Dynamic Sampling and Rendering of Algebraic Point Set Surfaces

Méthodes et algorithmes

20 Novembre 2016

William Caisson
Xavier Chalut
Christophe Claustre
Thibault Lejemble

A destination de :
Nicolas Mellado
David Vanderhaeghe

Table des matières

1 Contexte

Avec l'arrivée des méthodes de numérisation 3D telles que le scanner laser, la représentation par nuage de points devient bien plus courante pour représenter les surfaces d'objets en 3 dimensions. Cette représentation de plus en plus facile à obtenir, est assez complexe à visualiser. Les problèmes majeurs étant le bruit généré lors de l'acquisition des points et la quantité de données à manipuler si l'on souhaite représenter avec une grande précision la scène d'origine.

L'article [2] propose une méthode réglant le problème des points bruités et permettant une visualisation pertinente de la scène même avec une plus faible quantité de données pour la représenter.

L'objectif de ce chef d'œuvre est de parvenir à une implémentation de cette méthode de visualisation de nuage de points au sein d'un plugin du moteur de rendu Radium. Dans un second temps, l'objectif sera de rendre l'implémentation la plus efficace possible en se basant sur l'utilisation de structures de données et d'algorithmes optimisés pour GPU par l'utilisation de la technologie CUDA.

2 Algorithme principal

Visualisation de nuage de points (0) Implémentation dans *Radium* d'un plugin simple de visualisation de nuages de points. Ce plugin intègre un étalement des points afin de les visualiser correctement. Toute référence à un point sous-entend une structure contenant au minimum une position et une normale.

Algorithm 1 $Dynamic\_sampling(nuage\_points) \rightarrow nouveaux\_points$

```
visibles ← selection_visible(nuage_points) (1)
for all point in visibles do
    sous_points ← reechantillonage(p) (2)
    voisins ← selection_voisins(p, nuage_points) (3)
    for all sous_point in sous_points do
        sous_point.position ← reprojection(sous_point, voisins) (4)
        ajout(nouveaux_points, sous_point)
    end for
end for
return nouveaux_points
```

Sélection dynamique des points à traiter (1) Sélection des points du nuage de points qui devront être visible sur le rendu final du point de vue de la caméra.

Ré-échantillonnage (2) Ré-échantillonnage des points visibles. Le nombre de points finaux étant défini selon l'angle et la distance à la caméra du point d'origine.

Sélection des points nécessaires à la projection (3) Sélection du voisinage du point d'origine qui affectera la reprojection des points issus du ré-échantillonnage.

Reprojection des points issus du ré-échantillonnage (4) Afin de déterminer quelle projection doit être effectuée sur chaque point issu du ré-échantillonnage, une opération de détermination locale de la surface de la scène est effectuée. Cette opération de détermination de la surface utilisera le principe de l'APSS [3].

3 Visualisation de nuage de points

Radium, comme tout moteur 3D, utilise des bibliothèques qui se chargent de l'affichage et de sa paramétrisation. Dans notre cas, *Radium* utilise la spécification *OpenGL* qui intègre déjà un affichage de points. Le travail à effectuer se résumera donc à l'algorithme suivant :

Algorithm 2 $visualisation(points\_a\_visualiser, radius)$

```
visualiseur.type_primitive(POINT)
visualiseur.radius(radius)
visualiseur.set_constante("direction", camera.direction)
for all p in points_a_visuliser do
  visualiseur.dessiner(p)
end for
```

Faire uniquement cette étape nous affichera des carrés toujours orientés vers la caméra et non selon leur normale. Nous devons donc spécifier au pipeline graphique comment afficher nos points. Ceci se fera essentiellement via le fragment shader. Ici nos points étalés sont affichés en rouge.

Algorithm 3 $Fragment\ shader(pt\_pos, pt\_normal, coordonnee) \rightarrow color$

```
cercle3d ← calculCercle3d(pos, normal)
ellipse2D ← projection2DVersEcran(cercle3D)
if coordonnee_depuis_point(coordonnee) ∉ ellipse2D then
  jeter_frangment()
end if
color ← red
```

4 Sélection dynamique des points à traiter

Le calcul à effectuer étant assez complexe en temps et en mémoire, il serait contre productif de le faire pour des points qui seront de toutes façons non visibles sur le rendu final. L'algorithme consiste à écarter tous les points qui ne sont pas compris dans le champ de vision de la caméra, puis ceux dont la normale n'est pas orientée vers la caméra, comme illustre la Figure 1.

Pour déterminer si la normale du point est orientée vers la direction de la caméra, il faut étudier leur produit scalaire. Si celui-ci est négatif, alors les deux vecteurs sont orientés l'un vers l'autre, donc nous ne traiterons pas le point si le produit est positif.

La méthode se résumera donc à l'algorithme 4.

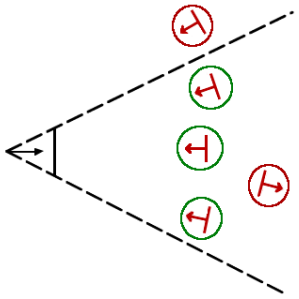

FIGURE 1 – Ici, nous avons à gauche la caméra et à droite les points du nuage de points. Les points non sélectionnés sont entourés en rouge.

Algorithm 4 *selection_visible*(*nuage_points*) → *visible*

```
for all p in nuage_points do
  if ecran.est_dans_champ_de_vision(p)
  & camera.est_oriente_vers(p) then
    visible.ajout(p)
  end if
end for
return visible
```

5 Ré-échantillonnage

A ce niveau là de l'algorithme principal, il est nécessaire de sur-échantillonner les points initiaux. Cependant le nombre de nouveaux échantillons pour avoir une bonne qualité visuelle au final n'est pas une valeur fixe. En effet ce nombre doit dépendre de l'angle de vue entre la caméra et le disque du point, et également de la courbure de la zone autour du point. Si un point est situé dans une **zone plane** faisant **face à la caméra** le nombre d'échantillons n'a pas besoin d'être très conséquent.

Pour cela, pour des questions de temps de calcul, nous ne pouvons pas calculer la surface réelle de la scène. Nous allons donc utiliser la courbure principale locale.

$$\lambda_i = \max_j \frac{\|n_i - n_j\|}{\|p_i - p_j\|} \tag{1}$$

Si p_i et n_i représentent respectivement la position et la normale de notre point, λ_i détermine, à partir des voisins de p_i, la courbure la plus grande pondérée par la distance à notre point. à partir de ceci, on peut déterminer le rayon de la sphère représentant le mieux la surface localement (notre rayon sera donc λ_i^{-1}).

Ensuite, nous pouvons calculer quelle sera l'erreur maximum générée par notre ré-échantillonnage sans la projection. Cette erreur est calculée en maximisant la distance entre la surface tangente à la sphère du point p_i et la sphère suivant la direction v_i, v_i étant la direction de vue.

Dans cette formule, on définit η comme étant le facteur d'échelle entre le point p_i et sa surface représentative et t_s la taille en pixel souhaitée par échantillon. t_s est un paramètre à fixer. Le m obtenu finalement nous permet de définir le nombre d'échantillon $m \times m$ nécessaire à une bonne représentation du point.

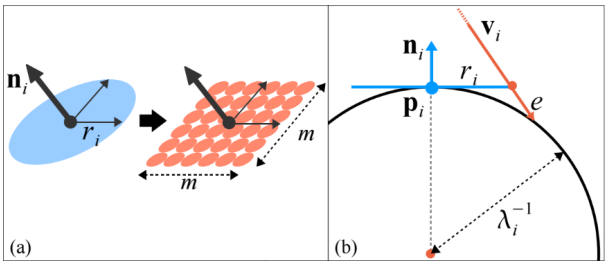

FIGURE 2 – (a) Illustration de l'algorithme de sur-échantillonnage. (b) Vue 2D illustrant la dépendance entre l'erreur géométrique et le point de vue. Le disque tangent représentant p_i est affiché en bleu et le vecteur de vue en rouge

$$m = \sqrt{\eta \frac{\min(e, r_i)}{t_s}} \tag{2}$$

En définissant η comme étant le facteur d'échelle entre le point p_i et sa surface représentative sur l'écran et t_s la taille en pixel souhaitée par échantillon, on peut appliquer l'Équation 2 pour obtenir le nombre d'échantillons $m \times m$ nécessaire.

6 Sélection des points nécessaires à la projection

Afin de visualiser une surface lisse qui approxime au mieux le nuage de points, une méthode dite de "Moving Least Square" permet de créer une telle surface sur laquelle les points du nuage sur-échantillonnés vont être projetés. Cette technique, décrite dans la Section 7, a besoin de connaître pour chaque point x du nuage, l'ensemble des points voisins p_i qui l'influencent. Cette influence entre points s'exprime suivant un calcul de poids donné par l'équation :

$$w_i(x) = \phi\left(\frac{||p_i - x||}{r_i h}\right) \tag{3}$$

où r_i est le rayon associé au point p_i et h est un paramètre d'échelle global permettant d'ajuster le rayon d'influence des points. La fonction ϕ est une fonction polynomiale décroissante dont la formulation compacte est :

$$\phi(u) = \begin{cases} (1-u^2)^4 & \text{si} \quad u < 1 \\ 0 & \text{sinon} \end{cases} \tag{4}$$

Ainsi, pour chaque point du nuage x, les points voisins p_i tels que $w_i(x) > 0$ doivent être sélectionnés. D'après l'Équation 3, il faut donc trouver toutes les boules centrées en p_i et de rayon $r_i h$ contenant x. Afin d'accomplir cette requête de voisinage le plus rapidement possible, une structure accélératrice, de type octree, partitionnant intelligemment l'espace doit être mise en place, éventuellement sur GPU (voir Section 8.1). L'article [2] propose et étudie les performances de différents types d'octree, parmi lesquels, l'**octree à référence unique** présente de meilleures performances en terme de temps de construction et de place mémoire.

Dans ce type d'octree, une boule de centre p_i et de rayon $r_i h$ est référencée uniquement dans la cellule de niveau l contenant p_i et de taille minimale et supérieure à $t \cdot 2r_i h$, où t est un paramètre global. Ainsi, lors de la requête de voisinage d'un point x, ses voisins sont forcément

contenus dans des cellules de niveau l intersectant une boule centrée en x et de rayon $\frac{w_l}{2t}$, où w_l correspond à la taille de la cellule de niveau l.

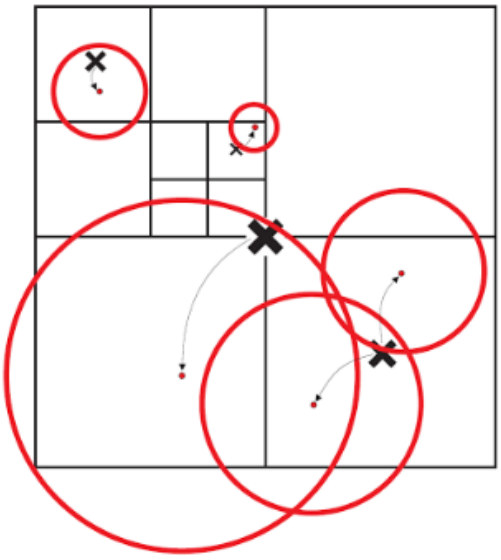

FIGURE 3 – Analogie 2D de l'octree à référence unique. Chaque boule est référencée par l'unique cellule qui contient son centre.

7 Projection des points issus du ré-échantillonnage

Afin de projeter nos points issus de l'étape précédente, l'article [2] suggère d'utiliser une représentation de la surface via le calcul d'une Algebraic Point Set Surface (APSS). Ensuite, la projection de nos nouveaux points sur cette APSS se fait via une projection dite presque orthogonal, comme décrite dans [1].

7.1 APSS

L'APSS est calculé de manière locale pour chacun des points du nuage de point. L'APSS se définit à partir de la sphère algébrique associée à un point x noté $u(x)$ minimisant sa distance par rapport au voisinage de x, avec w_i provenant de l'Équation 3 :

$$u(x) = \arg\min_u \sum_i w_i(x) \left(s_u(p_i)^2 + \|\nabla s_u(p_i) - n_i\|^2\right) \tag{5}$$

La sphère $u(x)$ est définie par le vecteur $[u_1, \ldots, u_4]^T$ et par :

$$c = -(\frac{1}{2u_4})[u_1, u_2, u_3]^T, r = \sqrt{(c^T c - \frac{u_0}{u_4})} \tag{6}$$

où c et r correspondent respectivement au centre et au rayon de la sphère. Avec $\tilde{w}_i = w_i / \sum_j w_j$, u_4 est un facteur d'échelle définit par :

$$u_4 = \beta \frac{1}{2} \frac{\sum w_i p_i^T n_i - \sum \tilde{w}_i p_i^T \sum w_i n_i}{\sum w_i p_i^T p_i - \sum \tilde{w}_i p_i^T \sum w_i p_i} \tag{7}$$

La variable β est paramétrable manuellement pour moduler la taille des sphères et ajuster le niveau de détail du rendu final.

7.2 Projection "presque" orthogonale

Après avoir ré-échantillonné nos points, il faut les projeter sur la surface sous-jacente. Pour cela l'article [2] suggère d'utiliser la projection presque orthogonale, une façon simple et efficace d'obtenir de bons résultats.

La méthode de projection presque orthogonale est décrite avec précision dans l'article [1]. Cette méthode s'initialise en calculant, à partir du point initial, un plan sur lequel on projette orthogonalement notre point d'origine. Puis plusieurs itérations sont effectuées, chacune consistant en le calcul d'un nouveau plan de projection à partir du point résultant de la projection précédente et une opération de projection (orthogonale) toujours appliqué sur le point d'origine. On répète les itérations jusqu'à convergence.

Algorithm 5 $presque\_orthogonale(x) \rightarrow x'$

$x' \leftarrow x$
repeat
  $a \leftarrow a(x')$
  $n \leftarrow n(x')$
  $x' \leftarrow x - n^T(a - x)n$
until $\|n^T(a - x')\| \leq \varepsilon$
return x'

Pour calculer $a(x)$ et $n(x)$ il faut trouver le plan minimisant les distances entre ce plan et l'intégrale des points de la sphère définit par APSS pondérés par leur distance au point x. Une fois ce plan déterminé, $a(x)$ représente la position du plan et $n(x)$ sa normale.

7.3 Intégration

La reprojection des points issus du ré-échantillonnage est gérée par la bibliothèque *Patate* qui nous a été conseillée par notre client. Cependant, cette bibliothèque ne gère que le premier niveau d'itération c'est à dire l'itération de la projection presque orthogonal sur une première estimation de la surface de l'APSS. Donc le travail à effectuer sera similaire à l'algorithme :

Algorithm 6 $reprojection(sp) \rightarrow sp'$

$actuel \leftarrow precedent$
repeat
  $precedent \leftarrow actuel$
  $actuel \leftarrow presque\_orthogonale(precedent)$
until $distance(preceddent, actuel) < \varepsilon$
return actuel

La bibliothèque *Patate* ne définissant aucun type, elle fonctionne selon un système de template, nous devrons donc définir tous les types à utiliser.

8 Accélération sur GPU

La partie accélération sur GPU étant un objectif secondaire de ce chef d'œuvre, cette partie ne seras pas très détaillée. Les paragraphe suivant servent à présenter les principaux avantages qu'offre l'accélération CUDA sur un GPU.

8.1 Octree sur GPU

Pour de meilleures performances, la construction de la structure de données pour l'accès au voisinage décrite en Section 6 peut être déléguées au GPU. L'octree à référence unique devient ainsi un octree complet, et donc statique, où toutes les feuilles sont explicitement créées et stockées. L'indice i d'un point p_i est alors mémorisé par toutes les cellules le recouvrant, et tous les indices de ces cellules sont enregistrés dans un tableau commun à tous les p_i. Les éléments de ce tableau entre les indices $b \cdot i$ et $b \cdot (i+1)$ correspondent alors aux indices des cellules qui contiennent p_i, où b est le niveau de profondeur maximum de l'octree.

L'un des principaux enjeux est le partage de cette structure de données par tous les threads y insérant les points du nuage. Puisque les mutex n'existent pas sur GPU, une autre approche est adoptée. Un vecteur M de taille m est alloué et partagé par tous les threads. Lorsque le thread d'indice i_t veut écrire dans une cellule d'indice i_c, il écrit son indice dans $M[i_c \% m]$ et vérifie que $M[i_c \% m] = i_t$. Si cette égalité n'est pas vérifiée (un autre thread tente donc d'accéder à la même cellule), le thread attend un court instant et ré-effectue l'opération, jusqu'à ce que le condition soit enfin valide (l'autre thread a ainsi terminé son opération). La cellule peut ensuite être modifiée sans problèmes. Les probabilités d'accès concurrents à une même cellule étant considérés faibles, le temps de construction de l'octree n'est pas affecté.

8.2 Ré-échantillonnage sur GPU

Le ré-échantillonnage peut être lui aussi effectué sur le GPU. Ceci permet tout d'abord de pouvoir essayer d'allouer un thread par p_i, ce qui va déjà grandement accélérer les temps de calcul. Aussi cela va permettre d'exploiter une cohérence temporelle de bas niveau. En effet, si à l'itération précédente le nombre de sous-échantillon généré est le même que le nombre dont on a besoin à l'itération courante, on peut simplement réutiliser ces échantillons calculés précédemment.

Pour exploiter cette cohérence temporelle, on va retenir tous les échantillons générés d'une itération à l'autre dans un tableau. Aussi on retiens, pour chaque p_i, le nombre nb_i d'échantillon généré et δ_i la position dans le tableau des échantillons du premier échantillon (les autres étant les $nb_i - 1$ suivant). Ainsi, au début d'une itération, on créé un autre tableau contenant l'état de chaque point, si un p_i n'est plus visible ou ne nécessite pas le même nombre de sous-échantillon, on le note comme étant $a\_remplacer$ sinon on le note comme étant $a\_reutiliser$.

Ensuite, on organise le tableau de façon à regrouper les points $a\_reutiliser$ afin de réserver l'espace mémoire pour leurs échantillons et ainsi connaître l'adresse mémoire à partir de laquelle on commence à retenir les échantillons qui doivent être calculés à cette itération.

Références

[1] ALEXA M. ADAMSON A. On Normals and Projection Operators for Surface defined by Point Sets. *In Proceedings of the Eurographics Symposium on Point-Based Graphics*, 2004.

[2] GUENNEBAUD G. GERMANN M. GROSS M. Dynamic Sampling and Rendering of Algebraic Point Set Surfaces. *EUROGRAPHICS 2008 / G. Drettakis and R. Scopigno (Guest Editors)*, 27(3), 2008.

[3] GUENNEBAUD G. GROSS M. Algebraic Point Set Surfaces. *ACM Transactions on Graphics (SIGGRAPH 2007 Proceedings)*, 26(3) :23.1–23.9, 2007.