Mémoire de stage

Arnaud Delpeyroux

Table des matières

Remerciements

Je tiens à remercier tout d'abord M. Gael Guennebaud, M. Pierre Bénard et M. Pascal Barla pour avoir accepter de me prendre en stage au seis de l'équipe Manao et pour m'avoir encadré et beaucoup aidé lors de ce stage.

Je tiens aussi a remercier les autres stagiaires, les doctorants et post-docs de l'équipe pour leur accueil, leur intérêt et leur aide, plus particulièrement M. Georges Nader et M. Thibaud Lambert.

Introduction

C'est au cours de mon cursus en master 1 Informatique, spécialité Informatique pour l'Image et le Son, que j'ai eu l'opportunité d'effectuer un stage optionnel lors de l'été 2017. Ce stage a donc été réalisé dans l'institut de recherche `INRIA` à Bordeaux et plus précisément dans l'équipe `Manao` sous l'encadrement de M. Gael Guennebaud, M. Pierre Bénard et M. Pascal Barla. Cette équipe a pour objectif d'étudier les interaction en formes, lumière et matière. Cela se concrétise par des travaux sur du rendus graphique temps réel ou non, d'étude de la lumière, de l'animation, du niveaux de détail, du rendu expressif. C'est donc dans cette équipe que j'ai réaliser mon stage ayant comme sujet : *Cartes de déplacement procédurales pour le rendu temps réel d'objets 3D détaillés*. L'objectif de ce stage est d'étudier la génération de géométrie (ou déformation) sur une surface 3D à l'aide de primitives géométriques déterminées et d'une distribution sur la surface choisie le tout en temps réel.

Nous detaillerons ici le travail éffectué pour répondre a notre objectif. Pour ce faire nous feront un petit état de l'art pour présenté le pipeline de rendu OpenGL, les méthode de `displacement mapping` et notre approche du problème. Ensuite comment nous avons calculée la distribution des primitive géométrique sur al surface, d'abord en temps réel, puis en pré calculant la distribution. Enfin comment nous évaluons les déformations de la surface selon la méthode choisie pour la distribution.

1 État de l'art

1.1 Pipeline OpenGL

1.1.1 Pipeline de rendu

Le pipeline OpengL est la séquence d'étapes que l'API vas suivre pour afficher des objets 3D. [1]

Afin de d'afficher une scène et donc d'avoir en sortie une image (un tableau de pixel), l'API a besoin en entrée d'un point de vue, la description de la scène (polygones, points, lumieres, ...), l'ensembles des attribus pour chaques objets (normales, tangentes, couleurs, ...).

Le pipeline fonctionne de la manière suivante :

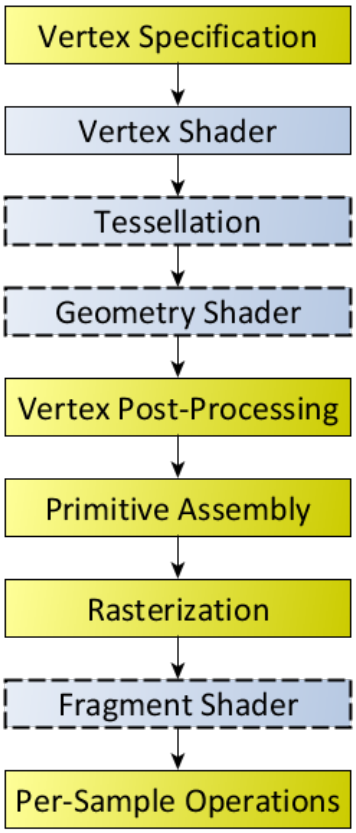


FIGURE 1 – Pipeline OpenGl.

- On prepare l'ensemble des informations concernant nos primitives (sommets, faces, ...) et leurs attributs (normales, couleurs, ...) et on les transmet à la carte graphique.
- `Vertex processing` : chaque sommet est traité par le `vertex shader` [2]. Le `vertex shader` est un étage programmable qui en entrée prends un sommets et tous ses attributs, effectuer un traitement déterminer par le developpeur, puis en sortie retourner le sommet modifié (avec modification possible de ses attributs). Ensuite le sommet passe dans deux étages optionnels du pipeline, la `tessellation` et le `geometry shader`. L'étage de `tessellation` sera décris plus loin étant donné qu'il prends une place importante dans la solution décrite dans ce mémoire. Le `geometry shader` est un `shader` qui en entrée prend une primitive (une face) en entrée et retourne en sortie zero ou plusieurs primitives.
- `Vertex Post-Processing` : à cette étape, les sommets en sortie des étages précedents passent par diverses opérations tel que le `clipping` qui est la suppression ou le découpage des primitives hors du cone de vision. A cette étape les coordonées des sommets sont envoyé dans le repere de la fenetre, ces coordonées seront utilisées pour la `rasterization`.
- `Primitive Assembly` : converssion d'un flux de sommets en primitives de bases (lignes, triangles, quads)

1. Cette section est essentiellement basée sur la page `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`

2. Un shader est un programme permettant de personnaliser une partie du pipeline graphique. Il en existe plusieurs types qui ont des finalitées différentes. Le plus souvents ils servent a calculer l'éclairement, l'ombrage, le déplacement de primitives ou divers effets de post-traitement

— `Rasterization` : procédé permettant de determiner quels sont les pixels "couvert" par une primitive donné. Ainsi pour chaque primitive de base on génère des `fragments` (correspondants à des pixels).

— `Fragment Shader` : étage programmable obligatoire qui va venir appliquer un traitement définis par le developpeur à chaque fragment

— `Per-Sample Processing` : chaque `fragment` est traité en sortie du `fragment shader` et le résulat est stocké dans divers buffers.

1.1.2 Tesselation matérielle :

Comme nous avons vu précédemment, la tessellation est une partie optionnelle du pipeline de rendu `OpenGL` [3]. Dans le pipeline elle prend place après le `vertex shader`. Cette partie est composée de deux étages programmables et d'un étage fixe. L'objectif de la tessellation est de subdiviser un patch, qui peut être triangulaire, quadrangulaire ou même une `isoline`, en des patchs de même nature.

Les trois étages sont :

— Le `Tessellation Control Shader` (programmable) ;

— Le `Tessellation Primitive Generator` (fixe) ;

— Le `Tessellation Evaluation Shader` (programmable).

Tesselation Control shader : L'objectif du `tessellation control shader` est principalement de déterminer le niveau de tessellation pour la primitive en entrée. Cet étage a aussi la responsabilité de garantir la correspondance des différents niveaux de tessellation entre deux patchs partageant une même arête. En effet, si deux patchs voisins n'ont pas le même niveau de tessellation alors des "déchirures" et des discontinuités apparaissent entre les patchs. Afin de garantir cette condition et de donner la possibilité d'utiliser différents niveaux de tessellation pour différents patchs, le `tesselaltion control shader` propose plusieurs niveaux de tesselation par patch, 4 extérieurs pour les arêtes (outter level) et deux intérieurs pour la tessellation à l'intérieur du patch (inner level). A noter que pour les patchs triangulaires seuls trois niveaux extérieurs et un intérieur sont nécessaires. Sur la figure suivante on peut voir a quoi correspondent ces différents niveaux.

3. `https://www.opengl.org/`

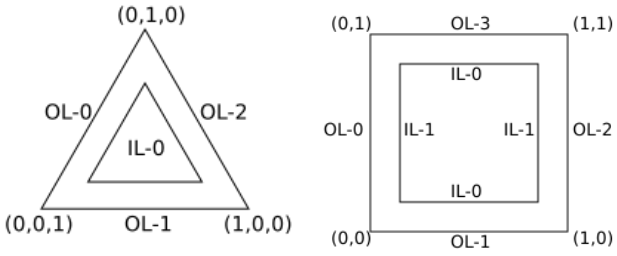


FIGURE 2 – Niveaux de tessellation.

De plus, dans le `tessellation control shader`, nous pouvons appliquer toutes transformations souhaitée sur les sommets du patch en entré. Cette possibilité ne sera pas utilisée dans la solution proposée dans ce mémoire.

Tesselation Primitive Generator : Cet étage fixe a pour fonction principale de générer les nouvelles primitives à partir du patch en entré et de différents paramètres. La génération des primitives dépend en effet des niveaux de tessellation renseignés lors de l'étage précédent, des règles d'espacements, du type des primitives d'entrées définies dans le `tessellation evaluation shader`, ainsi que l'ordre de génération. Dans la figure suivante on peut voir des exemples de tessellation sur différentes primitives avec différents niveaux.

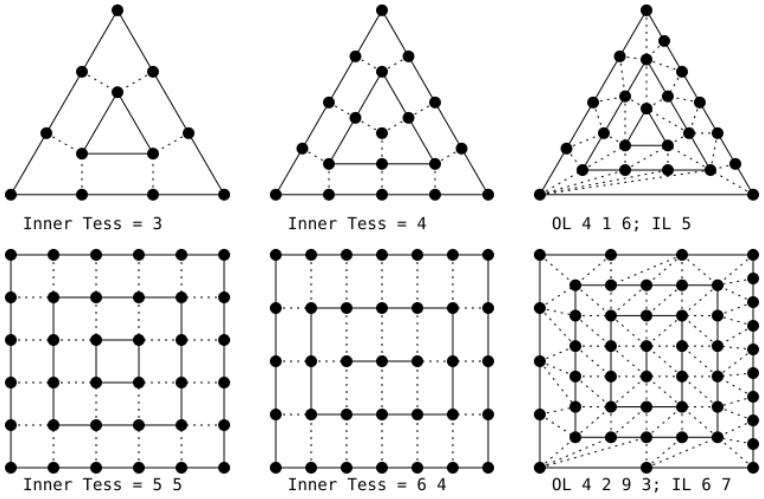


FIGURE 3 – Génération des primitives.

Tesselation Evalutaion Shader Quand le `tessellation primitive generator` finis de subdiviser le patch le `tessellation evaluation shader` est appelé pour chaque nouveau sommet du patch tessellé. Ainsi pour chaque sommet nous avons en entré les trois (ou quatre) sommets du patch d'origine et les coordonnées du sommet dans le patch (coordonnés barycentrique dans le cas d'un triangle). Il faut donc pour chaque nouveau sommet interpoler

sa position et tous ses autres attributs dans ce shader. Ensuite, on peut appliquer le traitement souhaité au sommet (un déplacement dans notre cas).

Nous nous intéresserons ici seulement de la tessellation matérielle dans le cas des faces triangulaires.

1.2 Displacement mapping

Cette partie visera à présenter le `displacement mapping`, mais aussi d'autres techniques permettant de créer de la géométrie à la surface d'un objet 3D.

La technique la plus utilisée afin de créer des détails géométriques sur un surface est le fait d'utiliser le `displacement mapping per vertex` en utilisant soit une `heigth map`[4] ou bien une fonction de bruit (bruit de gabor citeLLDD2009PNUSGC ou spot noise citevan-Wijk :1991 :SNT :127719.122751, par exemple). C'est-à-dire, pour chaque sommet de votre surface d'origine ou bien tessellé (le plus souvent tessellé, pour obtenir une résolution plus grande et donc des détails géométriques plus précis), on vient soit, évaluer une valeur depuis une fonction de bruit, ou lire dans une texture encodant les valeurs de déplacement pour appliquer un déplacement le long de la normale avec la valeur obtenue.

Nous pouvons voir ci-dessous deux exemples de déplacement utilisant des `height maps` dans le cas simple d'un simple plan. A noter que la première colonne represente l'objet d'origine en visualisant les coordonnées de textures.

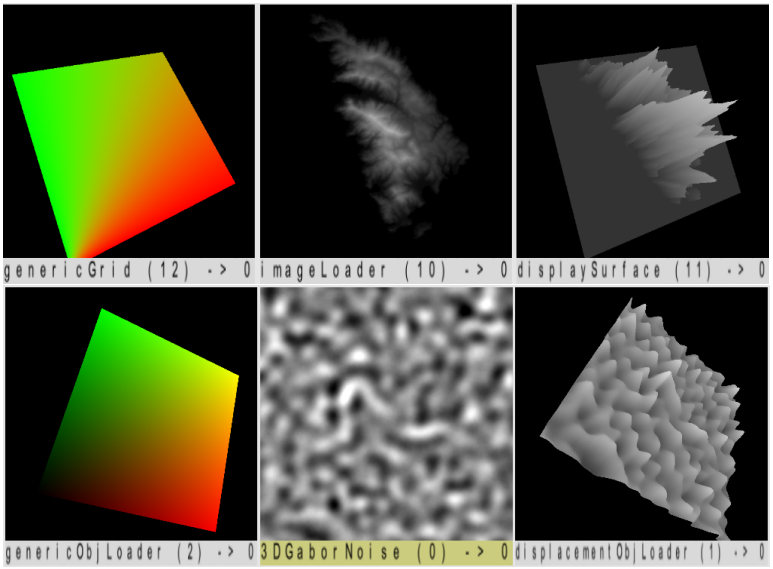


FIGURE 4 – Exemples de displacement mapping.

La technique décrite au-dessus à l'avantage d'être simple, mais comporte certaines limites. Quand on utilise une `height map`, nous avons besoin de renseigner des coordonées de textures

4. Cartes de déplacements, il s'agit de textures encodant des valeurs de déplacement. Ces valeurs de déplacement peuvent être en nuances de gris dans le cas d'un déplacement scalaire (déplacement le long de la normale au sommet courant), ou en RGB dans le cas d'un déplacement vectoriel (un déplacement quelconque par rapport au sommet courant)

pour tous nos sommet, de plus cela entraine des répétitions et limite la résolution à celle de la `height map` (même limites que dans l'utilisation d'une texture classique pour déterminer la couleur des pixels). Dans le cas de l'évaluation d'un bruit, plusieurs possibilités existent. En effet, on peut générer une `height map` à l'aide de notre fonction de bruit, mais ceci entraine les mêmes problèmes que décris précédemment. On peut aussi directement l'évaluer au moment d'appliquer notre déplacement, pour ce faire il nous faut une paramétrisation, on peut utiliser la position 3D du point pour évaluer un bruit 3D, utiliser les coordonnées de textures (si elles existent). Aussi, comme décris dans la référence [2], on peut (par exemple pour le bruit de gabor) projeter une distribution locale sur le plan tangent du sommet à déplacer pour évaluer le bruit sur ce plan pour enfin appliquer la valeur obtenue pour appliquer un déplacement (dans le cadre de l'article, cette technique est utilisée pour évaluer une texture à la surface, mais on peut aisément adapter cette technique pour venir déplacer nos sommets).

Aussi, d'autres techniques tel que les `Shell Maps`, présenté dans la référence [4], permettent d'ajouter des primitives géométriques (ici primitives au sens d'objets 3D) à la surface d'un objet 3D d'origine. Cette technique a pour but de mapper des volumes 3D a la surface d'un autre en définissant un volume correspondant à une couche de "peau" contenant des volumes géométriques sur une surface d'origine. Néanmoins, cette méthode fonctionne dans un rendu par lancé de rayons et non en temps réel.

Enfin, sans être une technique de `Displacement Mapping`, La technique des `Texture Sprites`, décrite dans la référence [3], permet de projeter de petits éléments de textures dynamiquement sur une surface. La méthode propose une structure pour stocker la position et la direction de la projection et y accéder depuis les shaders. On peut, au lieu de projeter des textures, projeter des cartes de déplacements représentant un déplacement unitaire, mais dans le cadre de l'article les textures sont plus grandes que les triangle, or notre objectif est d'apporter des détails géométriques plus petits que les triangles.

1.3 Notre approche

Étant donner que notre objectif est d'évaluer des déplacements en temps réel, nous avons dans un premier temps tout évaluer en temps-réel, la distribution des primitives ainsi que leur évaluation. Après avoir rencontré quelques difficultés nous avons choisi de pré-calculer une distribution, la stocker dans la carte graphique et y accéder dans les shaders pour évaluer un déplacement. Cette seconde démarche permettrait de pouvoir appliquer plusieurs couches et aussi d'utiliser des distributions plus intéressantes que la première.

De plus, l'objectif est de faire apparaitre des détails géométriques plus petits que les faces sur la surface. Donc, pour obtenir une bonne résolution nous utilisons la tesselation matérielle pour "subdiviser la surface.

Notre approche consiste donc pour chaque sommet de la surface tessellé à :

— Évaluation de la distribution, quelle soit faite en temps réel ou qu'elle soit pré-calculée, et récupération de toutes les primitives qui auront une influence sur le sommet à déplacer.

— Évaluer le déplacement pour chaque primitive et effectuer un "mélange" de toutes les contributions. Ce mélange peut être un max (on ne garde que la contribution la plus grande), une moyenne, ou bien tout mélange imaginable. Pour évaluer le déplacement nous donnons en entrée une `height map` qui vas correspondre à un élément atomique de

déplacement.

— Évaluer la nouvelle normale induite par le déplacement calculer. Cette étape est importante pour par exemple calculer l'éclairage de l'objet modifié, mais aussi pour enchainer des déplacements successifs. Pour évaluer les normales on vas venir avoir en entrée

Dans le cadre d'un déplacement multi-couche, on va re-itérer les étapes présentées précédemment autant de fois qu'il y a de couches. De plus chaque couche sera appliquer en utilisant les normales évaluer sur la couche précédente.

Veuillez noter que, pour nous aider dans la visualisation de certain de nos résultats nous utilisons une technique de visualisation des arêtes (`wireframe`) décris dans les références [5] et [1].

Aussi, je souhaite redéfinir ou clarifier le sens des mots qui seront utilisés dans la suite de ce mémoire.

— Primitive : sera utiliser pour parler d'un élément distribuer sur l'objet 3D. Ne correspond plus aux primitives géométrique d'OpenGL (triangle, quads, ...).

— Patch : correspond a une primitive géométrique lors de la tessellation. Ce terme sera utiliser plus généralement, ainsi, on parlera de triangle ou de patch.

Enfin l'ensemble de cette méthode sera implémenté dans Gratin [5]. C'est un logiciel développé par l'INRIA qui permet d'implémenter à l'aide d'un système de noeuds programmables divers algorithmes ou méthodes de rendus en temps réel sur GPU.

5. http ://gratin.gforge.inria.fr/

2 La distribution

Afin de générer des détails géométriques sur un objet 3D, il nous faut d'abord déterminer une distribution de ces détails. On verra dans cette section, notre première approche d'évaluation de la distribution en temps réel (en même temps que l'évaluation du déplacement), puis notre seconde approche (qui sera retenue), qui consiste à pré-calculer une distribution et la transmettre à la carte graphique pour l'utiliser lors de l'évaluation du déplacement.

2.1 Évaluation en temps réel

Comme notre objectif est d'évaluer un déplacement des points de la surface en temps réel, nous avons d'abord essayé d'évaluer la distribution "à la volé" dans les `shaders`. Nous avons mit en place diverses approche pour cela. Mais dans tout les cas le procédé est le même. Pour chaque sommet a déplacé de la surface on vas chercher a évaluer explicitement tout les points de la distribution autour de lui.

Tout d'abord nous avons utilisé les coordonnées de texture pour venir répartir notre distribution. L'idée est de venir construire une grille 2D dans l'espace de texture en utilisant deux paramètre de densité, le nombre de carreaux en `x` et en `y`. Ceci nous permet de générer une grille régulière. L'idée de cette grille est que un point de la grille correspond a une primitive géométrique, ainsi la taille des primitive est directement déterminée par les densités en `x` et `y`. Cette solution est très simple en mettre en place à la condition que notre surface possède des coordonnées de textures. Néanmoins, cette solution possède une limite importante, en effet dans le cas (fréquent) ou les coordonnées de textures ne sont pas continues, alors on viendra créer des discontinuités sur la surface.

Ensuite nous avons utilisé directement une grille 3D définie dans l'espace objet (espace dans lequel sont définis les points de la surface). L'idée est équivalente a la solution précédente à la différence que l'on a plus besoins de coordonnées de textures, on utilisera directement les coordonnées 3D des sommets a déplacés. Aussi, sauf dans le cas de discontinuités de la surfaces, le problèmes de discontinuités qui existe pour la première approche n'existe pas dans ce cas. Néanmoins, le résultat avec cette approche est moins instinctif et plus lent, étant donné que une dimension s'ajoute.

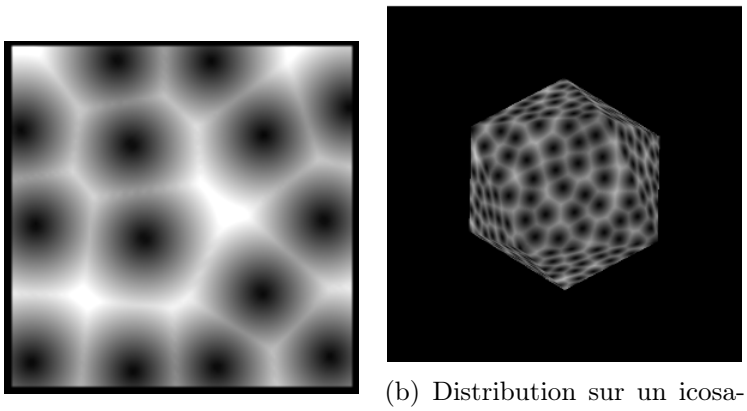
Dans le cas ou l'on veut obtenir une répartition plus "aléatoire", nous avons utiliser un générateur de nombre aléatoire pour venir déplacer chaque point de la grille afin d'obtenir par exemple une `jitterid grid`. Afin de garantir une cohérence entre les valeurs obtenues par ce générateur nous utilisons comme `seed` les coordonnées de la case de la grille, ainsi pour une mème case on viendra toujours obtenir les même valeurs. Ceci sera très important lors de l'évaluation. Néanmoins il est difficile d'obtenir des distribution plus complexes étant donné qu'il faut pouvoir, en tout point de notre espace donné, retrouver tout les points de la distribution autour de ce point, il nous faut donc pouvoir évaluer explicitement notre distribution, ce qui est réalisable seulement avec des distributions "simples".

2.2 Pré-calcul de la distribution

Nous avons donc fait le choix, de pré-calculer la distribution pour une surface. Ceci permet d'utiliser des distributions plus intéressante et complexes. Dans notre cas nous avons choisis une

distribution de `poison disk`. Nous avons choisis cette distribution car elle permet d'obtenir une répartition aléatoire tout en étant homogène, sans créer de zones avec de fortes densités d'échantillons et d'autres avec très peu d'échantillons.

L'objectif reste donc d'évaluer une distribution sur notre objet 3D. Pour ce faire nous avons utiliser un wrapper [6] de la `vcglib`[6] qui dispose des méthodes permettant de distribuer des échantillons sur un objet 3D suivant une distribution de `poisson disk`. Cependant ce wrapper a légèrement été modifier pour obtenir pour chaque échantillons de la distribution résultante le numéro de la face sur laquelle est l'échantillon. Nous pouvons observer sur la figure ci-dessous un plan et un icosaèdre sur lesquels nous avons évaluer une distribution, ce que nous observons est pour chaque sommet (après tessellation) est la distance (en nuances de gris) entre le sommet et la primitive la plus proche. On voit bien que la répartition est aléatoire et "naturelle", dans le sens où il n'y a pas d'amas de primitives ou bien de zones sans primitives.



(a) Distribution sur un plan.

(b) Distribution sur un icosaèdre.

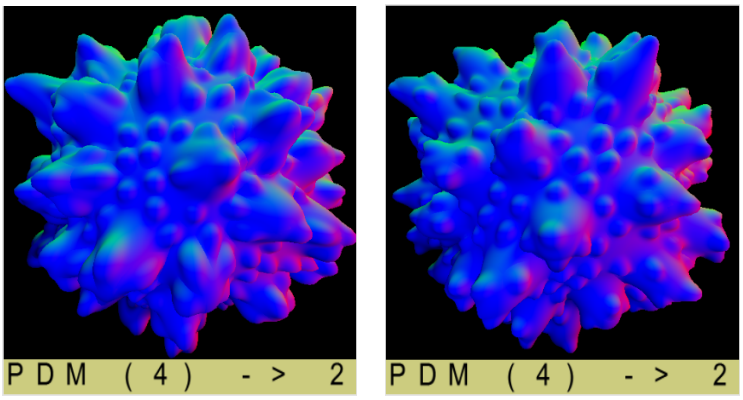
FIGURE 5 – Distribution sur une surface.

Quand on vient générer une distribution sur une surface on va donc obtenir l'ensemble des échantillons obtenus avec les informations suivantes pour chaque échantillon : leur position 3D dans le repère de l'objet, la normale de la surface a la position de l'échantillon, et le numéro de la face sur laquelle l'échantillon se trouve. Chacune de ces informations seront utiles pour la suite du processus. Le numéro de face sera essentielle lors de la construction de la structure de donnée qui sera utilisée pour stocké et utiliser la distribution dans les `shaders`. La position et la normales, elles seront utilisées pour évaluer la déformation de la surface dans les `shaders`. Aussi nous allons determiner un rayon, qui correspondra a la taille de la déformation relative à cet échantillon.

De plus, quand on viendra appliquer plusieurs couches de déplacements, on calculera une distribution par couche sur l'objet 3D. Ainsi pour obtenir des déformations plus agréable et justes on vas venir modifier la position et la normale des échantillons de chaque couche en utilisant toutes les couches précédentes. La méthode utilisée pour évaluer ces modification est la même que celle utilisée dans les `shaders` pour évaluer la déformation et les normales de la surface déformée. On peut observer la différence de résultat si on applique ou non cette

6. `http://vcg.isti.cnr.it/vcglib/`

modification sur les différentes distributions sur la figure 6. En effet sur la figure de gauche, les "bosses" de la seconde couche sont étirées sur les flancs des "bosses" de la première couche ; alors que sur la figure de droite, les "bosses" de la secondes couche conserves bien leur forme. A noter que les distributions ne sont pas les mêmes entre les deux figures.



(a) Sans modification de la distribution.

(b) Avec modification de la distribution.

FIGURE 6

2.3 Stockage de la distribution

Une fois la distribution calculée il nous faut la stocker et la faire parvenir a la carte graphique pour pouvoir évaluer les déformations. Pour ce faire nous avons choisis de stocker les échantillons par patchs (et donc par triangle de l'objet 3D). Ainsi pour chaque triangle on va stocker tout les échantillons qui contribue à ce triangle (tout les échantillons dont le rayon intersecte le triangle) ; ainsi pour chaque triangle on va maintenir à jour la liste des échantillon (leurs identifiants) qui contribuent à ce triangle.. Pour ce faire, on parcours les arrêtes du triangle et on vérifie deux chose. Si la distance entre le sommet à l'origine de l'arrête et l'échantillon est inférieur au rayon, alors l'échantillon contribue à toutes les faces autour du sommet, il faut donc les parcourir et mettre à jour la structure. Si la distance de l'échantillon à l'arrête (longueur du segment entre l'échantillon et le projeté orthogonal de l'échantillon sur l'arrête) est inférieur au rayon, on ajoute l'échantillon à la liste correspondant à la face de l'autre coté de l'arrête courante.

Une fois que nous avons, pour chaque face la liste de tout les échantillons contribuant à la face, on va pouvoir mettre à jours les `buffers OpengL` pour envoyer ces informations à la carte graphique. Dans notre cas on utilise 3 `buffers` (voir figure 7) :

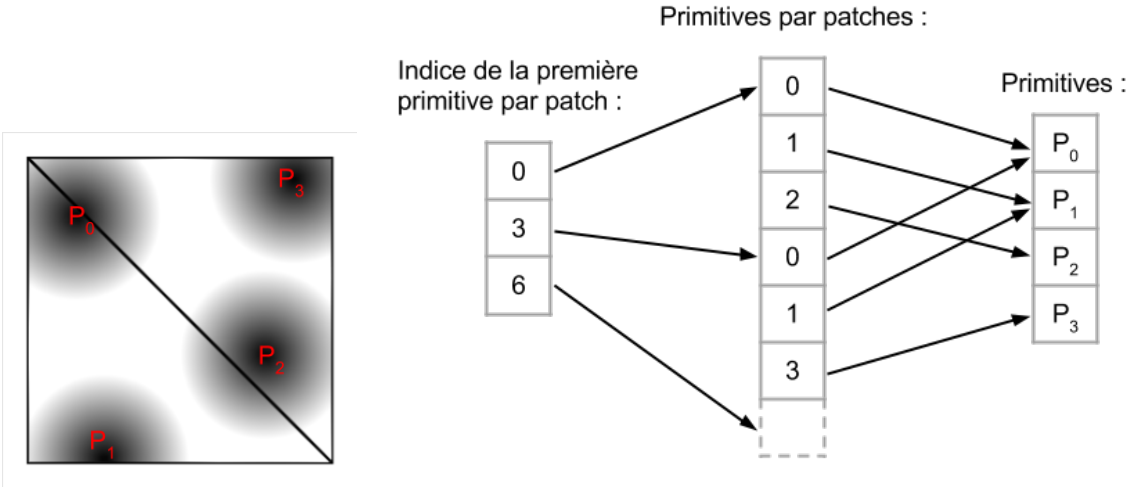
— Un `buffer` contenant l'indices ou trouver la première primitive par triangle.

— Un `buffer` contenant tout les indices des primitives par triangles organisé de la manière suivante : les indices pour la première face, puis pour la seconde et etc.

— Un `buffer` contenant toutes les primitives et leurs informations.

Dans le cas ou on viens distribuer plusieurs couches sur notre objet 3D, on viendra concaténer les différents `buffer`. Chaque `buffer` contiendra toujours les même informations

pour la premiere couche, puis pour la seconde et ainsi de suite. Il suffit juste d'utiliser le nombre de faces comme décalage pour venir chercher les primitives d'une face donnée pour une couche donnée.



(a) Distribution sur un plan.

(b) Structure de donnée.

FIGURE 7 – Stockage d'une distribution.

3 Évaluations des déplacements

3.1 Selon une distribution évaluée en temps réel

3.2 Selon une distribution pré-calculée

Dans le cas ou l'on utilise une distribution pré-calculée, l'évaluation dans le `tesselation evaluation shader` se réalise de la manière suivante.

Tout d'abord, on parcours les primitive contribuant au patch qui contient notre sommet a déplacer. Ce parcours se déroule de la manière suivante : On vient récupérer un indice de début et un indice de fin dans le `buffer` contenant les indices des premières primitives par patches en utilisant le numéro de patch accessible dans les `shader` à l'aide de la variable globale `gl_PrimitiveID`. On viens, grâce a ces indices, chercher les identifiants des primitives dans le patch et donc les informations de ces primitives. Ensuite, pour chaque primitive on viens évaluer la contribution de cette primitive sur le sommet. Pour ce faire, on vient projeter le sommet à déplacer dans le repère de la primitive qui est définis a l'aide de sa normale, du vecteur "u" et d'un vecteur "v" qui est le résultat du produit vectoriel entre la normale et "u". Puis, on transforme les composantes sur "u" et "v" pour les utiliser pour lire dans la texture qui vient définir le champs de hauteur de la primitive géométrique appropriée [7]. Enfin, grâce a la valeur obtenue, on peut appliquer notre fonction de mélange entre les différentes primitives. Dans notre cas, pour obtenir un résultat performant, il faut que la fonction de mélange puisse être calculer primitive par primitive. Une solution simple est de ne garder la primitive qui aura la contribution la plus grande ; ainsi on utilise simplement un max. Mais, il est possible d'imaginer d'autre fonctions de mélange.

Après cela le déplacement peut être appliquer au sommet le long de sa normale grâce à la valeur obtenue. Pour finir on vient modifier la normale a ce sommet pour obtenir la normale de la surface déformée.

3.3 Calcul des normales

Calculer les nouvelles normales de la surface déformé est un point important du processus pour deux raisons. La première est que si l'on veut utiliser le résultat dans un environnement 3D contenant des éclairage, la normale est nécessaire pour obtenir des résultats convenable. La seconde est que pour appliquer une couche de déformation nous déplaçons les sommets le long de la normale à la surface, et donc pour ajouter plusieurs couche nous devons évaluer la normale a la surface couche par couche pour que les déplacement se face une couche après l'autre. Ceci permet d'obtenir un déplacement vectoriel à l'aide de déplacement scalaires successifs.

Afin d'évaluer la nouvelle normale, on vient lire (lors de l'évaluation) dans une texture représentant le gradient de la primitive géométrique. A l'aide de ce gradient, on peut aisément calculer la normale dans le repère de la primitive (vecteur "u", vecteur "v" et normale de la primitive). Quand ceci est fait, on calcule la transformation entre cette normale et la normale de la primitive que l'on stocke dans un quaternion. En effet cette transformation est une rotation calculé à l'aide de l'angle entre les deux vecteurs et le vecteur résultant du produit vectoriel

7. On peut venir sélectionner entre diverses primitive d'entrée comme on le souhaite.

entre les deux vecteurs.Grace a cette transformation on peut venir appliquer la bonne rotation à la normale du sommet à déplacer.

Cette méthode permet donc de calculer les nouvelle normales pour tout les sommet de la surface. Ces normales seront utilisées pour enchainer les déplacements mais aussi en sortie du pipeline pour être utilisées pour calculer un éclairage ou autre. Néanmoins cette méthodes (calculer les normales par sommet) ne donne pas de bons résultats pour être utilisé en sortie du pipeline. En effet quand on vient propager ces normales vers la sortie du pipeline, on fait face a des problèmes d'interpolation (visibles sur la figure 8 dans la colonne du milieu). Afin de résoudre cela, nous avons choisis de calculer les normales par sommets (pour enchainer plusieurs couches de déplacement) et par fragment (un fragment correspond a un pixel ici). Pour éviter de devoir tout recalculer en double, on viens stocker les identifiant des primitives retenue pour évaluer le déplacement et les normales pour chaque sommet et chaque couche de déplacement. Ensuite, grâce au `geometry shader` on envoie au `fragment shader` les primitives retenues pour les trois sommet constituant le triangle contenant le fragment. Ainsi on a juste à déterminer laquelle des trois primitives (par niveau de déplacement) contribue au fragment et donc, d'évaluer la normale au fragment pour cette primitive. On obtien a la fin des normales plus précises que l'on peut visualiser sur la figure **??** (colonne de droite).

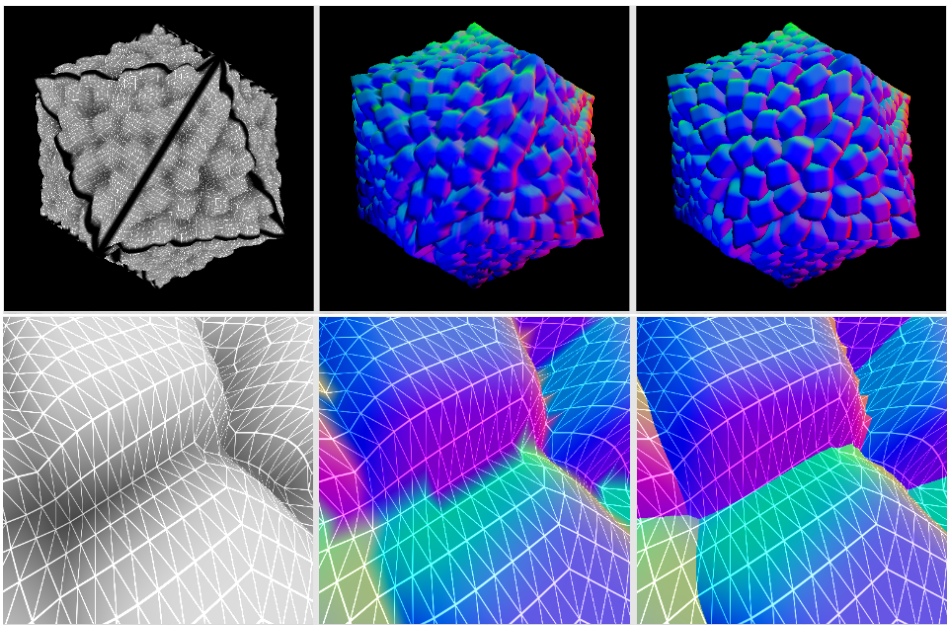


FIGURE 8 – Différences entre les normales par sommet et par fragment

3.4 Ajouts de plusieurs couches de déplacements

4 Résultats et évaluation

4.1 Résultats

A l'aide la méthode présenté dans ce rapport, nous pouvons obtenir les résultats présentées dans la figure 9. Cette méthode nous permet donc de générer des déformations géométrique sur une surface. On peut aussi générer plusieurs couches de déformations successive. Ces déformation sont contrôlées par une texture en entrée et peuvent être négative comme positives. Dans les exemples présents ci-dessous, les motifs de déformations sont les même sur une couche donné, pareil pour la taille de celles ci, néanmoins on pourrait très bien modifier les information des échantillons (normales, vecteurs u) pour chaque primitives individuellement et leur donner à chaque un motif différent. Ainsi on pourrait des résultats plus ou moins riches et intéressants. Aussi, la distribution peut être de la même nature pour toutes les couches, mais il serait possible d'utiliser différents types de distribution pour différentes couches, ou même effectuer des distributions locales (qui ne sont pas effectuées sur tout l'objet). De plus, les fonctions de mélanges entre les différentes primitives peuvent varier et donner des résultats encore différents ; dans les résultats présentés on utilise un simple maximum (de la valeur absolue du déplacement).

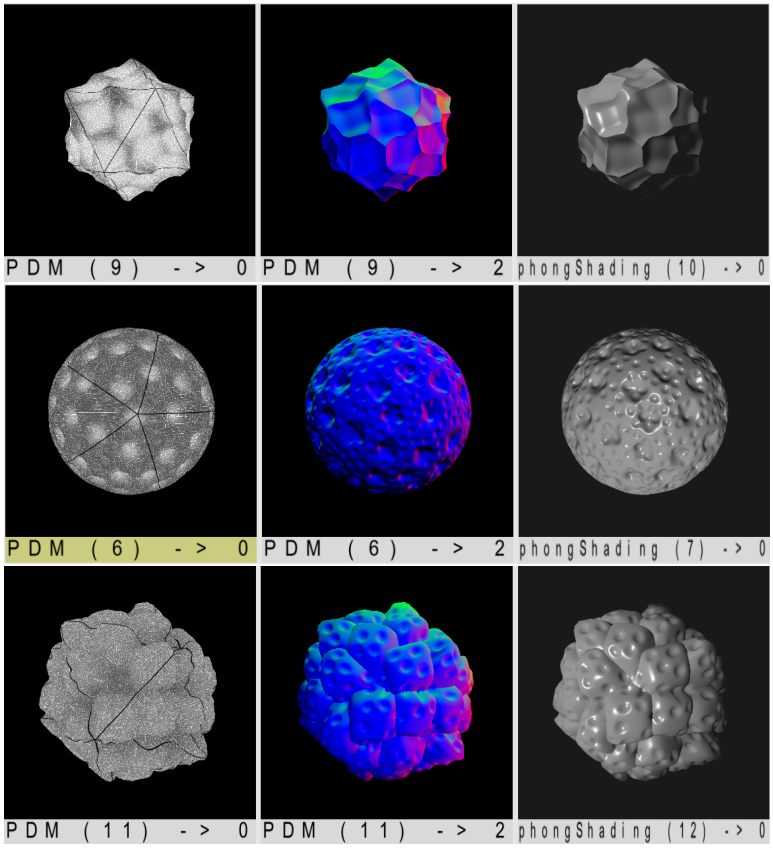


FIGURE 9 – Exemples de résultats.

La méthode présente malheureusement quelques limites. Tout d'abord lors du calcul et du stockage de la distribution, le fait de ne regarder que les faces voisines de la face courante d'un échantillon pose problème dans le cas ou le modèle présente des faces très fines. En effet, dans ce cas, un échantillon devrait contribuer a des faces plus lointaines que celles voisines, mais la méthode ne gère pas ces cas la. Ainsi on obtiens des discontinuités avec des troues dans la surfaces (visibles dans la figure 10).

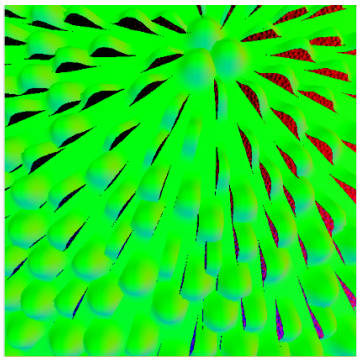


FIGURE 10 – Limites.

4.2 Améliorations

Les améliorations abordées dans cette partie sont des améliorations relatives aux performances de notre solution. L'utilisation de la tesselation et l'application de plusieurs couches de déplacement peuvent rapidement engendrer des problèmes de performances. Le principal problème est que, pour générer de la géométrie plus petite que nos triangles, il nous faut appliquer de la tesselation. Le niveaux de tessellation doit être d'autant plus élevé que la densité de primitive géométrique augmente ou que l'on ajoute des couches or si l'on a en entrée un mesh composé d'un grand nombre de face, le nombre de sommets générés lors de la tessellation et pour lesquels il faudra évaluer de potentiels déplacement devient rapidement très grand. Ainsi, pour des mesh comme la vache (environ 6000 faces), appliquer une couche de déplacement avec un niveau de tessellation permettant d'obtenir une résolution adéquate entraine des ralentissements et empêche d'interagir avec le rendu (faire bouger la caméra autour de l'objet) de manière fluide.

Ce problème a été résolu en mettant en place un `view frustum culling` et une stratégie de `LOD`[8]. A noter que ces deux méthodes sont complémentaires, en effet quand on vient s'éloigner de notre objet 3D, le `LOD` nous permets de maintenir l'apparence de l'objet en diminuant le nombre de sommet alors que si on se rapproche suffisamment, le `view frustum culling` vient éliminer des faces pour ne pas les afficher.

4.2.1 View Frustum Culling

Le `view frustum culling` consiste à n'afficher que des primitives présentes dans le `view frustum`, ce dernier étant la représentation de l'espace visible par la caméra.

8. Level Of Detail, niveaux de détails.

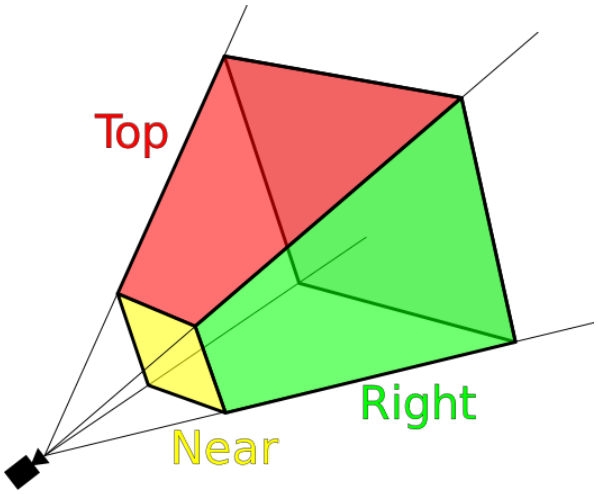


FIGURE 11 – Pyramide de vision.

Habituellement les objets pleinement à l'intérieur du `view frustum` seront affichés, les objets complètement à l'extérieur cachés et les objets à cheval sont découpés. Dans notre cas nous appliquons des règles plus simples. Pour les primitives à l'intérieure du `view frustum` et ceux à cheval nous appliquons un niveau de tessellation donné et pour ceux à l'extérieur on définit un niveau de tesselation égal à 1 pour ne pas créer de nouveaux sommet lors de la tesselation. On pourrait pour le cas des primitives à l'extérieur donner un niveau de tesselation de 0 ou de -1, ceci permettrait de faire disparaitre ces primitives et donc d'être plus performant, mais pour éviter de voir apparaitre ou disparaitre des primitives aux frontières du `view frustum` on préfère mettre une valeur de 1 pour conserver la primitive.

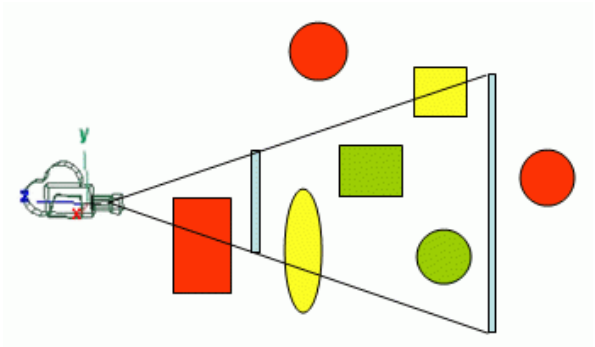


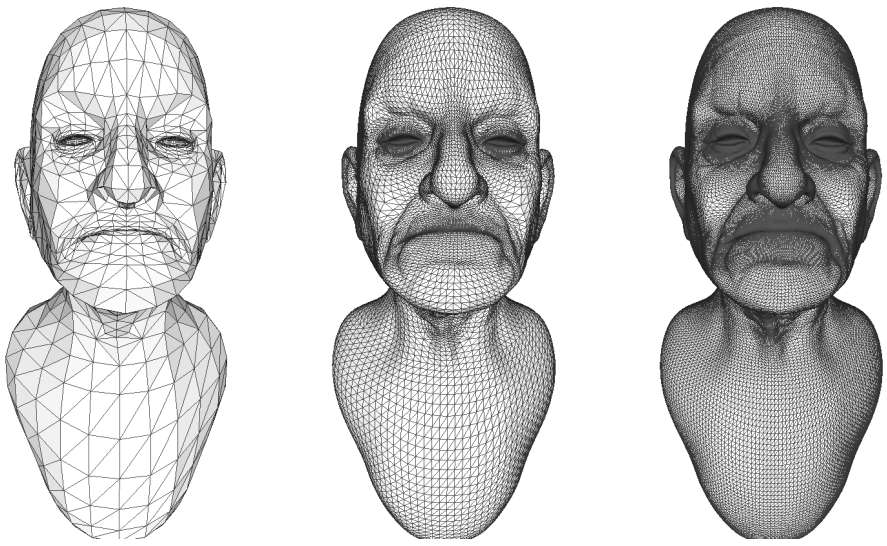
FIGURE 12 – Culling.

De plus, on va appliquer cette technique au déplacement en plus qu'à l'évaluation du niveau de tessellation. En effet, avant d'évaluer et donc d'appliquer un déplacement (ou plusieurs) à un sommet, on va venir voir si ce sommet est à l'intérieur ou proche du `view frustum`. Le principe est de vérifier si une sphère dont le centre est le sommet courant et le rayon la distance maximale au `view frustum` acceptée intersecte le `view frustum`. Si oui, alors on applique le déplacement, sinon on ne fait rien. On vient tester une sphère autour du sommet courant

pour éviter tout comportement étrange comme des discontinuités dans les déplacements aux frontières du `view frustum`.

4.2.2 Niveaux de détails

Le principe du `LOD`, ou niveaux de détails, est l'utilisation pour un même objet 3D de plusieurs modèles plus ou moins détaillés, selon la taille que cet objet a à l'écran. En effet, plus l'objet est petit à l'écran dû à un éloignement ou une transformation, on peut utiliser des modèles moins détailler qui vont conserver l'apparence de l'objet, mais en améliorant les performances en diminuant le nombre de sommets à traiter. On peut voir sur la figure ci-dessous plusieurs niveaux de détails pour un même objet 3D.



(a) Niveau 1. (b) Niveau 2. (c) Niveau 3.

FIGURE 13 – Niveaux de détails.

Dans notre cas nous n'allons pas utiliser différents modèles 3D, mais nous allons évaluer le niveau de tesselation dynamiquement selon la taille de l'objet à l'écran. Plus l'objet sera grand et donc visible, plus nous allons tesseller ses arêtes et ces faces pour percevoir les détails géométriques. Au contraire, plus l'objet sera loin et moins les variations géométriques visibles, plus on va diminuer le niveaux de tesselation et donc améliorer les performances. Pour ce faire nous allons déterminer le niveau de tessellation pour chaque arête en projetant la longueur de cette arête sur l'écran en nombre de pixel puis le diviser par un paramètre correspondante aux nombre de pixel par niveau de tessellation souhaité. Ceci permet de déterminer les niveaux de tessellation extérieurs (pour les arêtes), il suffit, par patch, de prendre la valeur maximale pour le niveau de tessellation intérieur. Pour projeter la longueur de l'arête sur l'écran on va projeter une sphère de centre le milieux de l'arête et de diamètre la longueur de l'arête sur l'écran. Cette solution a l'avantage d'être simple, mais ne prends pas en compte si la direction de la vue est rasante ou non sur l'arête. Néanmoins, cette solution reste simple et permet un

gain de performance significatif.

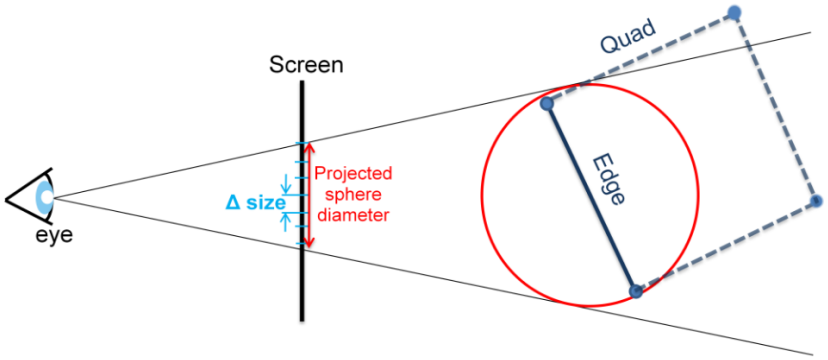


FIGURE 14 – Heuristique pour déterminer le niveau de tesselation.

L'ajout du niveau de détail a permis un grand gain de performances. Il nous permet d'obtenir des interactions fluides lors ce que les faces s'éloignent de la caméra. Néanmoins, lors du zoom ou d'un dé-zoom on peut observer des oscillations sur la silhouette de l'objet.

Conclusion

Conclusion, avec mises en parallèle des compétences acquise en stage vs M1. Ouverture sur les améliorations possibles et les possibilité “créatives” (avec des exemples comme : différentes primitives par layouts, déplacement positif/négatif, last normal layer, ...).

Références

[1] Andreas Bærentzen, Steen L. Nielsen, Mikkel Gjøl, Bent D. Larsen, and Niels Jørgen Christensen. Single-pass wireframe rendering. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[2] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, 28(3) :54–64, July 2009.

[3] Sylvain "Lefebvre, Samuel Hornus, and Fabrice" Neyret. "texture sprites : Texture elements splatted on surfaces". In *"ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)"*. "ACM SIGGRAPH", "ACM Press", "April" "2005".

[4] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. *ACM Trans. Graph.*, 24(3) :626–633, July 2005.

[5] Philip Rideout. Triangle Tessellation with OpenGL 4.0. `http://prideout.net/blog/?p=48`, `http://prideout.net/blog/?p=49`, Septembre 2010. [Online ; Dernier accès le 20-Juillet-2017].

[6] Rodolphe Vaillant. Source code for poisson disk sampling of a triangle mesh. `http://rodolphe-vaillant.fr/?e=37`, Octobre 2013. [Online ; Dernier accès le 20-Juillet-2017].