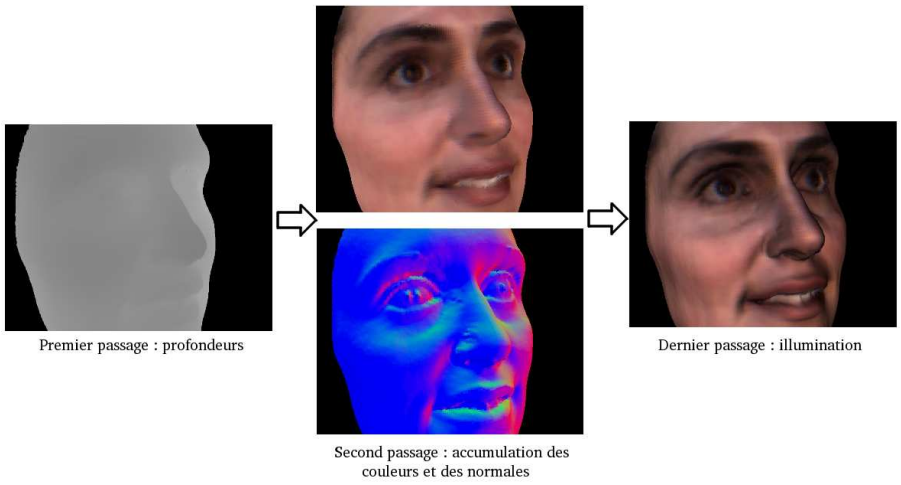


FIG. 8.9 – Déroulement du *Deferred Shading*

Profondeur de l'objet

FIG. 8.10 – remplissage du *Depth Buffer*

Le premier passage consiste donc seulement à remplir le buffer de profondeur (figure 8.10). La taille des splats est calculée dans le *Vertex Shader* et le test de sélection des fragments est effectué dans le *Fragment Shader* de la manière expliquée dans le paragraphe précédent. Il reste à écrire la profondeur du fragment dans le tampon qui y est consacré. Nous nous servirons pour cela de la variable $\mathbf{q}$ calculée précédemment qui correspond à la position du fragment en coordonnées de vue. La coordonnée en z est corrigée comme décrit dans l'article [BSK04] de manière à ce qu'elle prenne en compte l'ensemble du surfel :

$$z_{vp} = \frac{1}{\mathbf{q}_z}.\frac{fn}{f-n} + \frac{f}{f-n}$$

En effet, cette coordonnée correspond au centre du splat. Sans la corriger, il y aurait de grandes chances pour que lors de l'affichage, nous ne voyions que la moitié du surfel : dans l'espace 3D, la moitié des pixels d'un surfel se situe devant celui de son centre, l'autre moitié étant derrière.

Avant cela, une dernière modification doit être apportée à la valeur de $\mathbf{q}_z$. Lors de l'accumulation des propriétés matérielles (couleurs et normales), les splats qui se superposent doivent se mélanger de façon à ce que l'oeil ne voit que le moins possible la transition entre les disques. Cette opération ne peut pas se faire avec la seule équation précédente, car la profondeur ne peut être sauvegardée que pour le surfel le plus proche du point de vue. Il faut donc éloigner légèrement l'objet du point de vue de manière à ce que les valeurs de profondeur laissent passer les surfels qui se recouvrent. La valeur de $\mathbf{q}_z$ est donc augmentée légèrement d'une valeur *depthOffset*, calculée en fonction de la matrice de transformation et du rayon moyen des surfels.

Accumulation des propriétés matérielles

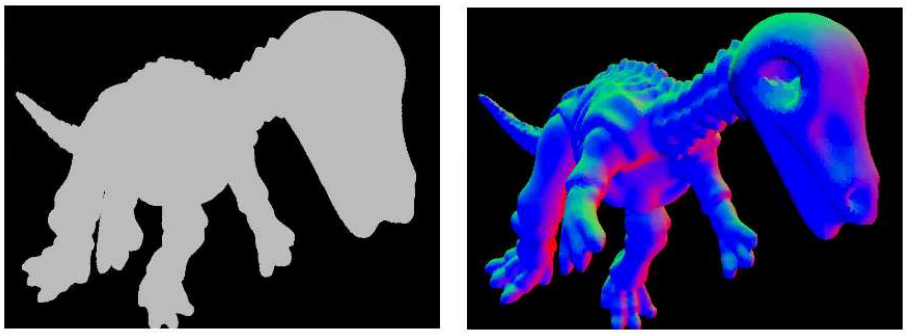


FIG. 8.11 – remplissage des textures «couleurs et normales»

Le second passage consiste à sauvegarder les attributs de l'objet. Nous remplissons les buffers avec les éléments qui nous intéressent (couleurs et normales), puis nous les récupérons sous forme de textures dans le programme principal car ils nous sont utiles pour la dernière étape (figure 8.11). C'est à ce moment qu'entre en jeu le tampon de profondeur que nous avons rempli lors du premier passage. Nous gardons seulement les attributs des pixels qui passent le test de pronfondeur.

Il est aussi nécessaire de superposer les attributs pour les splats qui se chevauchent. Pour cela, nous calculons un poids pour chaque pixel qui détermine la position du pixel dans le surfel. Plus le poids est élevé, plus la couleur du fragment sera prise en compte dans la couleur finale du pixel. Pour calculer ce poids, nous nous servons d'une fonction gaussienne :

$$h(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

Typiquement, le paramètre σ est égal à 1 et l'intervalle de x est initialisé à [0..2] mais peut être modifié par l'utilisateur. Le poids est alors calculé de la manière décrite dans [BHZK05] à l'aide des valeurs u et v vues précédemment :

$$w(x, y) = h\left(\sqrt{u^2 + v^2}\right)$$

Le mélange des couleurs est une opération délicate car il est nécessaire que la transition entre deux surfels soit la moins visible possible pour l'utilisateur. Nous utilisons l'*Alpha Blending* proposé par OpenGL pour effectuer cette opération et nous appliquons les couleurs dans le *Fragment Shader*. Nous avons choisi les paramètres de telle sorte que si la couleur du fragment source est définie par les quatre composantes (S_r, S_g, S_b, S_a), et celle du pixel de destination par (D_r, D_g, D_b, D_a), alors la couleur finale sera :

$$C = (S_r.S_a + D_r, S_g.S_a + D_g, S_b.S_a + D_b, S_a + D_a)$$

Sachant que nous utilisons le poids w comme composante *alpha*, on a alors $S_a = w(x, y)$, et nous pouvons donc remplacer dans l'équation précédente :

$$C = (S_r.w(x, y) + D_r, S_g.w(x, y) + D_g, S_b.w(x, y) + D_b, w(x, y) + D_a)$$

Cette méthode garantit que les composantes rouge, verte et bleue seront comprises entre 0 et 1 (en choisissant des paramètres corrects pour la gaussienne), et qu'il n'y aura ainsi pas de saturation dans la couleur finale.

Rendu de la scène

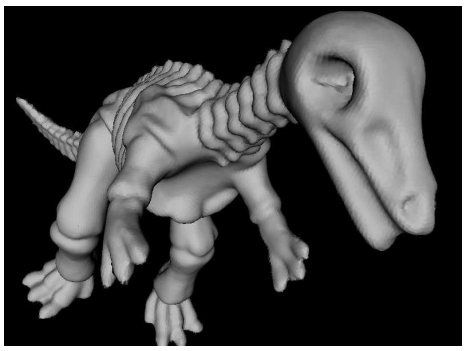


FIG. 8.12 – illumination

Le dernier passage consiste à effectuer le rendu de la scène. A cet instant, nous avons donc en notre possession d'une part le tampon de profondeur, et d'autre part les deux textures qui correspondent respectivement aux couleurs et aux normales de l'objet. La méthode utilisée est de dessiner un rectangle de la taille du *viewport*, puis d'utiliser le multitexturage pour récupérer nos attributs au niveau du GPU. Avant toute chose, il faut normaliser les attributs que nous récupérons dans les textures. L'*Alpha-Blending* utilisé précédemment ne garantit pas que la valeur *alpha* de la couleur (resp. normale) finale obtenue ne soit pas égale à 1. Il faut donc diviser chaque composante (r,g,b) de la couleur (resp. normale) par leur composante *alpha*.

Nous pouvons dès lors calculer la luminosité de la scène avec un *per-pixel phong shading*. Pour réaliser ce calcul, il est nécessaire de connaître la distance entre le pixel courant et la source de lumière. Nous avons choisi pour cela de se servir du *Frame Buffer*, non utilisé lors du remplissage du tampon de profondeur, et d'y ajouter les positions des pixels de l'objet dans l'espace de vue. Nous récupérons ainsi une troisième texture, en plus de celles des couleurs et des normales.

Les calculs nécessaires au *Phong Shading* peuvent alors être mis en oeuvre, l'avantage étant qu'ils sont fait une seule fois par pixel.

8.2.4 Complexité

Il n'est pas possible de connaître la complexité exacte de notre rendu. En effet, certains pixels sont pris en compte alors que d'autres sont rejetés. Cette sélection dépend de la position et de la forme des surfels qui composent l'objet.

En outre, lors des deux premiers passages, la taille de chacun des splats est calculée. Plus cette dimension sera élevée, plus la rastérisation sera longue. Pour une taille d, $d * d$ fragments seront générés, et chacun d'entre eux seront traités dans le *Fragment Shader*. Or, il n'est pas possible de connaître à l'avance le nombre de fragments générés et leur taille moyenne. Cela dépend de l'éloignement, de la forme et de la position de l'objet dans la scène.

Toutefois, nous pouvons déterminer la complexité du dernier passage car il ne dépend pas de l'objet qui fait varier la taille des splats. Nous dessinons ici un rectangle de la taille du *viewport*. Il y a donc exactement $w_{vp} * h_{vp}$ fragments générés, w_{vp} et h_{vp} étants respectivement la largeur et la hauteur du *viewport*. La complexité s du dernier passage pour un objet composé de n surfels peut donc être donnée par :

$$\forall n, s(n) = \Theta(w_{vp} * h_{vp})$$

On peut constater que la complexité ne dépend pas de n. C'est l'avantage que procure le *Deferred Shading* qui ne fait qu'un calcul par pixel. Néanmoins, la valeur de la constante associée à la borne supérieure est relativement élevée car de nombreux calculs comme des produits scalaires sont nécessaires pour l'illumination de la scène.

Pour les deux autres passages, nous savons seulement que le programme passe autant de fois dans le *Vertex Shader* qu'il n'y a de points dans l'objet. Nous ne connaissons pas la taille du carré rastérisé. On peut donc seulement dire que la complexité v de ces deux passages est :

$$\forall n, v(n) = \Omega(n)$$

Connaissant les bornes asymptotiques inférieures de chacun des passages, on peut en déduire celle définissant le rendu total f :

$$\forall n, f(n) = \Omega(n)$$

En effet, nous ajoutons à la complexité des deux premiers passages celle de l'illumination qui, même élevée, reste une constante. On ne peut néanmoins rien dire de plus.

Chapitre 9

Tests

Tous les tests seront effectués sur une machine composée d'une carte graphique Nvidia 6600 GT, d'un processeur Intel Pentium IV 2,8GHz et de 1024Mo de mémoire vive, sauf mention contraire.

9.1 Tests unitaires

9.1.1 LargeObjectsViewer

Tout au long du déroulement du projet, nous avons effectué de nombreux tests sur les diverses classes et structures composant notre application. Nous avons conservé les tests les plus significatifs, que nous présentons dans la suite de cette section.

Les tests ont été réalisés dans des fichiers séparés, un fichier par paquetage. Chacun de ces fichiers est placé dans le répertoire *tests* de son paquetage. Pour effectuer chaque test, il suffit de se rendre dans le répertoire *tests/* correspondant, de compiler le fichier (commande : *qmake && make*, qui compile aussi les fichiers testés), puis de l'exécuter dans le répertoire *bin/* à la racine du répertoire *LargeObjectsViewer/* (commande : ./<le nom du paquetage en minuscule>).

Remarque : nous ne testons pas ici le paquetage **GUI**, car son test équivaut au test global de l'application.

Le paquetage Utils

Dans ce paquetage, nous testons la classe **Surfel**. Nous vérifions plus précisément le bon fonctionnement du constructeur par copie (en s'assurant que l'instance et sa copie aient la valeur de leurs champs égaux) et de la méthode **addChild()** (en contrôlant que les éléments ajoutés soient bien ceux que l'on récupère).

Exécution du test

```
./utils
Start Utils test......Utils test complete
```

Les éléments du paquetage **Utils** testés ont donc le comportement souhaité.

Le paquetage FileLoader

Nous contrôlons ici le fonctionnement de la classe **PlyFormatFile** et par la même occasion celui de l'interface **FileLoaderInterface**. Nous avons créé pour cela un fichier *ply* de test (*test.ply*) que nous tentons de lire (par la méthode **load()**). Nous récupérons ensuite les pointeurs vers les tableaux de vertices et de faces extraits du fichier (méthodes **getVerticesList()** et **getFacesList()**). Nous vérifions tout d'abord que ces tableaux contiennent bien le bon nombre d'éléments, puis nous nous assurons que ces éléments soient bien les mêmes que ceux écrits dans le fichier. Pour ce faire, nous stockons dans deux autres tableaux les vertices et les faces effectivement présents dans le fichier, puis nous comparons le contenu de ces tableaux avec ceux obtenus par les méthodes de **PlyFormatFile**.

Exécution du test

```
./fileloader
Start FileLoader test......FileLoader test complete
```

Ici encore, le paquetage **FileLoader** semble avoir le comportement attendu. On peut en effet avancer que si la classe parvient à récupérer correctement les vertices et les faces présentes dans un fichier donné, elle y parvient également dans n'importe quel autre fichier de ce type.

Le paquetage Structure

Nous vérifions ici la bonne construction de l'arbre séquentiel de points (instance de **SPT**). L'exécution de ce test est un peu particulière. En effet, les différentes étapes d'élaboration de la structure sont réalisées par des méthodes privées à la classe, et donc indisponibles de l'extérieur. Pour réaliser le test, il est donc nécessaire de décommenter dans *SPT.h* les attributs et les méthodes publiques indiqués, et de commenter ces mêmes éléments dans la partie privée, tel qu'indiqué. L'exécutable prend en paramètre le nom d'un fichier *ply* à charger.

Nous commençons par instancier un objet **SPT**, avec comme paramètre le nom du fichier *ply*. Nous construisons ensuite entièrement l'arbre (par les méthodes **fillLeaves()**, **buildTree()**, **calculateGeometricErrorForAllSurfels()** et **setRmaxToAllSurfels()**), puis nous contrôlons celui-ci à l'aide d'un parcours en profondeur : nous vérifions que l'erreur *rMin* et le rayon *radius* de chaque noeud soient bien inférieurs ou égaux à ceux de leur père, et que la

valeur *rMax* de chacun des noeuds soit égale à la valeur *rMin* de son père. Nous transformons ensuite l'arbre en notre liste (**putSurfelsIntoVector()**), puis nous parcourons celle-ci pour s'assurer que pour chacun de ses surfels, *rMin* $\leq$ *rMax*. Enfin, nous trions la liste par ordre décroissant des *rMax* (**sortSPT()**), et nous vérifions que ce tri est correct. Par ces tests, nous nous assurons que l'arbre séquentiel de points respecte bien les caractéristiques décrites dans l'article [DVS03].

Exécution des tests

Nous allons tester ce paquetage avec deux fichiers différents :

- `./structure xyzrgb_manuscript.ply`

  ```
  Start Structure test...
  Number of vertices=2155617
  Number of faces=4305818
  Computing normals... Done.
  Computing splat sizes... Done.
  Building tree...Done.
  Calculate lower bound error...Done
  Calculate upper bound error...Done
  Sequentialization...Done
  ...Structure test complete
  ```

  Sur notre machine de test (dont la configuration est décrite au début du chapitre), lorsque l'objet décrit dans le fichier *ply* contient moins de 7 millions de vertices environ, la construction de la structure se déroule correctement.

-

  ```
  ./structure \textit{lucy.ply}
  Start Structure test...
  Number of vertices=14027872
  Number of faces=28055742
  Computing normals... Done.
  Computing splat sizes... Done.
  Building tree...Done.
  Calculate lower bound error...Done
  Calculate upper bound error...Done
  Sequentialization...Abandon
  ```

  Au delà de 7 millions de vertices, la construction de la structure échoue.

Le paquetage Viewer

L'exécutable de ce test prend également en paramètre le nom d'un fichier *ply*. Il consiste à lancer le visualiseur contenant l'objet 3D décrit par le fichier,

afin de vérifier que les fonctionnalités annoncées (déplacement autour de l'objet, zoom) sont bien présentes.

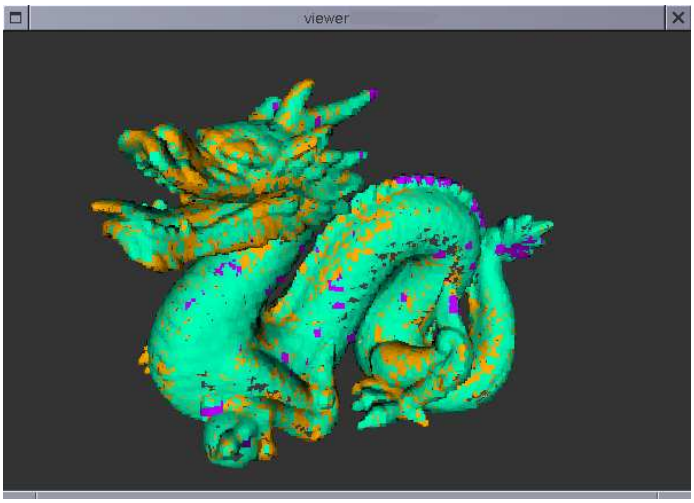


FIG. 9.1 – Visualiseur

Exécution du test

Les fonctionnalités du visualiseur ont bien été implémentées.

9.1.2 HQSSRenderer

Les tests concernant le rendu réaliste sont souvent visuels ; nous pouvons vérifier facilement nos calculs en exécutant le programme et en regardant le résultat directement sur l'objet. De plus, même s'il existe des méthodes qui permettent de récupérer des informations depuis le GPU vers le CPU, il est très difficile d'effectuer des tests rigoureux dans les shaders. Il n'existe par exemple aucune fonction qui permette d'écrire sur la sortie standard. Les tests se résument donc souvent à une coloration spéciale de l'objet suivant le résultat souhaité.

Affichage des surfels

Nous vérifions pour cela qu'aucun trou n'apparaisse pour l'affichage de plusieurs objets. La figure 9.2 montre que le rendu affiche parfois quelques trous de la taille d'un pixel sur l'objet. Cette erreur dépend de l'éloignement de l'objet par rapport à la caméra, mais aussi du dimensionnement du *viewport*.

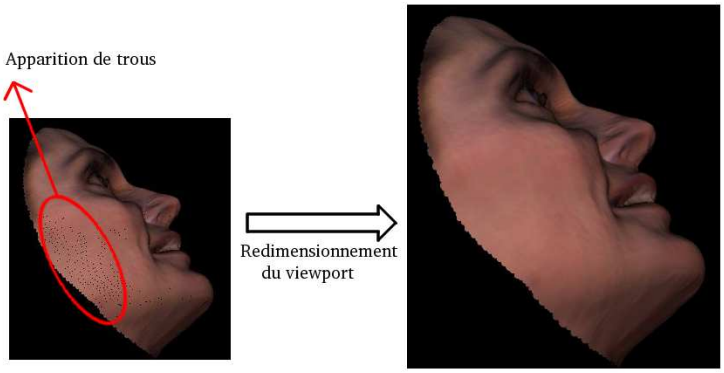


FIG. 9.2 – Quelques trous apparaissent si l'objet est placé à une certaine distance et suivant les dimensions du *viewport*

Il s'agit sûrement d'une erreur d'approximation car la taille des trous ne dépasse, en général, jamais celle d'un pixel. Le fait que le redimensionnement du *viewport* change le résultat nous laisse penser que cette erreur vient sûrement du ratio que nous faisons passer au GPU et qui permet de terminer le calcul de la taille du splat. C'est le seul élément du calcul qui fait apparaître les dimensions du *viewport*. Après avoir essayé de la corriger, nous avons choisi d'arrêter les recherches pour continuer le projet, principalement à cause du manque de temps. De plus, cette erreur n'apparaît que rarement, elle n'a donc pas empêché la continuation du projet.

Deferred Shading

Savoir si notre rendu *multipass* est correctement effectué n'a pas été un problème. Il nous suffit de vérifier lors du dernier passage que chacune des textures obtenues durant les passages précédents sont correctes. Les différents résultats obtenus sont illustrés par la figure 8.9.

La texture obtenue par le premier passage décrit bien les différentes profondeurs de l'objet. Celles qui représentent les couleurs et les textures montrent que le test de profondeur est correct (on ne voit que les pixels de devant), et que les couleurs et normales obtenues sont les bonnes. Enfin, les textures que nous récupérons lors du dernier passage nous permettent d'éclairer correctement la scène en fonction de la position de la lumière. Les tests d'affichage de chacune des textures obtenues montrent que le *Deferred Shading* est correct.

Alpha Blending

Ici encore, les tests visuels se sont révélés très importants. Notre objectif était de faire en sorte que la transition entre les surfels soit la plus fluide possible, de sorte que l'utilisateur ne voit pas les délimitations. La figure 9.3 illustre notre

utilisation du *blending* avec les équations présentées lors de la description du fonctionnement du *Deferred Shading*.

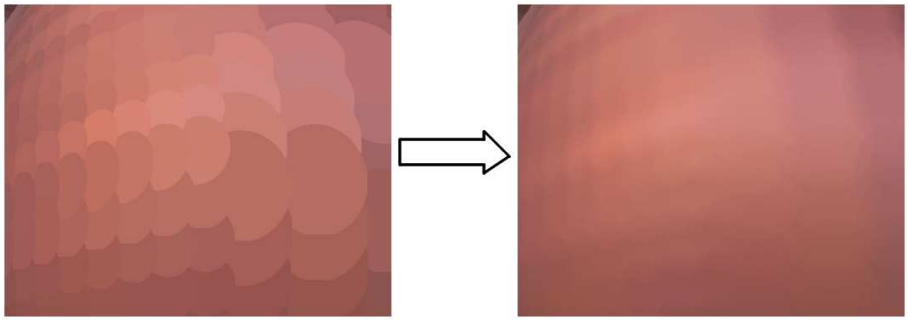


FIG. 9.3 – Représentation de quelques surfels avec et sans *Alpha Blending*

Nos résultats sont donc respectables par rapport à nos attentes.

9.2 Tests d'intégration

9.2.1 LargeObjectsViewer

Pour pouvoir tester l'intégration dans cette partie de projet, il faudrait au préalable écrire des classes s'appuyant sur les interfaces **FileLoader** et **SurfelsStructureInterface**.

FileLoader

Nous avons créé et testé une classe **TestFormatFile** qui implémente cette interface. Celle-ci renvoie toujours la même liste de vertices et de faces quelque soit le fichier passé en entrée dans la fonction **load()**.

Pour pouvoir l'utiliser, il faut modifier la classe de l'interface graphique pour qu'elle reconnaisse les fichiers *.test* comme étant affectés à cette classe (il aurait fallu un moyen de sélectionner le chargeur de fichier désiré par l'intermédiaire de l'interface par exemple).

SurfelsStructureInterface

Dans le but de tester la modularité de cette interface, nous avons créé une classe **TestSurfelStructure** qui l'implémente. Celle-ci ne dessine qu'une spirale.

De même que pour l'interface précédente, il aurait été judicieux de pouvoir choisir le type de structure désiré dans l'interface, en mettant en place un système de plugins par exemple.

9.2.2 HQSSRenderer

Le seul test d'intégration que nous ayons eu à faire est celui de la mise en place du plugin de rendu de *PointShop3D*. Il a fallu pour cela implémenter simplement les fonctions de l'interface **RendererInterface**. Il a d'ailleurs fallu faire ce travail dès le début (ce fut la première étape de notre planning), car il était nécessaire que nous puissions faire nos tests de rendu durant tout le projet.

Nos tests d'intégration sont donc validés par le fait que *PointShop3D* reconnaisse notre rendu et en affiche les résultats.

9.3 Tests de validation

9.3.1 LargeObjectsViewer

Le modèle de référence devait être *lucy.ply*. Hélas, nous ne pouvons charger cet objet dans notre structure par manque de mémoire. Pour y parvenir, il aurait fallu quantifier les données du modèle afin de réduire l'espace mémoire utilisé, mais nous n'avons pas eu le temps d'implémenter cette quantification et nous avons préféré nous attarder sur d'autres points plus importants. Notre nouveau modèle de référence est donc *happy_vrip.ply* : il est composé de 543 652 vertices et de 1 087 716 faces.

Construction de l'arbre séquentiel de points

La construction de l'arbre séquentiel de points correspondant à ce modèle est réalisée dans un temps compris entre 5 et 6 secondes. Il faut noter que la lecture du fichier *ply* ne fait pas partie de la construction de la structure et n'est donc pas pris en compte dans ce calcul. La durée de construction étant en partie dépendante du nombre de vertices de l'objet, pour tous les autres modèles essayés contenant moins de vertices que *happy_vrip.ply*, cette durée est moindre.

Affichage de l'objet

Nous nous sommes aperçus que mesurer le nombre de FPS n'est pas pertinent. En effet, il faut tenir compte du nombre de détails de l'objet et de la distance entre la camera et celui-ci. Ainsi, quelque soit l'objet, lorsque celui-ci est relativement loin de l'observateur, le nombre de FPS est maximal (a peu près 125), et ce nombre réduit lorsqu'on s'approche (avec *happy_vrip.ply*, lorsque la caméra est "collée" à l'objet, le nombre de FPS est environ de 20).

Commentaires des résultats

Il est difficile de comparer les résultats obtenus avec ceux annoncés, les modèles utilisés étant différents. Nous obtenons des performances bien supérieures à celles qui avaient été prévues ultérieurement, ce qui peut laisser suggérer que nous aurions atteint les résultats escomptés avec *lucy.ply*. Néammoins, nous ne

pouvons l'affirmer car nous n'avons aucune idée du temps supplémentaire requis pour quantifier et "déquantifier" les données.

Remarque : les tests de construction de la structure ont été réalisés lors des tests unitaires.

Comparaison avec les logiciels existants

Nous pourrions comparer nos performances avec les logiciels *QSplat* et *PointShop3D*, mais :

- *QSplat* n'utilisant pas les capacités des cartes graphiques modernes, nous ne pouvons pas les mettre en concurrence puisque les résultats sont biaisés ;
- *PointShop3D* n'utilisant pas le LOD, nous ne pouvons pas les comparer (de plus, nous ne connaissons pas précisément la structure utilisée).

9.3.2 HQSSRenderer

Performances

Fichier	nb Splats	FPS	nb Splats/seconde
face.sfl	40 880	80	3 270 400
dinosaur.sfl	56 194	80	4 495 520
santa.sfl	75 781	65	4 925 765
male.sfl	148 138	56	8 295 728
dragon.sfl	1 236 969	7	8 658 783

FIG. 9.4 – test effectué dans une fenêtre de 800 x 580 pixels

Ces tests montrent que nous atteignons bien les résultats voulus. Nous souhaitions obtenir une performance d'environs 3 000 000 de splats par seconde. Nous dépassons donc largement nos espérances, car les chiffres que nous obtenons sont pratiquement trois fois supérieurs à nos attentes.

Ce tableau montre toutefois certaines incohérences : on remarque que le nombre de splats par seconde augmente avec la taille de l'objet. Ceci s'explique par le fait que les résultats varient considérablement suivant la position et l'éloignement de l'objet. Par exemple, nous avons constaté que le nombre de frames par seconde de l'objet *dinausor.sfl* pouvait varier entre 50 et 110. Nous affichons donc ici seulement des moyennes. Plusieurs facteurs entrent en jeux pour la rapidité du rendu, notamment la taille des carrés à rastériser et le nombre de surfels qui passent le test de profondeur. Comme nous l'avons expliqué précédemment, la complexité exacte n'est pas calculable et ne dépend pas uniquement de la taille de l'objet, cela explique l'irrégularité des résultats obtenus.

Comparaison avec d'autres plugins de rendu

Nous avons voulu comparer nos résultats avec les autres plugins de rendu de *Pointshop3D*. Ils ont été développé par des personnes dont certaines, comme Matthias Zwicker, ont participées à la rédaction des documents sur lesquels nous travaillons. Voici les images que nous obtenons avec les plugins téléchargeables sur le site de *Pointshop3D* :

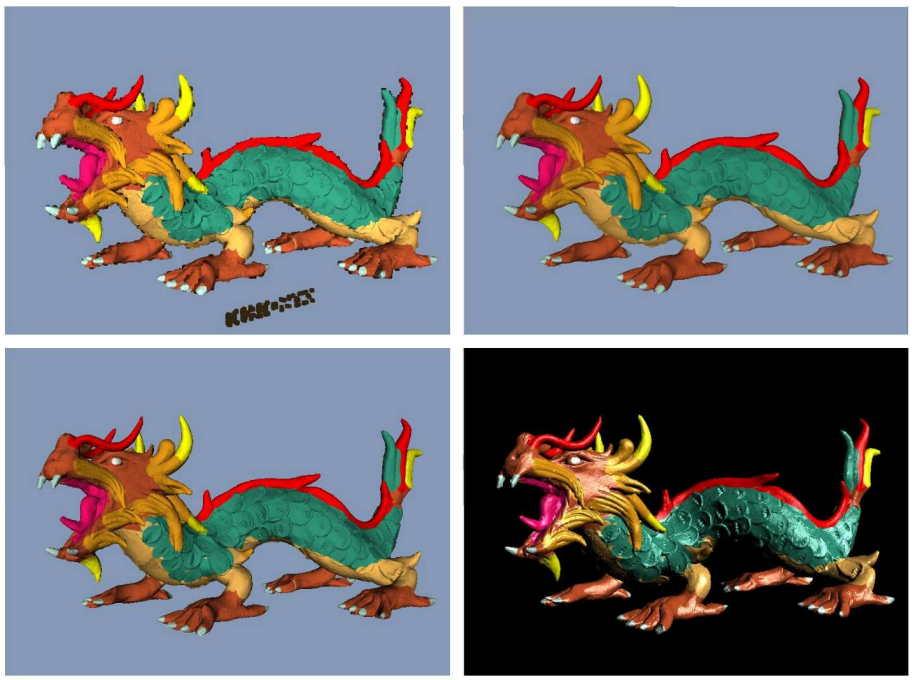


FIG. 9.5 – En haut à gauche : **OpenglRenderer**, en haut à droite : **SurfelRenderer**, en bas à gauche : **EWARenderer**, en bas à droite : **HQSSRenderer**

Les plugins **OpenglRenderer** et **SurfelRenderer** sont ceux fournis par défaut dans le logiciel. Le premier effectue un rendu simple, en utilisant principalement les fonctions d'OpenGL pour les calculs. Il affiche des surfels sans prendre en compte l'*Alpha Blending*, et en les éclairant avec une lumière diffuse. La taille des splats n'est pas calculée précisément car lorsqu'elle devient trop grande (que l'on rapproche l'objet de l'écran), des trous apparaissent dans la scène. De plus, des artefacts apparaissent car les surfels sont souvent plus gros qu'ils ne devraient l'être. Le plugin **SurfelRenderer** est beaucoup plus élaboré car il ne contient pas d'artefacts, se préoccupe du recouvrement des splats, et prend en compte l'intensité spéculaire lors du calcul de la luminosité. Enfin, le dernier plugin **EWARenderer** effectue un rendu rapide et de qualité, mais sans prendre en compte la lumière spéculaire.

Voici maintenant les performances que nous obtenons avec ces même plugins :

Fichier	nb Splats	Surfel	Opengl	EWA	HQSS
face.sfl	40 880	3	5	90	80
dinosaur.sfl	56 194	3	4	76	80
santa.sfl	75 781	2	3	56	65
male.sfl	148 138	2	2	26	56
dragon.sfl	1 236 969	<1	<1	3.5	7

FIG. 9.6 – test effectué dans une fenêtre de 800 x 580 pixels, les résultats représentent des nombres de frames par seconde

La comparaison des performances fait apparaître plusieurs résultats significatifs : les plugins par défaut sont très lents, notamment **SurfelRenderer** qui prend énormément de temps pour les calculs. Ils n'utilisent sûrement pas de méthodes de communication optimisées pour le GPU telles que l'utilisation des VBOs. Quand au plugin **EWARenderer**, développé par Michael Waschbuesch qui est enseignant-chercheur à l'université de Zurich, ses résultats sont équivalents pour les objets de petites tailles. Nos performances sont néanmoins meilleures lorsque les objets possèdent un grand nombre de points : nous atteignons un rendu de 8 658 783 de splats par seconde (lumière spéculaire comprise) alors que le rendu *EWA* se limite à environ 4 350 000.

Notre rendu est donc assez performant comparé aux autres plugins. Toutefois, nous aurions aimé faire les tests de comparaison avec un autre plugin nommé **HGLSplatRenderer** et développé par Matthias Zwicker. Nous n'avons pas pu le faire, celui-ci n'étant pas compilable sous Linux.

9.4 Tests de couverture

9.4.1 LargeObjectsViewer

Gestion de la mémoire

Pour pouvoir détecter les erreurs et les fuites mémoires, nous avons utilisé le programme *Valgrind*.

Celui-ci détecte efficacement les erreurs de mémoire comme un test sur un objet non initialisé par exemple ; ou encore les octets alloués mais non désalloués par la suite.

Après plusieurs tests, *Valgrind* lancé avec notre programme en paramètre n'est pas exploitable. En effet, cela se termine par une erreur de segmentation peu après le chargement de l'objet ; erreur qui ne survient pas si le programme est exécuté normalement, avec le même fichier objet. Cela doit être dû à quelque fonction présente dans l'interface graphique, puisque le même test sur le widget **Viewer** s'exécute normalement. Nous avons donc regardé la trace mémoire avec le test du widget **Viewer** ; mais celui-ci étant peu clair, nous avons aussi testé la construction de la structure, là où les fuites mémoires ont plus de risques d'apparaître.

Commande de lancement de *Valgrind* :

```
valgrind --leak-check=yes --show-reachable=yes ./executable <chemin\_de\_l'objet>
```

Construction de la structure (Voir Annexe B)

Le résultat révèle une fuite mémoire quasi constante (quelque soit la taille de l'objet). D'après les fonctions affichées, toutes les fuites proviennent des fonctions de la bibliothèque **ply**. La somme totale des-dites fuites est d'à peu près 500 Ko.

Exemples de fonction responsable des fuites mémoires :

```
 12 bytes in 1 blocks are indirectly lost in loss record 1 of 6
==5810==    at 0x401CCDB: realloc
==5810==    by 0x804CB40: add_property
==5810==    by 0x804CDDF: ply_read
==5810==    by 0x804CEE6: ply_open_for_reading
==5810==    by 0x804D0F1: PlyFormatFile::load(char*)
==5810==    by 0x804F9D1: SPT::fillLeaves()
```

....

Diagnostic avec le test de la structure avec le fichier *bunny.ply* (35 947 vertices) :

```
==5808== LEAK SUMMARY:
==5808==    definitely lost: 44 bytes in 5 blocks.
==5808==    indirectly lost: 600 bytes in 35 blocks.
==5808==      possibly lost: 0 bytes in 0 blocks.
==5808==    still reachable: 16 bytes in 1 blocks.
==5808==         suppressed: 0 bytes in 0 blocks.
```

Diagnostic avec le test de la structure avec le fichier *happy.ply* (543 652 vertices) :

```
==5810== LEAK SUMMARY:
==5810==    definitely lost: 36 bytes in 5 blocks.
==5810==    indirectly lost: 430 bytes in 27 blocks.
==5810==      possibly lost: 0 bytes in 0 blocks.
==5810==    still reachable: 16 bytes in 1 blocks.
==5810==         suppressed: 0 bytes in 0 blocks.
```

Etrangement, on notera que la fuite est moins importante.

La construction de la structure (c'est-à-dire le chargement, la construction de l'arbre, le calcul des erreurs et la séquentialisation) ne semble pas provoquer de fuites de mémoire.

On remarquera qu'aucune erreur sur la gestion de la mémoire n'est apparue :

```
==5810== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 57 from 1)
```

Widget *Viewer* (Voir Annexe ??)

Le widget a été lancé avec l'objet *happy.ply*. Nous avons vu apparaître de nombreuses erreurs dans l'analyse : certaines proviennent de la bibliothèque OpenGL des pilotes Nvidia (version des pilotes : 8178),

```
==6478== Conditional jump or move depends on uninitialised value(s)
==6478==    at 0x52FDA11: (within /usr/lib/opengl/nvidia/lib/libGLcore.so.1.0.8178)
...
==6478== Syscall param ioctl(generic) points to uninitialised byte(s)
==6478==    at 0x4EED319: ioctl (in /lib/libc-2.3.6.so)
==6478==    by 0x52FD932: (within /usr/lib/opengl/nvidia/lib/libGLcore.so.1.0.8178)
==6478==    Address 0xBEB56C0C is on thread 1's stack
```

d'autres proviennent, semble-t-il, de la bibliothèque *QT* (ici, 4.1.1) ou de la bibliothèque de *XFree86/Xorg* (ici, Xorg 7.0.0) :

```
==6478== Syscall param write(buf) points to uninitialised byte(s)
==6478==    at 0x4D3A3F3: __write_nocancel
==6478==    by 0x4A0539E: _X11TransWrite
==6478==    by 0x4A097E3: (within /usr/lib/libX11.so.6.2.0)
==6478==    by 0x4A09874: _XReply
==6478==    by 0x49EF930: XInternAtom
==6478==    by 0x4A046DD: XSetWMProperties
==6478==    by 0x44A177B: QWidgetPrivate::create_sys(...)
==6478==    by 0x44710A7: QWidget::create(..)
==6478==    by 0x44712C5: QWidgetPrivate::init(...)
==6478==    by 0x44718B3: QWidget::QWidget(...)
==6478==    by 0x42FE401: QGLWidget::QGLWidget(...)
==6478==    by 0x4069BA1: QGLViewer::QGLViewer(...)
...
==6478==  Address 0x580ED98 is 320 bytes inside a block of size 16,384 alloc'd
==6478==    at 0x401CBE7: calloc
==6478==    by 0x49F4F80: XOpenDisplay
==6478==    by 0x44869E4: qt_init(...)
==6478==    by 0x443484B: QApplicationPrivate::construct()
==6478==    by 0x4434C23: QApplication::QApplication(...)
==6478==    by 0x805543D: main
```

Aucune fonction présente dans notre application n'apparaît dans le listing.
Résumé des erreurs :

```
ERROR SUMMARY: 2085 errors from 148 contexts (suppressed: 111 from 2)
```

De même, les fuites mémoires proviennent des bibliothèques *QT* et *XFree86/Xorg*

```
==6478== 2,048 bytes in 1 blocks are still reachable in loss record 70 of 81
==6478==    at 0x401B511: malloc (in .../valgrind/x86-linux/vgpreload_memcheck.so)
```

```
==6478==    by 0x4A19582: _XlcCreateLocaleDataBase
==6478==    by 0x4A1DABE: _XlcCreateLC
==6478==    by 0x4A3F379: _XlcUtf8Loader
==6478==    by 0x4A241FD: _XOpenLC
==6478==    by 0x4A244ED: _XrmInitParseInfo
==6478==    by 0x4A0D5BB: XrmGetStringDatabase
==6478==    by 0x49E92BC: XGetDefault
...
==6478== 184,312 bytes in 1,125 blocks are still reachable in loss record 81 of 81
==6478==    at 0x401B511: malloc (.../valgrind/x86-linux/vgpreload_memcheck.so)
==6478==    by 0x4996CAF: (within /usr/lib/libfontconfig.so.1.0.4)
==6478==    by 0x49981E9: FcDirCacheReadDir
==6478==    by 0x499DDB7: FcDirScanConfig
==6478==    by 0x4998B15: FcConfigBuildFonts
==6478==    by 0x49A158B: FcInitLoadConfigAndFonts
==6478==    by 0x49A17F4: FcInit
==6478==    by 0x448793E: qt_init(...)
==6478==    by 0x443484B: QApplicationPrivate::construct()
==6478==    by 0x4434C23: QApplication::QApplication(...)
==6478==    by 0x805543D: main
...
```

On retrouve aussi les mêmes fonctions de la bibliothèque *ply*, mais toujours aucune fuite dûe à notre code.

```
==6478== LEAK SUMMARY:
==6478==    definitely lost: 180 bytes in 9 blocks.
==6478==    indirectly lost: 654 bytes in 41 blocks.
==6478==      possibly lost: 0 bytes in 0 blocks.
==6478==    still reachable: 417,558 bytes in 5,210 blocks.
==6478==         suppressed: 0 bytes in 0 blocks.
```

Profil

Le profilage de l'application a été réalisé avec *gprof*, qui induit lors de la compilation l'argument «-pg» supplémentaire.

L'intégralité du rapport sur le profil du programme (c'est-à-dire le «**main**» avec l'interface graphique, contrairement lors de la gestion de mémoire) se trouve en annexe C.

Voici une sélection du profil, à analyser :

```
  %   cumulative   self
 time   seconds   seconds    calls  name
 53.85      0.07     0.07 44415547  Surfel::~Surfel()
 23.08      0.10     0.03 44413964  Surfel::Surfel(Surfel const&)
 15.38      0.12     0.02     1583  __gnu_cxx::__normal_iterator...
  7.69      0.13     0.01   813399  Surfel::Surfel()
```

```
0.00      0.13     0.00  5981820  get_ascii_item
0.00      0.13     0.00  5981820  store_item
0.00      0.13     0.00  2719114  my_alloc
0.00      0.13     0.00  1631378  get_words
0.00      0.13     0.00  1631368  ascii_get_element
0.00      0.13     0.00  1631368  ply_get_element
0.00      0.13     0.00   811816  Surfel::~Surfel()
...
0.00      0.13     0.00   536328 SPT::calculateTangentialError(...)
0.00      0.13     0.00   268164 SPT::calculateGeometricError(...)
0.00      0.13     0.00   191185 SPT::combineLeaves(...)
0.00      0.13     0.00   191184 SPT::partition(...)
0.00      0.13     0.00    92395 SPT::combineNodes(...)
0.00      0.13     0.00    90697 SPT::combineNodes(...)
0.00      0.13     0.00    85072 SPT::combineNodes(...)
...
0.00      0.13     0.00     1583 SPT::drawSelectedSurfelsList(...)
...
```

Les élements importants sont :

- les éléments les plus «consommateurs» de temps CPU sont les opérations faites lors de l'étape du *preprocessing* et le temps passé respecte le temps fixé (voir les tests de validation) ;
- la fonction sensée être la plus rapide (**SPT : :drawSelectedSurfelsList** utilisant une fonction de la STL d'iterateur dans un vector) utilise 2 millisecondes de temps CPU pour 1583 appels à *draw()* (30 secondes d'utilisation : rotation, translation,...), ce qui nous paraît rapide ;
- la différence de la somme des constructeurs avec la somme des destructeurs est nulle, ce qui nous conforte dans la bonne gestion de la mémoire.

9.4.2 HQSSRenderer

Gestion de la mémoire

Beaucoup d'erreurs sont semblables à celle obtenues pour la partie **LargeObjectsViewer**, notamment celles concernant *QT* (ici, version 3.3.4) et *XFree86/Xorg*. Nous n'avons pas activé l'option d'affichage d'erreur, car il y'en a beaucoup et proviennnent toutes de *PointShop3D* ou des erreurs précedentes.

Au niveau des fuites mémoires nous avons comparés les tests entre *PointShop3D* seul, le rendu de EWA, et le notre. On observe que *PointShop3D* provoque une grande fuite mémoire lors de chacun des tests.

```
==8886== 4675616 bytes in 65 blocks are still reachable in loss record 624 of 624
==8886==    operator new[](unsigned)pas
==8886==    ChannelComponent::ChannelComponent(QSize)
==8886==    BrushChannel::setComponents(QSize, int, float, QImage const*)
==8886==    BrushChannel::load(_IO_FILE*, QString, int, QSize)
```

```
==8886==    Brush::load(QString, QString)
==8886==    BrushDirWalk::applyFileMethod()
==8886==    DirWalk::execute(UserDataInterface*)
==8886==    DirWalk::execute(UserDataInterface*)
==8886==    BrushChooserTool::updateBrushes()
==8886==    BrushChooserTool::addButtonToToolBar(QToolBar*)
==8886==    ResourceToolBar::addDefaultTools()
==8886==    PointShopWindow::initToolBars()
```

Il devient alors très difficile de déterminer si notre plugin possède ou non des fuites mémoire, sachant que *PointShop3D* lui-même en possède beaucoup. Le fait de comparer des plugins entre eux n'entraîne pas non plus de constatations claires puisqu'ils peuvent avoir eux-même des fuites. Nous ne pourrons donner qu'une idée, mais sans pour autant en être persuadés.

Voici le résultat obtenu avec *PointShop3D* seul :

```
 LEAK SUMMARY:
==8579==    definitely lost: 16715 bytes in 10 blocks.
==8579==    indirectly lost: 120 bytes in 10 blocks.
==8579==      possibly lost: 2864 bytes in 3 blocks.
==8579==    still reachable: 7440472 bytes in 16691 blocks.
==8579==         suppressed: 0 bytes in 0 blocks.
```

Celui de EWA :

```
LEAK SUMMARY:
==9701==    definitely lost: 4731 bytes in 70 blocks.
==9701==    indirectly lost: 187811 bytes in 3330 blocks.
==9701==      possibly lost: 19248 bytes in 4 blocks.
==9701==    still reachable: 16421649 bytes in 19384 blocks.
==9701==         suppressed: 0 bytes in 0 blocks.
```

Et enfin le nôtre :

```
 LEAK SUMMARY:
==8886==    definitely lost: 16703 bytes in 7 blocks.
==8886==    indirectly lost: 120 bytes in 10 blocks.
==8886==      possibly lost: 2864 bytes in 3 blocks.
==8886==    still reachable: 7447695 bytes in 16820 blocks.
==8886==         suppressed: 0 bytes in 0 blocks.
```

On remarque que l'on possède moins de pertes mémoire que le plugin **EWA-Renderer** ou même que *Pointshop3D* seul. Cela laisse penser que notre plugin n'a pas de pertes mémoire (bien que nous ne puissions le prouver). Nous ne trouvons d'ailleurs pas de fonctions appartenant à nos classes dans les traces de *Valgrind*.

Concernant les erreurs de gestion de mémoire, *Pointshop3D*, ainsi que chacun des plugins en contiennent énormément. Le fait de passer le tableau de surfels à OpenGL provoque des erreurs de sauts conditionnels :

```
==9819== Conditional jump or move depends on uninitialised value(s)
==9819==    (within /usr/lib/opengl/nvidia/lib/libGLcore.so.1.0.8178)
==9819==    HQSSRendererWidget::initializeGL()
(in /home/romain/pdp/trunk/Pointshop3D2.0Src/bin/linux/
release/Renderers/preview/libHQSSRenderer.so)
==9819==    QGLWidget::glInit()
==9819==    QGLWidget::resizeEvent(QResizeEvent*)
==9819==    QWidget::event(QEvent*)
==9819==    QApplication::internalNotify(QObject*, QEvent*)
==9819==    QApplication::notify(QObject*, QEvent*)
==9819==    QApplication::sendPostedEvents(QObject*, int)
==9819==    QWidget::show()
==9819==    QWidgetStack::raiseWidget(QWidget*)
==9819==    PointShopWindow::handlePreviewRendererEnabled(bool)
```

Etant donné que nous avons repris le code d'autres plugins pour l'implémentation de notre programme, nous n'avons pas géré les nombreuses autres erreurs de ce type. Pour chacun des tests effectués, le résultat obtenu est :

```
More than 30000 total errors detected.  I'm not reporting any more.
==9819== Final error counts will be inaccurate.  Go fix your program!
==9819== Rerun with --error-limit=no to disable this cutoff.  Note
==9819== that errors may occur in your program without prior warning from
==9819== Valgrind, because errors are no longer being displayed.
==9819==
==9819==
==9819== ERROR SUMMARY: 30000 errors from 136 contexts (suppressed: 140 from 2)
==9819== malloc/free: in use at exit: 7465731 bytes in 16819 blocks.
==9819== malloc/free: 1084676 allocs, 1067857 frees, 301040662 bytes allocated.
==9819== For counts of detected errors, rerun with: -v
==9819== searching for pointers to 16819 not-freed blocks.
==9819== checked 8771632 bytes.
```

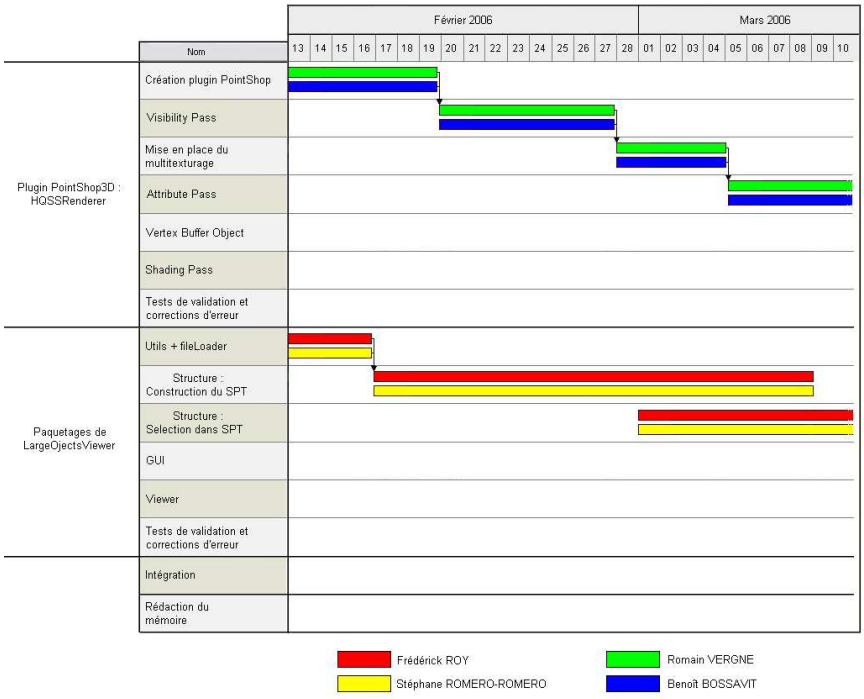
Profil

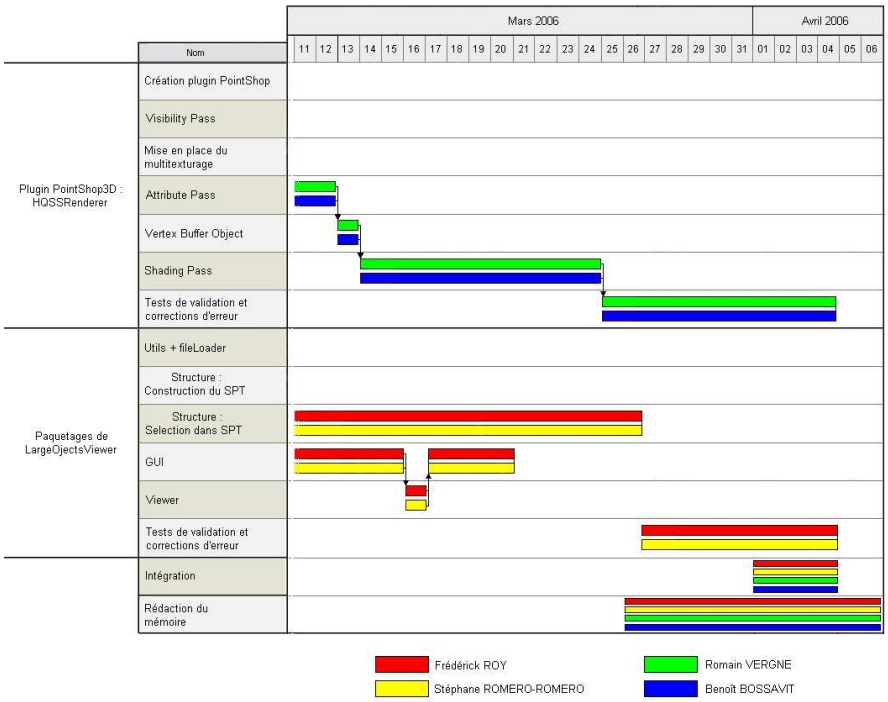
Nous n'avons pas pu effectuer ce test car il aurait fallu que nous compilions entièrement *PointShop3D* avec l'option spécifique. Un trop grand nombre de fonctions est appelé lors du lancement du logiciel. Néanmoins, nous voyons facilement dans notre code que notre fonction de rendu est celle qui sera appelée à chaque fois que la fenêtre sera repainte. Nous avons donc fait attention à ce qu'il n'y ait pas trop de méthodes à l'intérieur de celle-ci. De plus, nous faisons attention à ne jamais appeler de fonctions pour chaque surfel de l'objet. Le seul moment où nous avons eu besoin de le faire, pour la modification de la couleur des pixels sélectionnés, nous avons utilisé une fonction *inline* qui permet d'insérer le code statiquement lors de la compilation.

Chapitre 10

Bilan

10.1 Planning effectif





10.2 Analyse du planning et problèmes rencontrés

Le planning effectif est en effet assez éloigné de celui prévu à l'origine pour les deux projets. Nous avons rencontré de nombreuses difficultés au cours de l'implémentation.

10.2.1 LargeObjectsViewer

Le planning concernant l'implémentation du paquetage **FileLoader** a été relativement respecté ; la difficulté résidant dans le moyen peu banal de récupérer les données (i.e vertex et face). Nous avions d'abord décidé de faire une structure contenant trois flottants, mais il nous est rapidement apparu plus pratique de faire une structure de trois éléments quelconques en utilisant les *templates*. Pendant les tests unitaires, nous nous sommes aperçus que la bibliothèque de référence (http ://astronomy.swin.edu.au/ pbourke/dataformats/ply/) n'arrivait pas à lire correctement les fichiers *ply* binaires. Nous nous sommes procurés alors une version améliorée corrigeant le problème.

Le paquetage **Structure** a été le plus long et le plus sujet à modifications. Ainsi, nous voulions dès le départ construire les surfels à partir des faces, ce qui est le plus logique pour obtenir les normales. Mais nous avons eu des difficultés

à trouver l'algorithme, et après implémentation, le chargement fut assez long. De plus, nous nous sommes aperçus, qu'en moyenne, un objet contient deux fois plus de faces que de points. Nous avons alors repris le code de *QSplat*, que nous avons transformé pour pouvoir utiliser nos structures (on a diminué ainsi la mémoire occupée et augmenté la rapidité d'exécution). Le calcul des erreurs a été problématique lui aussi, en particulier celle de l'erreur tangentielle. La documentation étant succinte, nous avons dû réfléchir plus longuement. La séquentialisation et la présélection nous a pris 20 jours.

La sélection, et plus généralement la mise en place des shaders, a été le plus difficile à mettre en oeuvre dû au manque de documentation sur les nouvelles techniques comme les *Vertex Buffer Objets* et *GLSL*. L'équipe s'occupant du plugin *PointShop3D* nous a été d'une aide précieuse pour le rendu et nous a permis de vérifier la validité de notre sélection. C'est durant cette période que nous avons essayé de trouver la raison pour laquelle l'erreur est beaucoup plus faible que la distance de vue (ratio de 150). Après une dizaine de jours d'essais infructueux, nous avons envoyé un courrier électronique à un des auteurs de l'article [DVS03] et l'avons interrogé sur ce problème entre autres. A ce jour, nous n'avons malheureusement reçu aucune réponse. La durée totale de l'implémentation de cette partie est 24 jours, en alternance avec la partie précédente et le **GUI**.

La classe **Viewer**, héritant de **QGLViewer**, n'a pas posé spécialement de problème et a été rapidement implémentée (1 jour).

Enfin, pour l'interface graphique, nous avions commencé à la developper avec l'aide de la bibliothèque graphique *QT3* de *TrollTech*. Mais après quelques jours et une interface rudimentaire, nous avons réadapté le code avec la bibliothèque *QT4*, plus portable (spécialement pour Windows). L'écriture et l'utilisation d'un thread pour la construction de la structure fut rapidement mise en oeuvre, après avoir constaté le gel de l'interface en son absence. L'ensemble du paquetage **GUI** nous a pris finalement une dizaine de jours.

10.2.2 HQSSRenderer

Tout d'abord, il a fallu préparer le plugin pour *PointShop3D*. Cette installation a nécessité une semaine, afin de savoir comment fonctionnait le système de plugin et comment ils sont intégrés au sein du logiciel.

Nous avons ensuite consacré une semaine pour implémenter une première ébauche du *Visibility Pass*, le temps de découvrir les possibilités du langage et de comprendre les différents types de repères : espace, vue, projection, écran. La taille du *splat* et sa transformation en ellipse sont en effet calculées à partir de coordonnées sur différents repères.

L'implémentation de l'*Attribute Pass* a nécessité deux semaines. Dans un premier temps, nous avons mis en place la technique de multitexturage afin d'observer le résultat de plusieurs étapes consécutives. Les données sont d'abord mises dans le tampon d'image dans les *shaders*, puis copiées dans une texture. La texture peut ensuite être affichée à l'écran. Ce mécanisme empêche de créer plusieurs textures lors d'une même étape.

Ensuite, la gestion du *blending* nous a apporté quelques complications. Il y avaient beaucoup d'inconnues : comment choisir une fonction gaussienne adéquate, les paramètres d'OpenGL pour le *blending*, mais aussi le tampon de profondeur mal défini qui apportait de mauvais résultats.

Au départ nous utilisions un tableau pour stocker les surfels, mais l'exécution du rendu en été ralentie et nous avons donc mis en place les *Vertex Buffer Objects*, avec l'aide indispensable de l'équipe chargée de la construction de la structure. Cette implémentation a augmenté considérablement la rapidité du rendu.

La dernière étape, *Shading Pass*, a posé problème à cause d'une différence de référentiel. Il faut à partir du tampon de profondeur calculer la position 3D et faire la différence avec la position de la source lumineuse. La transformation sur un même repère de ces deux composantes est assez délicate. Nous avons aussi, bien plus tard, remarqué un mauvais passage des normales dans leur texture ce qui apportait quelques résultats surprenants.

10.3 Les extensions possibles

Bien que les résultats obtenus paraissent satisfaisants, certaines améliorations restent à effectuer.

10.3.1 LargeObjectsViewer

La première amélioration que nous pourrions apporter est le chargement de modèles contenant des couleurs. Il suffirait pour cela de rajouter un attribut supplémentaire à la classe **Surfel** stockant cette couleur. Ce chargement augmenterait bien évidemmment la taille de la structure, mais la raison pour laquelle nous n'avons pas implémenté cette fonctionnalité est que tous les objets 3D couleur au format *ply* que nous avons trouvé contenaient trop de vertices pour être gérés par notre structure.

Nous sommes limités dans la taille des objets 3D que l'on charge en priorité par manque de mémoire vidéo. En effet, nous stockons actuellement toute la structure dans celle-ci, par soucis d'efficacité du rendu (voir l'explication de l'algorithme de **LargeObjectsViewer**). Il faudrait donc pour économiser la mémoire changer notre technique de mise en mémoire vidéo, ce qui aurait sans aucun doute des répercussions néfastes sur les performances.

Notre seconde limite est le manque de mémoire vive. Pour remédier à cela, il faudrait réduire la taille d'une instance de la classe **Surfel**, qui est actuellement de 56 octets, c'est-à-dire mettre en place un système de quantification tel que celui décrit dans [BWK02].

10.3.2 HQSSRenderer

Antialiasing

Nous avons commmencé à implémenter la méthode d'antialiasing proposée dans [BHZK05]. Elle consiste à une approximation de EWA donnant des résultats qui semblent satisfaisants.

Lors de l'*Attribute Pass*, on effectue un test afin de savoir quels pixels sont à afficher. Pour mettre en place la technique d'antialiasing, il faut prendre le minimum entre cette valeur de comparaison u^2+v^2 et $(d(x,y)/\sqrt{2})^2$, avec $d(x,y)$ la distance entre le centre du splat projeté et la position du pixel courant. Ce minima sert pour le nouveau test de sélection des pixels, ainsi que pour la pondération des couleurs et normales.

Nous avons rencontré un problème lors de la méthode d'antialiasing, pour le calcul du «rayon 2D» du splat. En effet, pour cela il faut positionner les deux coordonnées dans le même repère. Pour obtenir des résultats cohérents il faut transformer les coordonnées du splat projeté aux «coordonnées écran». Le plan de projection homogénéisé est dans un repère compris entre -1 et 1, alors que le repère de l'écran est compris entre 0 et la taille en largeur du *viewport* pour l'axe des abscisses et entre 0 et la hauteur pour l'axe des ordonnées. Il suffit de multiplier les coordonnées du splat projeté par la taille du *viewport* pour effectuer une mise à l'échelle et de faire une translation du centre de ce repère. Cette réalisation a pu être effectuée grâce à une prise de contact avec Mme Sylvie Alayrangues, professeur au LaBRI, qui abordait le sujet dans l'un de ses cours. Malgré ces calculs, les résultats obtenus semblent cohérents mais ne sont pas exacts. Lorsque l'objet et proche, le «rayon 3D» est supérieur au «rayon 2D», ce qui signifie qu'il n'y a pas d'aliasing, et dès qu'on s'éloigne on remarque que la pondération est saturée (la couleur est blanche) en commençant par les points les plus éloignés. Voici un aperçu des tests de comparaisons des «rayons» à gauche (en vert lorsque le «rayon 2D» est inférieur au «rayon 3D») et le résultat de l'affichage des couleurs à droite.

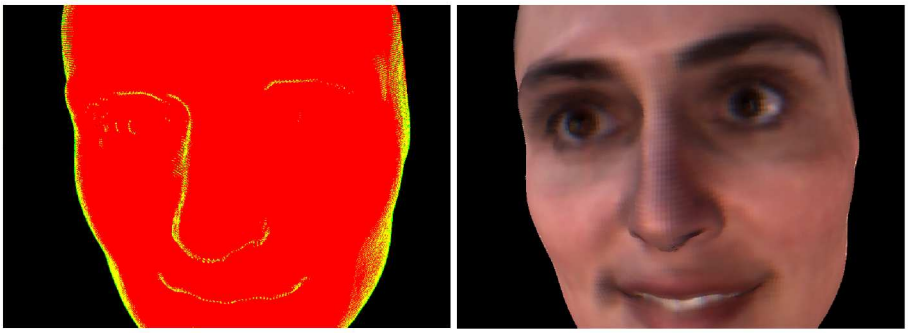


FIG. 10.1 – résultat des tests d'antialiasing

Taille des splats

Comme nous l'avons montré dans l'explication des algorithmes pour **HQSS-Renderer**, le calcul de la taille des splats montre quelques défaillances. On observe des «trous» à certains endroits. Cela semble venir d'un calcul quelque peu erroné en rapport avec la taille du *viewport*. Nous n'avons pas pu résoudre ce problème, ceci pouvant être une correction à apporter ultérieurement.

Mise en place des *Frame Buffer Objects*

Dans le programme actuel, le tableau de surfels est parcouru deux fois lors du second passage dans le *Deferred Shading* : pour l'accumulation des couleurs, et celle des normales. Les FBOs permettent d'éviter le parcours de trop grâce à une technique appelée le *multiple render target*. Au lieu d'écrire directement dans le *Frame Buffer* au niveau du *Fragment Shader*, il s'agit ici d'écrire simultanément dans deux textures différentes, chacune de la taille du *viewport*, puis de les récupérer dans le programme principal à l'aide de fonctions spécifiques des FBOs.

Nous avons essayé d'implémenter cette méthode, mais nous n'avons pas pu terminer à cause d'une erreur d'utilisation des FBOs. Nous obtenons bien nos deux textures de couleurs et de normales à la fin du second passage. Néanmoins, il semble que le test de profondeur utile à la sélection des surfels ne se fasse pas correctement. En effet, tout se passe comme si le test de profondeur ne se faisait plus lors de l'accumulation des propriétés matérielles. Nous n'avons malheuresement pas eu le temps de terminer, même si nous touchons presque au but. Pour résoudre ce problème, nous pourrions essayer de faire passer la texture de profondeur au second passage, et de faire le test de profondeur nous-même. Mais peut-être existe-t-il une solution plus adéquate. Le fait que les FBOs soient une nouvelle extension d'OpenGL et qu'il n'existe que très peu de documentation à leur sujet nous a empêché de les utiliser facilement.

Malgré le fait que nous n'ayions pas réussi à mettre complètement les FBOs en place, les résultats que nous obtenons nous laissent penser que les performances que nous atteignons avec leur utilisation ne changeront pas (ou très peu).

Voici les tests effectués :

Fichier	nb Splats	HQSSRenderer	HQSSRenderer + FBOs
face.sfl	40 880	80	100
dinosaur.sfl	56 194	80	115
santa.sfl	75 781	65	90
male.sfl	148 138	56	64
dragon.sfl	1 236 969	7	10

FIG. 10.2 – test effectué dans une fenêtre de 800 x 580 pixels, les résultats représentent le nombres de frames par seconde

D'après ces résultats, on constate que le gain de performances se situe entre 1/4 et 1/3 avec l'utilisation des FBOs. Le nombre de splats affichés par seconde pourrait alors atteindre les 12 369 690. Sachant que les chercheurs qui ont écrit l'article [BHZK05] obtiennent un résultat d'un peu plus de 20 000 000 de splats par seconde en utilisant une carte Geforce 6800 ultra GPU, un processeur Pentium 4 de 3.0GHz et sur une fenêtre de 512 x 512 pixels, il est possible que nos performances soient à peu près équivalentes, ce qui est très encourageant.

NPR

Une extension possible serait de créer de nouveau type de rendu comme par exemple un rendu non photo-réaliste tel que le *Toon Shading*.

10.3.3 Intégration

La troisième partie de notre projet consiste à mettre en commun la sélection dans l'arbre séquentiel de points des surfels à afficher avec les techniques de rendu réaliste implémentées en tant que *plugin PointShop3D*.

Le principal problème auquel nous sommes confrontés est l'adaptation des données fournies par l'arbre séquentiel de points à celles utilisées par le rendu réaliste. Par exemple, le rendu utilise pour afficher un surfel ses vecteurs tangents u et v (voir l'explication de l'algorithme de **HQSSRenderer**), et calcule le rayon à partir de ceux-ci, alors que la structure fournit directement le rayon mais pas u et v, qu'il faut donc calculer à partir de la normale.

L'autre grand problème de cette intégration est la récupération et l'éventuelle conversion des données de fenêtrage de *QGLViewer*.

Nous avons rencontré des complications au niveau du changement de rendu. Certains paramètres activés par des rendus ne devant pas l'être pour un autre. Il faut alors réinitialiser les paramètres OpenGL au début de chaque méthode de rendu.

Au niveau du rendu réaliste, les trois passes sont bien exécutées. Cependant, nous observons des défauts d'affichage, une transparence qui n'a pas lieu d'être dans la couleur des surfels. Comme nous l'avons décrit dans l'algorithme de profondeur de l'objet, il faut effectuer un décalage de la profondeur afin de laisser passer les surfels qui recouvrent. Cette valeur dépend d'une matrice fournie par *PointShop3D*. Nous n'avons pas eu le temps de nous consacrer à ce calcul qui, semblerait-il, résoudrait le problème, étant donné qu'actuellement certains surfels ne sont pas pris en compte.

10.4 Epilogue

Ce projet nous a permis de découvrir un nouveau domaine d'application : l'imagerie 3D. Cet apprentissage a renforcé notre envie de poursuivre nos études dans ce domaine pour la majorité d'entre-nous.

Les technologies utilisées étant trés récentes, nous avons été confrontés à un véritable manque de documentation. Ainsi, le temps qui nous a été imparti a été principalement consacré à des recherches et des expérimentations sur ces éléments.

Nous avons en outre appris la technique de programmation des shaders, et un des langages associés : GLSL.

C'est la première fois que nous nous retrouvions face à un projet d'une telle ampleur, il nous a donc fallu utiliser des techniques de génie logiciel avant de commencer l'implémentation effective, puis tout au long de celle-ci. Enfin, malgré les tensions que peuvent engendrer un tel projet, nous avons réussi à conserver la cohésion du groupe et la bonne ambiance en son sein.

Chapitre 11

Remerciements

Nous tenons à remercier M. Desbarats, qui nous a suivi tout le long de notre projet et nous a donné de nombreuses indications, en particulier sur la rédaction du mémoire. Nous remercions également M. Boubekeur, qui a toujours répondu très rapidement à nos questions concernant les points techniques de notre projet.

Bibliographie

[BHZK05] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. *Proceedings of the Eurographics Symposium on Point-Based Graphics*, 1 :1–8, 2005.

[BSK04] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. *Proceedings of the Eurographics Symposium on Point-Based Graphics*, 1 :1–8, 2004.

[BWK02] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *EGRW '02 : Proceedings of the 13th Eurographics workshop on Rendering*, pages 53–64, 2002.

[DVS03] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Trans. Graph.*, 22(3) :657–662, 2003.

[Gou71] Henri Gouraud. *Computer display of curved surfaces.* PhD thesis, 1971.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6) :311–317, 1975.

[RL00] S. Rusinkiewicz and M. Levoy. Qsplat : a multiresolution point rendering system for large meshes. In *SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 343–352, 2000.

[ZPKG02] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3d : an interactive system for point-based surface editing. In *SIGGRAPH '02 : Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 322–329, 2002.

[ZPvBG02] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 2002.

Annexe A

Manuel Technique de LargeObjectsViewer

Pour construire la partie **LargeObjectsViewer**, nous avons fourni un script shell personnalisé. Cela est nécessaire puisqu'il faut les bibliothèques *QT* (version supérieure ou égale à 4.0) et *gsl*.

En lançant le script (*./build.sh*), il est demandé les choses suivantes :

```
Chemin du repertoire contenant les binaires de qt4 (qmake, uic, moc)?[/usr/bin/]
Chemin du repertoire contenant les "include" de qt4 ?[/usr/include]
Chemin du repertoire contenant les bibliothèques de qt4 ?[/usr/lib]
Chemin du repertoire contenant les "include" de GSL ?[/usr/include/]
Chemin du repertoire contenant les bibliothèques de GSL ?[/usr/lib/]
```

Le script attend un répertoire valide. Le cas échéant, le script s'arrête. Si l'utilisateur ne tape rien, le choix par défaut est choisi.

Ensuite la construction de QGLViewer commence si cela n'a pas déja été fait.

```
Compilation de QGLViewer
```

Enfin, la compilation du **LargeObjectsViewer** se lance :

```
Compilation de LargeObjectsViewer
```

Un nettoyage des fichiers objets, temporaires, ... est effectué, et il ne reste plus qu'à exécuter le programme dans le répertoire */bin*.

Il ne faut pas oublier de se placer dans le répertoire *bin* pour l'exécuter (c'est-à-dire ne pas lancer *./bin/LargeObjectsViewer*), sinon le programme n'arrivera pas à lire les shaders.

Ce script est assez rudimentaire : par exemple, il ne vérifie pas si le répertoire donné comprend bien le bon fichier. Il aurait fallu utiliser les *autotools* mais des difficultés se sont présentées lors de la vérification de la bibliothèque *QT4*, assez peu répandu à l'heure actuelle puisque assez récente.

Annexe B

Valgrind LargeObjectsViewer : Structure

```
[21:34:29][keitaro@asgard][~/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests]
>> valgrind --leak-check=full --show-reachable=yes ./structure ~/temporaire/ply/bunny.ply
==5808== Memcheck, a memory error detector.
==5808== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==5808== Using LibVEX rev 1471, a library for dynamic binary translation.
==5808== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==5808== Using valgrind-3.1.0, a dynamic binary instrumentation framework.
==5808== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==5808== For more details, rerun with: -v
==5808==
Start Structure test...
Loading : /home/keitaro/temporaire/ply/bunny.ply
Number of vertices=35947
Number of faces=69451
Computing normals... Done.
Computing splat sizes... Done.
Building tree...Done.
...Structure test complete
==5808==
==5808== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 57 from 1)
==5808== malloc/free: in use at exit: 660 bytes in 41 blocks.
==5808== malloc/free: 570,190 allocs, 570,149 frees, 25,258,325 bytes allocated.
==5808== For counts of detected errors, rerun with: -v
==5808== searching for pointers to 41 not-freed blocks.
==5808== checked 634,612 bytes.
==5808==
==5808== 16 bytes in 1 blocks are still reachable in loss record 1 of 6
==5808==    at 0x401CBE7: calloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5808==    by 0x4446D26: (within /usr/lib/opengl/nvidia/lib/libGL.so.1.0.8178)
==5808==
==5808==
==5808== 20 bytes in 1 blocks are indirectly lost in loss record 2 of 6
==5808==    at 0x401CCDB: realloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5808==    by 0x804CB40: add_property (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CDDF: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==
==5808==
==5808== 296 (36 direct, 260 indirect) bytes in 4 blocks are definitely lost in loss record 3 of 6
==5808==    at 0x401B511: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5808==    by 0x804A0D1: my_alloc (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804A8FC: ply_put_comment (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CE78: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
```

```
==5808==
==5808==
==5808== 122 bytes in 17 blocks are indirectly lost in loss record 4 of 6
==5808==    at 0x401B511: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5808==    by 0x436B1FF: strdup (in /lib/libc-2.3.6.so)
==5808==    by 0x804A90A: ply_put_comment (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CE78: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==
==5808==
==5808== 348 (8 direct, 340 indirect) bytes in 1 blocks are definitely lost in loss record 5 of 6
==5808==    at 0x401CCDB: realloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5808==    by 0x804AF75: add_element (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CCC8: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==
==5808==
==5808== 458 bytes in 17 blocks are indirectly lost in loss record 6 of 6
==5808==    at 0x401B511: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5808==    by 0x804A0D1: my_alloc (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804AEE0: add_element (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CCC8: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5808==
==5808== LEAK SUMMARY:
==5808==    definitely lost: 44 bytes in 5 blocks.
==5808==    indirectly lost: 600 bytes in 35 blocks.
==5808==      possibly lost: 0 bytes in 0 blocks.
==5808==    still reachable: 16 bytes in 1 blocks.
==5808==         suppressed: 0 bytes in 0 blocks.
[21:36:28][keitaro@asgard][~/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests]
>> valgrind --leak-check=full --show-reachable=yes ./structure ~/temporaire/ply/happy.ply
==5810== Memcheck, a memory error detector.
==5810== Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
==5810== Using LibVEX rev 1471, a library for dynamic binary translation.
==5810== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==5810== Using valgrind-3.1.0, a dynamic binary instrumentation framework.
==5810== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==5810== For more details, rerun with: -v
==5810==
Start Structure test...
Loading : /home/keitaro/temporaire/ply/happy.ply
Number of vertices=543652
Number of faces=1087716
Computing normals... Done.
Computing splat sizes... Done.
Building tree...Done.
...Structure test complete
==5810==
==5810== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 57 from 1)
==5810== malloc/free: in use at exit: 482 bytes in 33 blocks.
==5810== malloc/free: 8,809,207 allocs, 8,809,174 frees, 388,466,083 bytes allocated.
==5810== For counts of detected errors, rerun with: -v
==5810== searching for pointers to 33 not-freed blocks.
==5810== checked 634,612 bytes.
==5810==
==5810== 12 bytes in 1 blocks are indirectly lost in loss record 1 of 6
==5810==    at 0x401CCDB: realloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5810==    by 0x804CB40: add_property (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CDDF: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==
==5810==
==5810== 16 bytes in 1 blocks are still reachable in loss record 2 of 6
==5810==    at 0x401CBE7: calloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5810==    by 0x4446D26: (within /usr/lib/opengl/nvidia/lib/libGL.so.1.0.8178)
==5810==
==5810==
==5810== 213 (28 direct, 185 indirect) bytes in 4 blocks are definitely lost in loss record 3 of 6
==5810==    at 0x401B511: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5810==    by 0x804A0D1: my_alloc (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804A8FC: ply_put_comment (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CE78: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804D0F1: PlyFormatFile::load(char*) (in
```

```
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==
==5810==
==5810== 90 bytes in 13 blocks are indirectly lost in loss record 4 of 6
==5810==    at 0x401B511: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5810==    by 0x436B1FF: strdup (in /lib/libc-2.3.6.so)
==5810==    by 0x804A90A: ply_put_comment (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CE78: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==
==5810==
==5810== 253 (8 direct, 245 indirect) bytes in 1 blocks are definitely lost in loss record 5 of 6
==5810==    at 0x401CCDB: realloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5810==    by 0x804AF75: add_element (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CCC8: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==
==5810==
==5810== 328 bytes in 13 blocks are indirectly lost in loss record 6 of 6
==5810==    at 0x401B511: malloc (in /usr/lib/valgrind/x86-linux/vgpreload_memcheck.so)
==5810==    by 0x804A0D1: my_alloc (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804AEE0: add_element (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CCC8: ply_read (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804CEE6: ply_open_for_reading (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804D0F1: PlyFormatFile::load(char*) (in
/home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x804F9D1: SPT::fillLeaves() (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==    by 0x805218A: main (in /home/keitaro/eclipse/pdp/trunk/LargeObjectsViewer/src/Structure/tests/structure)
==5810==
==5810== LEAK SUMMARY:
==5810==    definitely lost: 36 bytes in 5 blocks.
==5810==    indirectly lost: 430 bytes in 27 blocks.
==5810==      possibly lost: 0 bytes in 0 blocks.
==5810==    still reachable: 16 bytes in 1 blocks.
==5810==         suppressed: 0 bytes in 0 blocks.
```

Annexe C

Profil GProf de LargeObjectsViewer

Profil obtenu avec gprof, tronqué pour seulement faire apparaitre les détails (nombres d'appels,...)

```
[Profil obtenue avec gprof, tronqué pur seulement faire apparaitre les détails (nombres d'appels,
...)]

Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 53.85      0.07     0.07 44415547     0.00     0.00  Surfel::~Surfel()
 23.08      0.10     0.03 44413964     0.00     0.00  Surfel::Surfel(Surfel const&)
 15.38      0.12     0.02     1583     0.01     0.01  __gnu_cxx::__normal_iterator<Surfel*, std::vector<Su
rfel, std::allocator<Surfel> > > std::upper_bound<__gnu_cxx::__normal_iterator<Surfel*, std::vector<Surfel
, std::allocator<Surfel> > >, Surfel, SurfelsDesc>(__gnu_cxx::__normal_iterator<Surfel*, std::vector<Surfe
l, std::allocator<Surfel> > >, __gnu_cxx::__normal_iterator<Surfel*, std::vector<Surfel, std::allocator<Su
rfel> > >, Surfel const&, SurfelsDesc)

   7.69      0.13     0.01   813399     0.00     0.00  Surfel::Surfel()
   0.00      0.13     0.00  5981820     0.00     0.00  get_ascii_item
   0.00      0.13     0.00  5981820     0.00     0.00  store_item
   0.00      0.13     0.00  2719114     0.00     0.00  my_alloc
   0.00      0.13     0.00  1631378     0.00     0.00  get_words
   0.00      0.13     0.00  1631368     0.00     0.00  ascii_get_element
   0.00      0.13     0.00  1631368     0.00     0.00  ply_get_element
   0.00      0.13     0.00   811816     0.00     0.00  Surfel::~Surfel()
   0.00      0.13     0.00   811815     0.00     0.00  Surfel::addChild(Surfel*)
   0.00      0.13     0.00   811815     0.00     0.00  void std::__unguarded_linear_insert<__gnu_cxx::__nor
mal_iterator<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, Surfel, SurfelsDesc>(__gnu_cxx::__no
rmal_iterator<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, Surfel, SurfelsDesc)
   0.00      0.13     0.00   719420     0.00     0.00  std::vector<Surfel*, std::allocator<Surfel*> >::_M_i
nsert_aux(__gnu_cxx::__normal_iterator<Surfel**, std::vector<Surfel*, std::allocator<Surfel*> > >, Surfel*
 const&)

   0.00      0.13     0.00   536328     0.00     0.00  SPT::calculateTangentialError(Surfel*&)
   0.00      0.13     0.00   268164     0.00     0.00  SPT::calculateGeometricError(Surfel*&)
   0.00      0.13     0.00   191185     0.00     0.00  SPT::combineLeaves(int, int)
   0.00      0.13     0.00   191184     0.00     0.00  SPT::partition(int, int)
   0.00      0.13     0.00    92395     0.00     0.00  SPT::combineNodes(Surfel*, Surfel*, Surfel*, Surfel*)
   0.00      0.13     0.00    90697     0.00     0.00  SPT::combineNodes(Surfel*, Surfel*, Surfel*)
   0.00      0.13     0.00    85072     0.00     0.00  SPT::combineNodes(Surfel*, Surfel*)
   0.00      0.13     0.00    82499     0.00     0.00  __gnu_cxx::__normal_iterator<Surfel*, std::vector<Su
rfel, std::allocator<Surfel> > > std::__unguarded_partition<__gnu_cxx::__normal_iterator<Surfel*, std::vec
tor<Surfel, std::allocator<Surfel> > >, Surfel, SurfelsDesc>(__gnu_cxx::__normal_iterator<Surfel*, std::ve
ctor<Surfel, std::allocator<Surfel> > >, __gnu_cxx::__normal_iterator<Surfel*, std::vector<Surfel, std::al
locator<Surfel> > >, Surfel, SurfelsDesc)

   0.00      0.13     0.00     1602     0.00     0.00  SPT::hasData()
   0.00      0.13     0.00     1583     0.00     0.01  SPT::drawSelectedSurfelsList(Vector3D<float>, Vector
3D<float>, float, float, int)

   0.00      0.13     0.00     1348     0.00     0.00  LargeObjectsApplication::updateProgressBar()
   0.00      0.13     0.00       86     0.00     0.00  equal_strings
   0.00      0.13     0.00       56     0.00     0.00  LargeObjectsApplication::putFrameRate()
   0.00      0.13     0.00       21     0.00     0.11  std::vector<Surfel, std::allocator<Surfel> >::_M_ins
ert_aux(__gnu_cxx::__normal_iterator<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, Surfel const&)
```

```
  0.00      0.13     0.00        6     0.00     0.00  find_element
  0.00      0.13     0.00        5     0.00     0.00  get_prop_type
  0.00      0.13     0.00        4     0.00     0.00  add_property
  0.00      0.13     0.00        4     0.00     0.00  copy_property
  0.00      0.13     0.00        4     0.00     0.00  find_property
  0.00      0.13     0.00        4     0.00     0.00  ply_get_property
  0.00      0.13     0.00        2     0.00     0.64  SPT::~SPT()
  0.00      0.13     0.00        2     0.00     0.00  Viewer::setData(SurfelsStructureInterface*)
  0.00      0.13     0.00        2     0.00     0.00  add_element
  0.00      0.13     0.00        2     0.00     0.00  ply_get_element_description
  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to _ZN15StructureThreadC2E
P23LargeObjectsApplicationP25SurfelsStructureInterface

  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to _ZN23LargeObjectsApplic
ation16staticMetaObjectE

  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to _ZN23LargeObjectsApplic
ationC2Ev

  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to _ZN3SPTC2Ev
  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to _ZN6SurfelC2Ev
  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to _ZN6ViewerC2EP7QWidget
  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to main
  0.00      0.13     0.00        1     0.00     0.00  global constructors keyed to vertProps
  0.00      0.13     0.00        1     0.00     0.00  PlyFormatFile::getFacesList(std::vector<Vector3D<int
>, std::allocator<Vector3D<int> > >*&)

  0.00      0.13     0.00        1     0.00     0.00  PlyFormatFile::getVerticesList(std::vector<Surfel*,
std::allocator<Surfel*> >*&)

  0.00      0.13     0.00        1     0.00     6.68  PlyFormatFile::load(char*)
  0.00      0.13     0.00        1     0.00     0.00  PlyFormatFile::PlyFormatFile()
  0.00      0.13     0.00        1     0.00     0.00  StructureThread::StructureThread(LargeObjectsApplica
tion*, SurfelsStructureInterface*)

  0.00      0.13     0.00        1     0.00     0.00  LargeObjectsApplication::stopProgress()
  0.00      0.13     0.00        1     0.00     0.00  LargeObjectsApplication::putNbVertices(int)
  0.00      0.13     0.00        1     0.00     0.00  LargeObjectsApplication::switchLODView()
  0.00      0.13     0.00        1     0.00     0.64  LargeObjectsApplication::open()
  0.00      0.13     0.00        1     0.00     0.00  LargeObjectsApplication::quit()
  0.00      0.13     0.00        1     0.00     0.64  LargeObjectsApplication::work(QString)
  0.00      0.13     0.00        1     0.00     0.00  LargeObjectsApplication::createGUI()
  0.00      0.13     0.00        1     0.00     0.00  LargeObjectsApplication::LargeObjectsApplication()
  0.00      0.13     0.00        1     0.00     0.64  LargeObjectsApplication::~LargeObjectsApplication()
  0.00      0.13     0.00        1     0.00     6.68  SPT::fillLeaves()
  0.00      0.13     0.00        1     0.00     0.00  SPT::findRadius(std::vector<Vector3D<int>, std::allo
cator<Vector3D<int> > > const*)

  0.00      0.13     0.00        1     0.00     3.30  SPT::buildLeaves(int, int)
  0.00      0.13     0.00        1     0.00     0.00  SPT::getNbVertices()
  0.00      0.13     0.00        1     0.00     0.00  SPT::initGLAndShader()
  0.00      0.13     0.00        1     0.00     0.00  SPT::setRMaxToAllSurfels(Surfel*&, int)
  0.00      0.13     0.00        1     0.00     0.00  SPT::findNormalsFromFaces(std::vector<Vector3D<int>,
 std::allocator<Vector3D<int> > > const*)

  0.00      0.13     0.00        1     0.00     4.74  SPT::putSurfelsIntoVector(Surfel*&, int)
  0.00      0.13     0.00        1     0.00     0.00  SPT::putStructureIntoMemory()
  0.00      0.13     0.00        1     0.00     0.00  SPT::calculateGeometricErrorForAllSurfels(Surfel*&)
  0.00      0.13     0.00        1     0.00   108.56  SPT::build()
  0.00      0.13     0.00        1     0.00    93.84  SPT::sortSPT()
  0.00      0.13     0.00        1     0.00     3.30  SPT::buildTree()
  0.00      0.13     0.00        1     0.00     0.00  SPT::SPT(char*, FileLoaderInterface*, short)
  0.00      0.13     0.00        1     0.00     0.00  SPT::SPT()
  0.00      0.13     0.00        1     0.00     0.00  Viewer::setMode(int)
  0.00      0.13     0.00        1     0.00     0.00  Viewer::Viewer(QWidget*)
  0.00      0.13     0.00        1     0.00     0.00  void std::__insertion_sort<__gnu_cxx::__normal_itera
tor<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, SurfelsDesc>(__gnu_cxx::__normal_iterator<Sur
fel*, std::vector<Surfel, std::allocator<Surfel> > >, __gnu_cxx::__normal_iterator<Surfel*, std::vector<Su
rfel, std::allocator<Surfel> > >, SurfelsDesc)

  0.00      0.13     0.00        1     0.00    81.10  void std::__introsort_loop<__gnu_cxx::__normal_itera
tor<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, int, SurfelsDesc>(__gnu_cxx::__normal_iterato
r<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, __gnu_cxx::__normal_iterator<Surfel*, std::vect
or<Surfel, std::allocator<Surfel> > >, int, SurfelsDesc)

  0.00      0.13     0.00        1     0.00    12.74  void std::__final_insertion_sort<__gnu_cxx::__normal
_iterator<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, SurfelsDesc>(__gnu_cxx::__normal_iterat
or<Surfel*, std::vector<Surfel, std::allocator<Surfel> > >, __gnu_cxx::__normal_iterator<Surfel*, std::vec
tor<Surfel, std::allocator<Surfel> > >, SurfelsDesc)

  0.00      0.13     0.00        1     0.00     0.00  Vector3D<int>* std::__uninitialized_fill_n_aux<Vecto
r3D<int>*, unsigned int, Vector3D<int> >(Vector3D<int>*, unsigned int, Vector3D<int> const&, __false_type)

  0.00      0.13     0.00        1     0.00     0.00  Surfel** std::fill_n<Surfel**, unsigned int, Surfel*
>(Surfel**, unsigned int, Surfel* const&)

  0.00      0.13     0.00        1     0.00     0.00  add_comment
  0.00      0.13     0.00        1     0.00     0.00  check_types
  0.00      0.13     0.00        1     0.00     0.00  get_native_binary_type
  0.00      0.13     0.00        1     0.00     0.00  ply_close
```

```
  0.00      0.13     0.00        1     0.00     0.00  ply_open_for_reading
  0.00      0.13     0.00        1     0.00     0.00  ply_put_comment
  0.00      0.13     0.00        1     0.00     0.00  ply_read

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
   else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
   function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
   the function in the gprof listing. If the index is
   in parenthesis it shows where it would appear in
   the gprof listing if it were to be printed.
```

Annexe D

Exemple de fichier *ply*

```
ply
format ascii 1.0
comment author: Greg Turk
comment object: another cube
element vertex 8
property float x
property float y
property float z
property red uchar                    { start of vertex color }
property green uchar
property blue uchar
element face 7
property list uchar int vertex_index  { number of vertices for each face }
element edge 5                        { five edges in object }
property int vertex1                  { index to first vertex of edge }
property int vertex2                  { index to second vertex }
property uchar red                    { start of edge color }
property uchar green
property uchar blue
end_header
0 0 0 255 0 0                         { start of vertex list }
0 0 1 255 0 0
0 1 1 255 0 0
0 1 0 255 0 0
1 0 0 0 0 255
1 0 1 0 0 255
1 1 1 0 0 255
1 1 0 0 0 255
3 0 1 2                           { start of face list, begin with a triangle }
3 0 2 3                           { another triangle }
4 7 6 5 4                         { now some quadrilaterals }
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
0 1 255 255 255                   { start of edge list, begin with white edge }
1 2 255 255 255
2 3 255 255 255
3 0 255 255 255
2 0 0 0 0                         { end with a single black line }
```