Projet de Fin d'Études

Subdivision de Surfaces en temps réel sur CPU et GPU

GUITTENY Fabrice
IDIART Baptiste
LE GOFF Erwan
PONSONNET Olivier

Table des matières

Table des figures

Résumé

Les surfaces de subdivision sont des représentations flexibles et efficaces de surfaces lisses, pouvant intégrer un contrôle de singularité (arêtes vives, semi-vives, etc...). Elles permettent d'obtenir à moindre coût des surfaces lisses par morceaux et de topologie arbitraire [7]. Cette représentation est aujourd'hui la plus utilisée dans l'industrie de l'animation et de la synthèse d'image (récemment dans les films Seigneur des Anneaux ou King Kong par exemple) et constitue un outil de modélisation très puissant dans les logiciels de modélisation (3DSMax, Maya, SoftImage, Blender, Lightwave,...). Certains travaux récents (moteur de particules [20], détection de collisions [23],...) ont mis l'accent sur leur implémentation dans les applications temps réel et, en particulier, sur le GPU[1]. On se propose ici de comparer les performances entre une implémentation de l'algorithme de subdivision de Catmull-Clark sur GPU et une implémentation de ce même algorithme sur CPU[2].

Mots-clés : surfaces de subdivision, GPU, shader GLSL, Catmull-Clark

[1]Graphical Processing Unit
[2]Central Processing Unit

Abstract

Subdivision surfaces are flexible and fast representaions for smooth surfaces. They even include singularities controls such as sharp or semi-sharp creases. With this kind of representation smooth surfaces with desired topology can be obtained at low cost [7]. Nowadays, subdivision surfaces are commonly used in animation industry and computer graphics (Lord of the rings or King Kong movies...). Actually, this is one of the most powerful modelling tools in 3D software (3DSMax, Maya, SoftImage, Blender, Lightwave,...). Recently, several researches (particle systems [20], collision detection [23],...) were carried out to compute subdivision surfaces in realtime, using the power of GPUs. So we'll try to compare a Catmull-Clark type subdivision algorithm computed on one hand on the CPU and, on the other hand, using the GPU.

Key words : subdivision surfaces, GPU, shader GLSL, Catmull-Clark

Remerciements

Tout d'abord, nous tenons à remercier particulièrement Tamy Boubekeur, membre de l'INRIA, de nous avoir accordé toute son attention pour nous aider tout au long de ce projet. Nous lui sommes également grés d'avoir répondu favorablement à notre demande.

Nous saluons Pascal Guitton, enseignant à l'Université Bordeaux 1, sans qui l'attribution du projet nous aurait été impossible. Nous lui sommes aussi reconnaissant pour ses conseils avisés.

Nous remercions également toutes les personnes qui nous ont guidées lors de nos formations et sans qui nous n'aurions pas pu réaliser ce projet.

Nous n'oublions pas non plus le personnel du Cremi et de l'Université de Bordeaux qui nous ont fourni les moyens de mener à bien ce projet mais aussi les personnes de nos entourages pour leur soutien et leurs encouragements.

Chapitre 1

Introduction au domaine d'application

1.1 Définitions de termes spécifiques à la 3D

Voici tout d'abord quelques termes simples mais néanmoins nécessaires pour une pleine compréhension de notre sujet.

– Sommet (vertex) : point de l'espace 3D défini par ses coordonnées x, y et z.

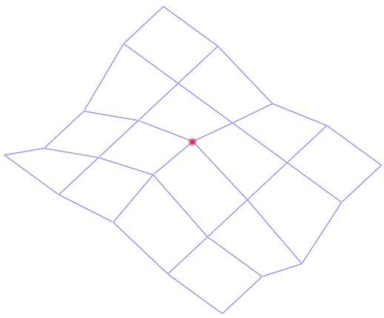

FIG. 1.1 – Sommet

– Arête (edge) : ligne reliant deux sommets consécutifs d'un polygone.

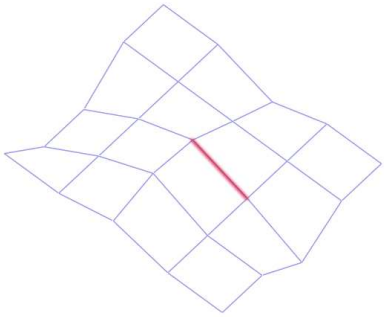

FIG. 1.2 – Arête

– Face ou polygone : surface créée en reliant plusieurs sommets.

– Quad : face composée de quatre arêtes et donc de quatre sommets.

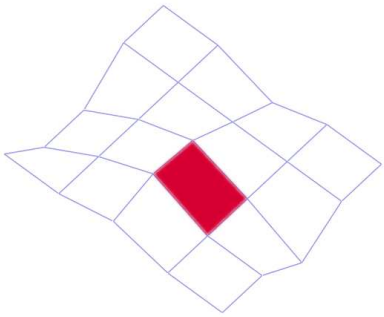

FIG. 1.3 – Quad

– Valence : caractéristique d'un point référant au nombre de points reliés à ce dernier. C'est donc le nombre d'arêtes issues de ce point.

– Centroïde : Point situé à la moyenne des coordonnées des sommets de la face.

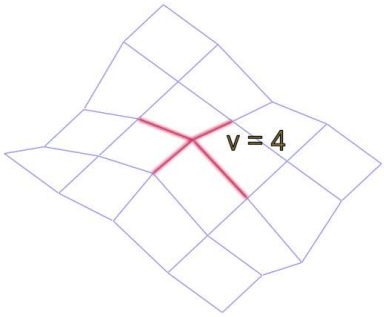

FIG. 1.4 – Valence

1.2 *Shaders*

Les *shaders* permettent d'améliorer considérablement la qualité de rendu. Ce sont de petits programmes executés par la carte graphique. Ils étaient autrefois écrits en pseudo assembleur et aujourd'hui en pseudo C. Ils permettent de reprogrammer certaines parties du *pipeline*[1] de rendu.

Il existe deux types de *shaders* :

- les *vertex shaders* [2] ou *vertex program*[3] : ils sont executés pour chaque sommet du maillage 3D et remplacent la partie transformation et éclairage du *pipeline* de rendu.
- les *pixel shaders*[2] ou *fragment program*[3] : ils sont executés pour chaque pixel à afficher et remplacent la partie filtrage, texture et mélange du *pipeline* de rendu.

1.2.1 Le *pipeline* de rendu

Les différentes étapes du rendu

1. **Tessellation**
   C'est la transformation des surfaces courbes en surfaces triangulées exploitables par le GPU[4].
2. **Transformation (vertex *pipeline*)**
   Cela consiste à placer les objets dans la scène 3D. Des calculs matriciels effectuent les opérations de translation, rotation et mise à l'échelle. Durant cette phase, aucun

[1] technique de conception des processeurs où l'exécution de plusieurs instructions se chevauchent à l'intérieur même du processeur

[2] nom employé pour l'API DirectX

[3] nom employé pour l'API OpenGL

[4] Graphic Processor Unit : processeur de la carte graphique

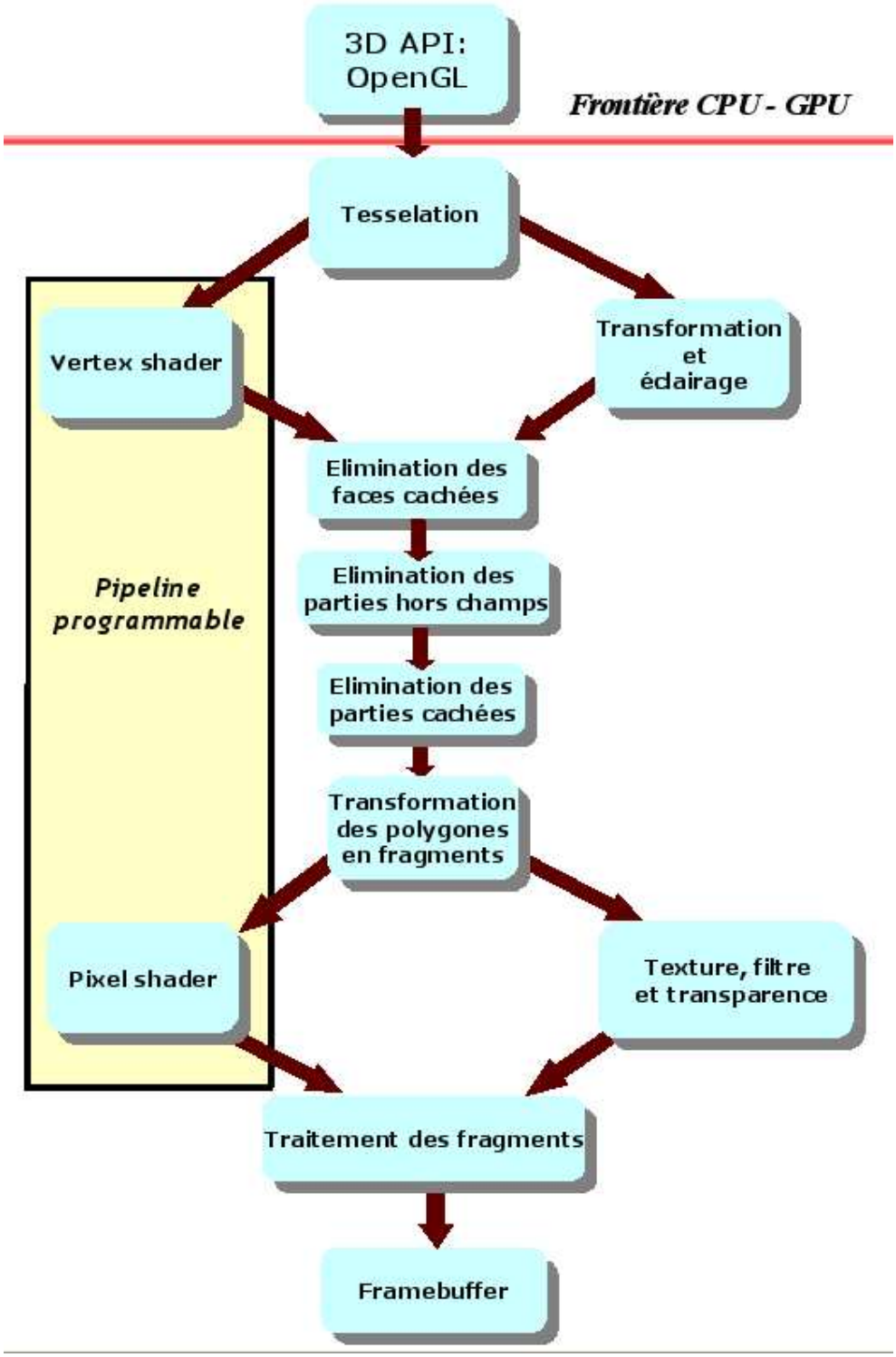

FIG. 1.5 – Schéma d'un pipeline graphique [29]

polygone n'est généré. En cas d'agrandissement d'un objet, celui-ci sera défini par le même nombre de polygones.

3. **Eclairement (vertex *pipeline*)**
   Effectue les calculs d'éclairement des vertices en fonction, entres autres, de la position de lumière, de la couleur et du matériau de l'objet auquel appartient le vertex traité. Cette étape demande beaucoup de calculs à la carte graphique.
4. **Elimination des faces cachées**
   Cette étape supprime les faces orientées dans le même sens que la caméra.
5. **Elimination des parties cachées**
   Certains polygones peuvent être cachés par d'autres polygones. Ces polygones, devenant de ce fait invisibles à l'écran, sont éliminés du rendu.
6. **Elimination des parties hors champ**
   On supprime de la scène les morceaux d'objets se situant en dehors du champ de vision.
7. **Transformation des triangles en fragments**[2]
8. **Texturage, filtrage et mélange (pixel *pipeline*)**
   Chaque fragment a pour couleur le mélange d'un texel[3] par texture en prenant en compte l'éclairage par interpolation des vertices.
9. **Tests sur les fragments**
   La couleur du fragment et sa profondeur sont déjà calculées. Les composantes RGBA[4] sont stockées dans le tampon de couleurs et la profondeur dans le tampon de profondeur. Ces tampons, stockés dans le *framebuffer*, permettront de constituer l'image à afficher. La manière dont seront utilisés ces tampons est définie par l'unité de rastérisation de la carte graphique. Contrairement aux pixel et vertex *pipelines*, l'unité de rastérisation n'est pas programmable. Elle effectue les opérations suivantes :
   - **Le test alpha** rejette des fragments suivant le résultat de la comparaison entre la valeur alpha d'un fragment arrivant et une valeur constante définie.
   - **Le test du stencil** élimine les fragments en fonction des valeurs du *stencil buffer*[5]
   - **Le test de visibilité** vérifie pour chaque fragment si il est masqué par un autre fragment. Si c'est le cas, la nouvelle couleur du fragment est calculée en fonction de la transparence.

[2]pixel contenant des informations supplémentaires comme la profondeur
[3]plus petite partie d'une texture
[4]Red Green Blue Alpha
[5]stencil buffer : matrice longueur x hauteur de l'écran

1.2.2 Le *vertex shader*

Le *vertex shader* (cf. figure 1.6) est un programme traitant les sommets du maillage. Il remplace la partie transformation et éclairage du *pipeline* de rendu.
Il existe deux types de paramètres pour un *vertex shader* :

- Les paramètres variables : les positions, couleur, normales et coordonnées de texture des sommets à traiter.
- Les paramètres constants : les matrices de transformations, la position et les caractéristiques de la lumière. Les matrices de transformations représentent les opérations géométriques à appliquer aux sommets pour les projeter en 2D sur l'écran.

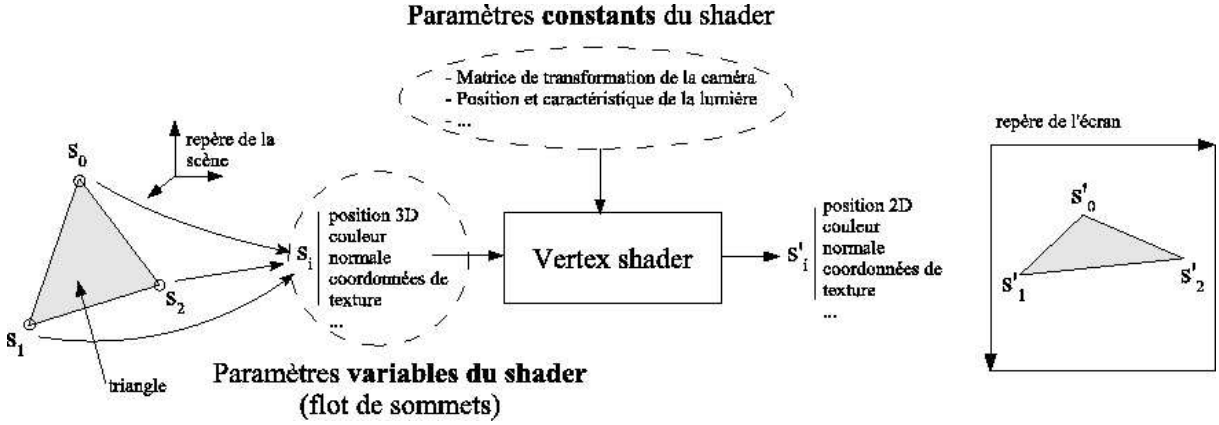

FIG. 1.6 – Execution d'un *vertex shader* [12]

1.2.3 Le *pixel shader*

Le *pixel shader* (cf. figure 1.7) est un programme executé pour chaque pixel d'un triangle avant d'être affiché. Les sommets formant le triangle ont été transformés auparavant par le *vertex shader*.
Il existe deux types de paramètres pour un *pixel shader* :

- Les paramètres variables : ce sont des interpolations des données des sommets. Cette interpolation est effectuée par la carte graphique.
- Les paramètres constants : les textures, la position et les caractéristiques de lumière.

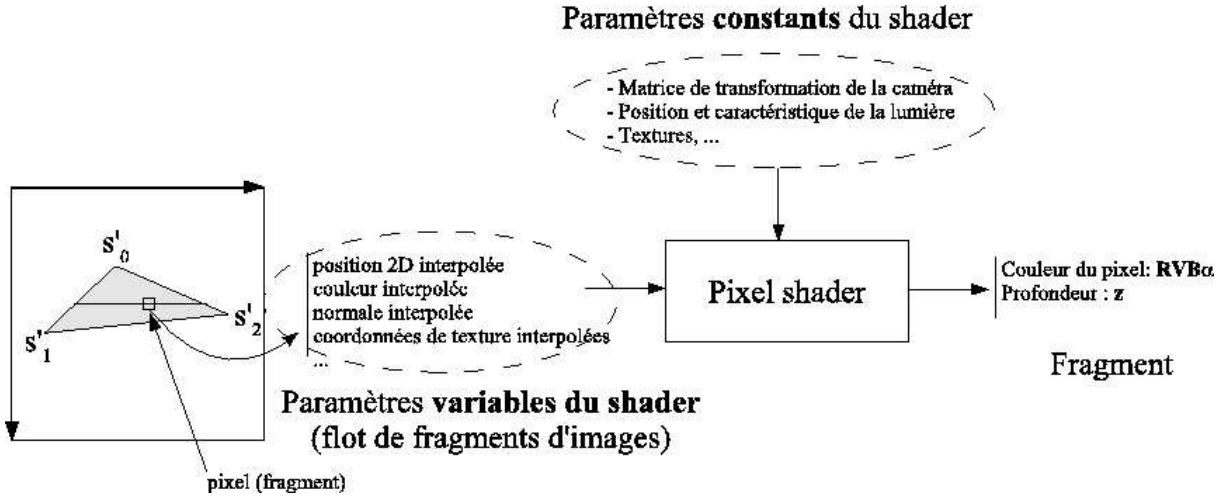

FIG. 1.7 – Execution d'un *pixel shader* [11]

1.3 General-Purpose Computing on Graphics Processing Units

Les GPU sont des processeurs dédiés au graphisme. Cependant, de nombreux projets sont actuellement en cours afin de les détourner de leur usage. Le GPGPU [6] consiste à utiliser le GPU pour des traitements génériques.

1.3.1 La différence fondamentale CPU / GPU : le parallélisme

Les CPU éprouvent les pires difficultés à effectuer des calculs parallèles. A tel point que se développent les multi-core qui combinent plusieurs processeurs indépendants sur une unique puce de silicone. En revanche un GPU est adapté à ce type de calculs. La raison est simple : le GPU et le CPU ne traitent pas les mêmes données. Les GPU ne traitent que des vertices et des pixels indépendants les uns des autres. Un *pixel pipeline* (resp. *vertex unit*) traite un pixel (resp. vertex). Les pixels (resp. vertices) étant indépendants, plusieurs *pixel pipeline* (resp. *vertex units*) permettent le traitement de plusieurs pixels (resp.vertices) simultanément. De nos jours, les cartes graphiques possèdent environs 32 *pixels pipelines*.

[6] General-Purpose Computing on Graphics Processing Units [16]

1.3.2 Pourquoi utiliser le GPU ?

Les processeurs graphiques sont plus rapides. A titre d'exemple, un processeur Pentium 4 dual-core effectue 24.6 gflops[7] alors qu'un processeur GeForce Fx 7800 réalise 165 gflops [16].

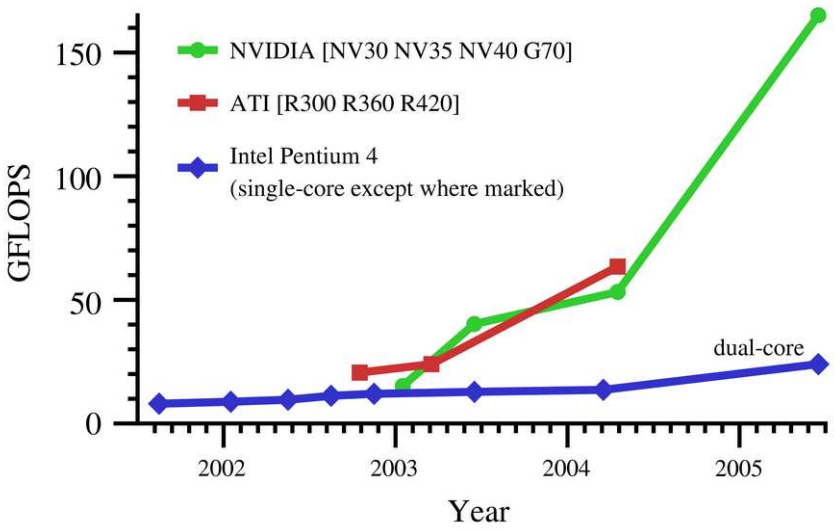

FIG. 1.8 – Comparatif de l'évolution de la puissance des GPU et CPU [16]

Cette tendance devrait se confirmer. En effet, le marché du jeu vidéo génère des budgets colossaux qui favorisent l'innovation.
Les GPU sont spécialisés dans la manipulation de nombres à virgules flottantes.
Enfin, le récent développement de langages de shader (GLSL, HLSL, cg) rendent accessibles la programmation des GPU. Toutefois, le potentiel énorme des GPU n'est que faiblement utilisé par les applications actuelles. Ces dernières étant majoritairement exécutées sur des CPU, elles ont été développées dans un esprit de séquentialité. Exécuter des applications sur GPU revient donc à repenser entièrement les algorithmes et non pas simplement à importer du code exécuté sur CPU (cf. tab 1.9).

[7]Giga FLoating point OPérations per Second : Unité de mesure de la vitesse de calcul d'une machine en milliard d'opérations en virgule flottante par seconde

CPU	GPU
Tabeau 1D	Texture 1D
Tableau 2D	Texture 2D
Tableau 3D	Texture 3D
Traitement de chaque élément du tableau	*Pixel shader* appliqué à chaque texel de la texture.

FIG. 1.9 – Analogie CPU-GPU

1.4 Subdivision de surfaces

1.4.1 Introduction

Le principe général de la subdivision de surfaces est d'affiner un objet 3D défini par un maillage initial plus ou moins grossier en subdivisant récursivement ce maillage afin d'obtenir un objet plus réaliste et donc plus agréable visuellement (cf. figure 1.10). Ce réalisme est en grande partie obtenu par le fait que la subdivision a un effet de lissage sur l'objet ce qui rend les arêtes moins vives et fait donc moins apparaître les polygones.
Ce résultat de lissage peut être également obtenu avec d'autres méthodes de modélisation comme les surfaces implicites équipotentielles ou les surfaces paramétriques (NURBS [8] notamment) avec lesquelles elles sont souvent comparées mais qui sont plus difficiles à manipuler.
Les surfaces de subdivisions ont été utilisées dans l'industrie pour la première fois en 1997 dans le court métrage Geri's Game de Pixar et leur utilisation ne cesse d'augmenter depuis. Elles sont, aujourd'hui, intégrées dans la plupart des outils de modélisation, d'animation et de rendu.

[8] Non Uniform Rational B-Splines

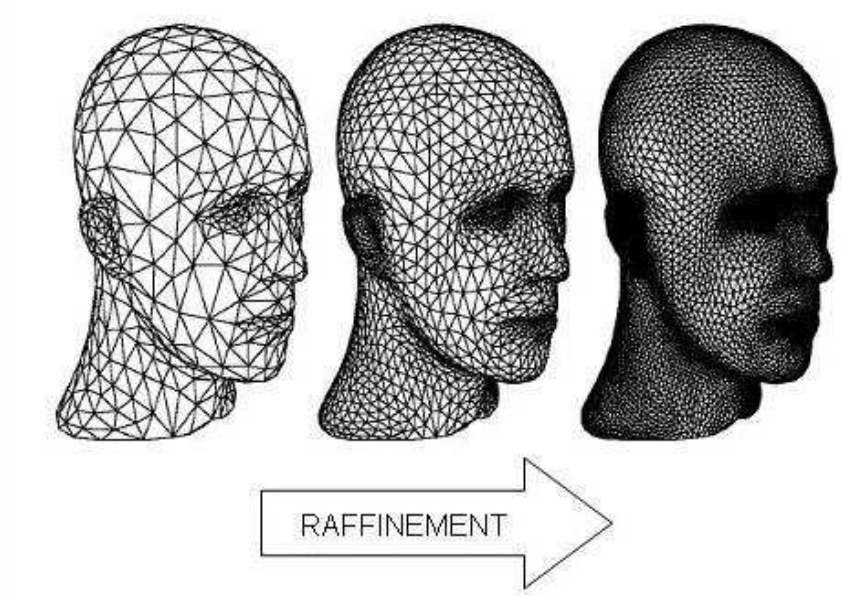

FIG. 1.10 – Subdivision par l'algorithme de Loop

1.4.2 Intérêts

Les techniques de subdivision de surfaces sont très employées dans l'industrie notamment car elles combinent les avantages de la représentation polygonale à savoir des topologies quelconques et la rapidité du rendu matériel (les GPU sont optimisés pour afficher des polygones) mais aussi ceux des surfaces splines (parmi lesquelles les NURBS) qui permettent d'obtenir des objets lissés.

De plus, il ne suffit que de créer un modèle basse résolution de l'objet (avec des outils de modélisation polygonale classiques) comme maillage de départ : le modèle haute résolution étant généré après des subdivisions successives. Ainsi, on obtient à moindre coût des images de haute qualité mais également des applications temps réel d'un plus grand réalisme. En effet, l'animation d'un objet se fait aisément : on anime le maillage de base qui comporte peu de polygones et la subdivision permet d'obtenir le maillage raffiné animé. La subdivision de surfaces a donc pour avantages d'être simple, à faible coût, explicite mais elle peut être également adaptative (pour certains schémas de subdivision) : on ne raffine qu'une partie du maillage afin de ne faire apparaître que certains détails.

1.4.3 Inconvénients

Comme toute méthode, la subdivision de surfaces présente certains désavantages. Un des plus importants est la difficulté d'obtenir des applications temps réel sur des objets comportant plusieurs milliers de polygones après 4 ou 5 subdivisions. En effet, pour du temps réel, le taux de rafraîchissement des images est crucial et il faut donc garder un taux de FPS [9] aussi élevé que possible. On estime à environ 10 FPS la limite pour un nombre significatif de polygones. Ceci pose donc problème quand un objet est trop détaillé : la liaison entre le CPU et le GPU se comporte alors comme un goulot d'étranglement puisque le CPU doit subdiviser à chaque image le maillage de contrôle et envoyer au GPU énormément d'informations liées au maillage raffiné (position des points, couleur des points, polygones...). L'envoi et le traitement de ces informations peuvent alors exiger un temps de calcul et d'exécution trop élevé pour garder un taux de FPS raisonnable. L'application est donc saccadée et le critère de temps réel n'est pas respecté. Afin de palier à ce problème, il existe des techniques comme celle que nous avons implémentée qui permettent de décharger le CPU et d'effectuer la subdivision directement au niveau du GPU qui est, rappelons-le, optimisé pour effectuer des calculs sur les polygones. Un autre inconvénient concerne le lissage des objets. Pour certaines applications comme la conception de pièces industrielles par exemple, il est nécessaire de raffiner un objet sans pour autant le rendre entièrement lisse : il faut garder les arêtes vives qui lui sont caractéristiques. Dans ce cas, il existe des méthodes de subdivision avec des masques spécifiques qui s'appliquent à de telles arêtes préalablement marquées comme vives par l'utilisateur. Bien entendu, cela complexifie les calculs et les rend donc plus longs.

1.4.4 Rappels sur les maillages polygonaux

Avant de nous pencher sur les différentes techniques de subdivision de surfaces, rappelons tout d'abord quelques notions sur les maillages polygonaux qui nous seront utiles lors de la description de ces méthodes. Tout d'abord, il existe différents types de maillages suivant le nombre de côtés des polygones utilisés mais les plus répandus sont les maillages en quadrilatère et les maillages triangulaires. Les points de ces maillages, appelés points de contrôle, peuvent être soit ordinaires soit extraordinaires : cette distinction dépend de leur valence. Ainsi, les points réguliers intérieurs (resp. du bord) au maillage ont une valence de 6 (resp. 4) dans un maillage triangulaire et de 4 (resp. 3) dans un maillage quadrangulaire. Les points singuliers sont tous ceux qui ne respectent pas ces règles (cf. figure 1.11).

Rappelons également que subdiviser un maillage ne crée que des points réguliers mais ne permet pas de supprimer de point irrégulier.

[9] Frames Per Second

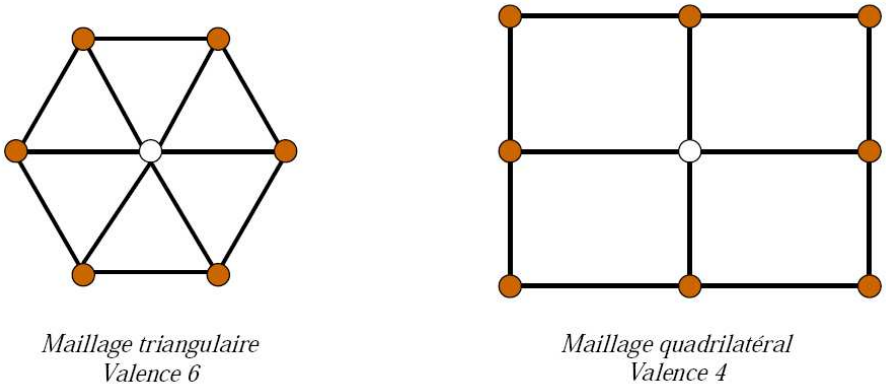

FIG. 1.11 – Exemples de maillages réguliers

1.4.5 Principe

La subdivision de surfaces se base sur le principe de subdivision de courbes : on raffine une courbe anguleuse jusqu'à tendre vers une courbe lisse appelée courbe limite (ie limite d'une séquence de subdivisions successives). Le principe de la subdivision de surfaces est d'appliquer une technique similaire sur des surfaces anguleuses : plus on subdivise, plus le lissage est important et donc plus les angles s'estompent. A chaque itération n, on crée donc de nouveaux points qui sont exprimés comme une combinaison linéaire des points existants à l'itération précédente $n-1$.

1.4.6 Subdivision de courbes : algorithme de Chaikin (1973)

L'algorithme de Chaikin [9] permet de subdiviser récursivement une courbe de la manière décrite ci-dessous.

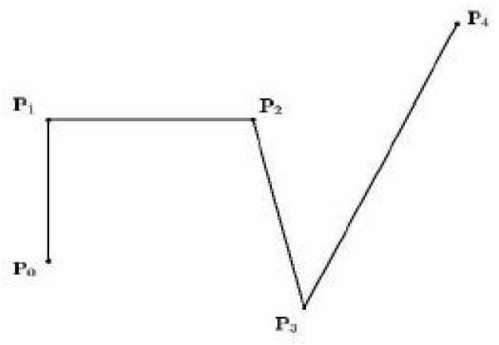

FIG. 1.12 – Algorithme de Chaikin : courbe initiale [extrait de [18]]

Sur la courbe initiale (cf. figure 1.12), les angles sont très marqués. Afin de lisser cette courbe, on crée de nouveaux points de la manière suivante : pour chaque "arête" de longueur d de la courbe, on crée les points à distance $\frac{d}{4}$ des extrémités. On crée ensuite des arêtes entre ces points ce qui permet d'augmenter le lissage puisque les angles entre chaque nouvelle arête sont moins marqués (cf. figure 1.13).

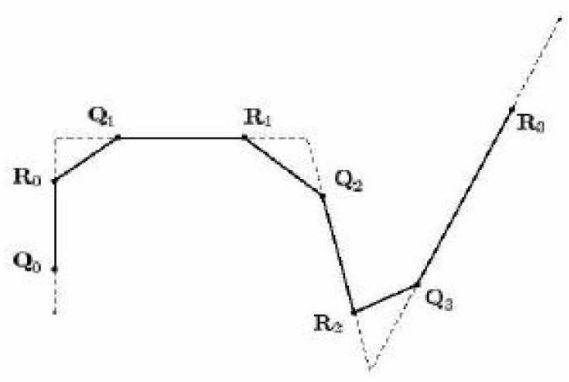

FIG. 1.13 – Algorithme de Chaikin : première subdivision [extrait de [18]]

Le procédé est alors réitéré jusqu'à masquer au maximum les angles très marqués initialement et à obtenir une courbe lisse (cf. figure 1.14).

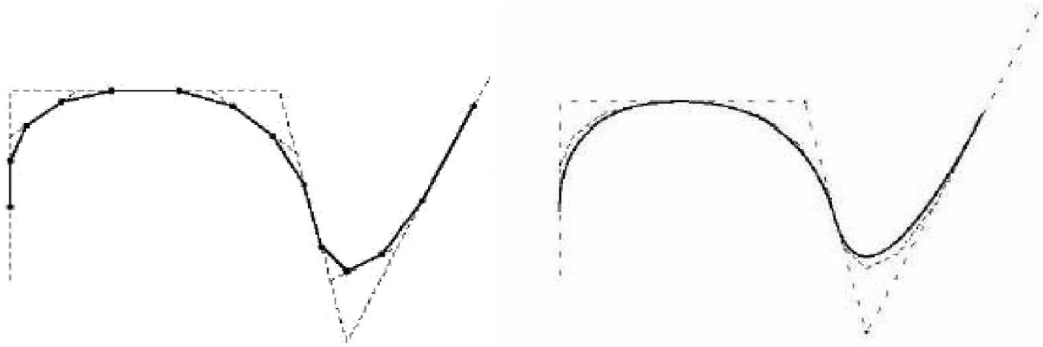

FIG. 1.14 – Algorithme de Chaikin : les subdivisions successives aboutissent à une courbe lisse [extrait de [18]]

1.4.7 Les différents types de subdivisions de surfaces

Avant de décrire les principales méthodes de subdivision de surfaces et notamment celle de Catmull-Clark que nous avons implémentée, rappelons qu'il existe deux types de schémas de subdivision :

- ***Schéma d'interpolation :***

  La surface (resp. courbe) limite passe par les points initiaux. Ces derniers ne sont donc jamais modifiés par le processus : les sommets du maillage initial et chaque nouveau sommet obtenu par subdivision appartiennent à la surface limite.

- ***Schéma d'approximation :***

  À chaque subdivision, la position des anciens sommets est modifiée et les nouveaux points sont placés pour converger vers la surface limite (sans l'atteindre).

De même, on dit d'une subdivision qu'elle est **primale** si elle est basée face : les faces initiales sont donc divisées en plusieurs faces par la création d'un sommet entre les sommets d'une face. A contrario, on dit qu'elle est **duale** si elle est basée sommet : les points initiaux sont divisés en plusieurs points.

Il existe, par conséquent, plusieurs schémas de subdivision qui se démarquent les uns des autres par leurs propriétés respectives comme le type de maillage sur lequel elle s'appliquent, l'interpolation ou l'approximation de ce maillage, la continuité de la surface générée, le réalisme du rendu, le temps de calcul ou bien l'adaptativité de la subdivision. Nous allons donc, dans cette partie, décrire de manière non exhaustive quelques méthodes de subdivision parmi les plus connues et les plus utilisées : Loop [24], Butterfly [14], Kobbelt [21], Doo-Sabin [13], $\sqrt{3}$ [22] et bien entendu Catmull-Clark [8].

Subdivision de Loop

C'est un schéma approximant et primal qui fut proposé en 1987 par Charles Loop [24]. Ces avantages principaux sont qu'il est juste, qu'il converge rapidement et qu'il s'applique à des maillages triangulaires ce qui en fait le plus utilisé actuellement (les GPU sont optimisés pour afficher des polygones et notamment des triangles). Il permet d'assurer une continuité C^2 pour les points réguliers, C^1 pour les autres.

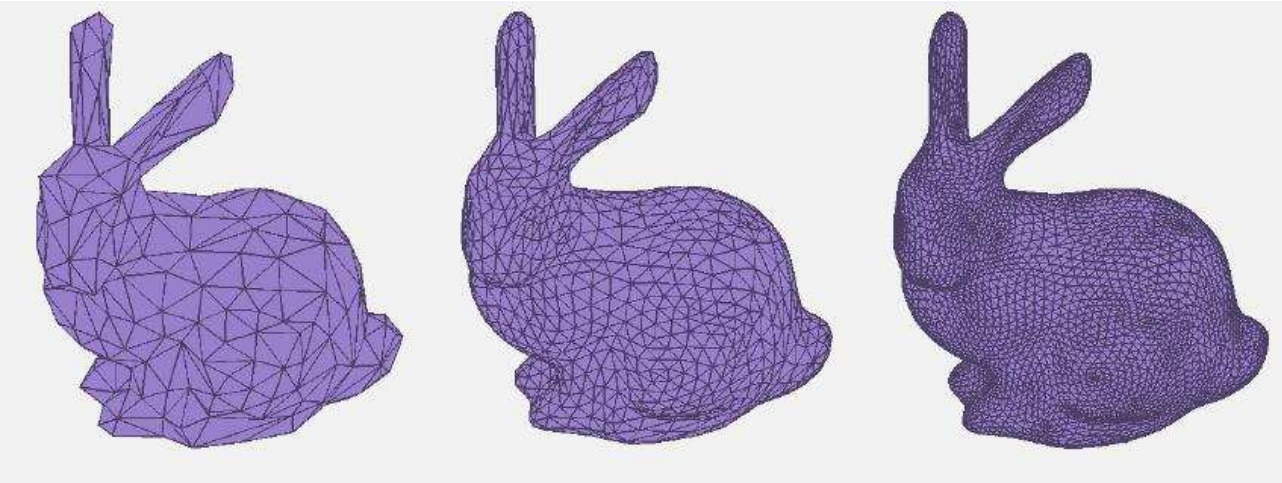

FIG. 1.15 – Exemple de subdivision par l'algorithme de Loop

Le principe est le suivant :

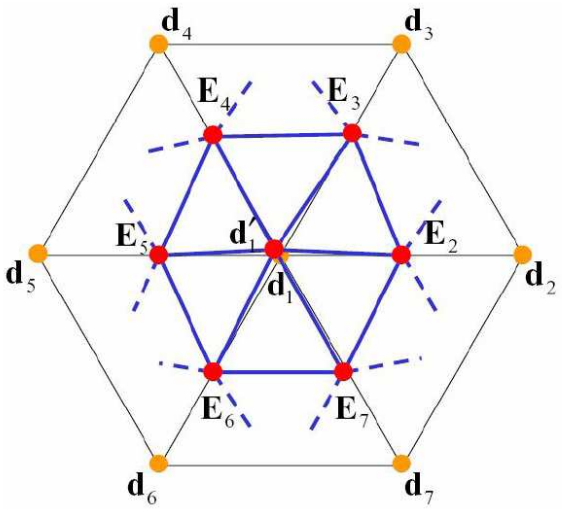

FIG. 1.16 – Schéma de subdivision de Loop (maillage régulier) [extrait de [18]]

Les points oranges représentent les points du maillage initial et les points rouges ceux du maillage subdivisé (ceci est le cas également pour les masques des autres schémas de subdivision présentés par la suite).

Les E_i sont les nouveaux points et d'_1 le sommet d_1 après repositionnement.

$$E_i = \frac{3}{8}(d_1 + d_i) + \frac{1}{8}(d_{i-1} + d_{i+1})$$

Les points d_i sont repositionnés selon un coefficient α_n tel que :

$$\alpha_n = \frac{3}{8} + \left(\frac{3}{8} + \frac{1}{4} \cdot cos\left(\frac{2\pi}{n}\right)\right)^2$$

n désignant la valence du sommet d_i.

Ensuite, la nouvelle position du sommet d_1 est obtenue par l'équation :

$$d'_1 = \alpha_n \cdot d_1 + \frac{1-\alpha_n}{n} \cdot \Sigma_{j=2}^{n+1} d_j$$

Pour les sommets du bord, on applique un masque spécifique :

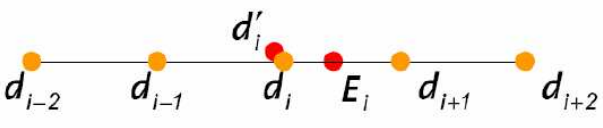

FIG. 1.17 – Schéma de subdivision de Loop (sommets du bords) [extrait de [18]]

Le point E_i est le milieu de l'arête $[d_i, d_{i+1}]$: $E_i = \frac{1}{2}(d_i + d_{i+1})$

Le point d'_i est alors obtenu grâce à l'équation : $d'_i = \frac{3}{4}d_i + \frac{1}{8}(d_i + d_{i+1})$

Subdivision de Butterfly

C'est un schéma interpolant et primal [14] qui s'applique à des maillages triangulaires et assure une continuité C^1 pour les points réguliers, C^0 pour les autres. Il est moins juste qu'un schéma approximant et peut générer des torsions. Le fait qu'il soit interpolant implique que la figure obtenue reste proche de la forme du polygone initial car les sommets du maillage d'origine ne sont pas déplacés.

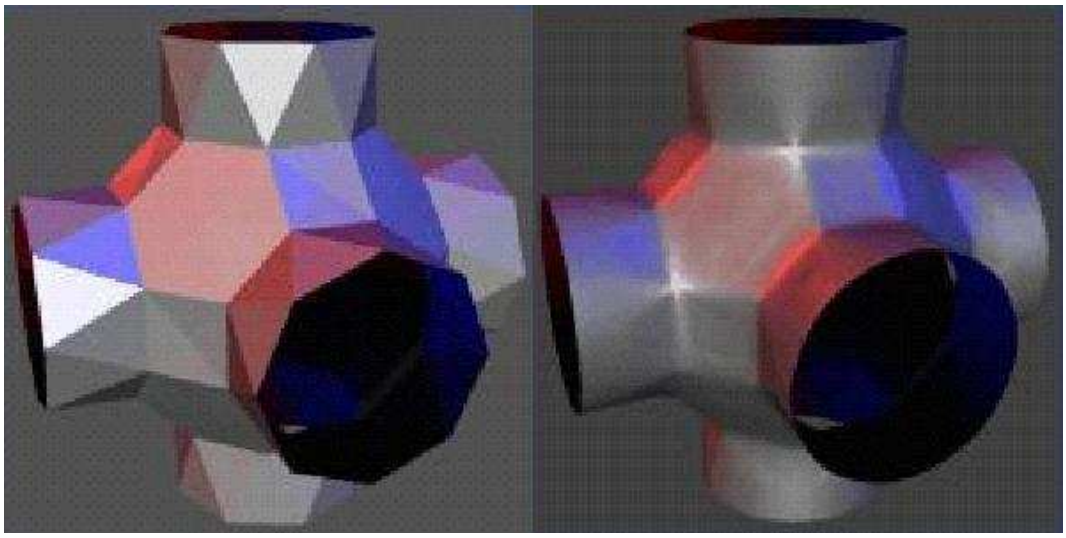

FIG. 1.18 – Exemple de subdivision par l'algorithme de Butterfly

Le principe est le suivant :

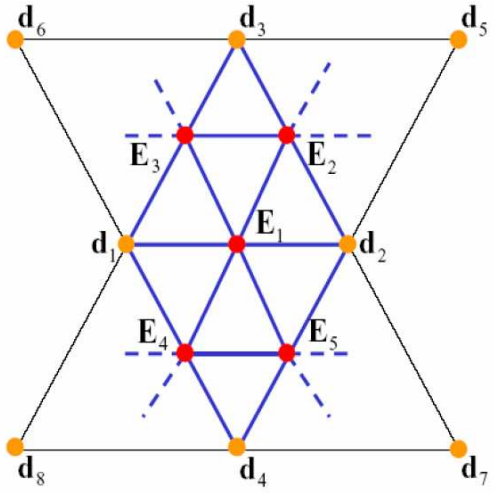

FIG. 1.19 – Schéma de subdivision de Butterfly (maillage régulier) [extrait de [18]]

Les points du maillage subdivisé sont les points E_i et les points d_i du maillage initial.

Les points E_i sont calculés de la façon suivante :

$$E_i = \frac{1}{2}(d_1 + d_2) + \frac{1}{3}(d_3 + d_4) + \frac{1}{16}(d_5 + d_6 + d_7 + d_8)$$

Pour les sommets du bord ou pour un des sommets irréguliers, on applique des masques spécifiques :

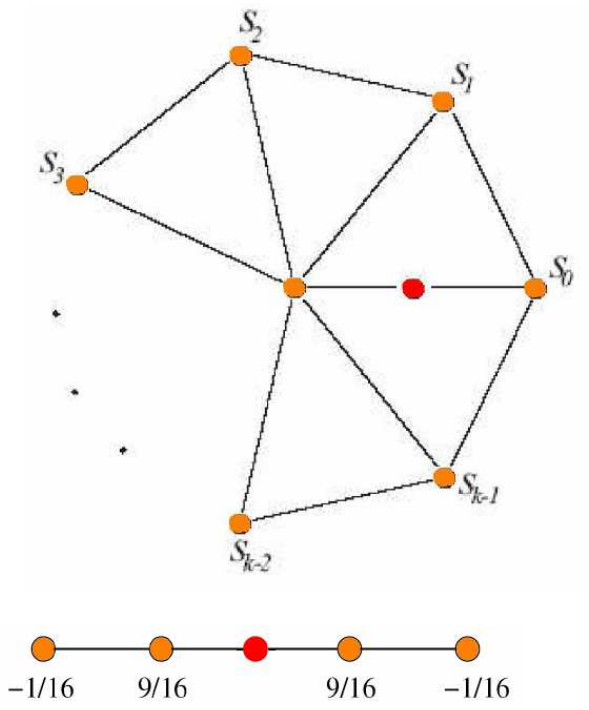

FIG. 1.20 – Schéma de subdivision de Butterfly (maillage irrégulier ou bords) [extrait de [18]]

Les points créés sont les barycentres affectés des coefficients $\frac{9}{16}$ et $\frac{-1}{16}$:

$$E_i = \frac{9}{16}(d_i + d_{i+1}) - \frac{1}{16}(d_{i-1} + d_{i+2})$$

Subdivision de Kobbelt

C'est un schéma interpolant et primal [21] qui s'applique à des maillages quadrangulaires et assure une continuité C^1 pour tous les points qu'ils soient réguliers ou non. Comme pour Butterfly, les points du maillage initial sont conservés et restent donc dans le maillage après subdivision.

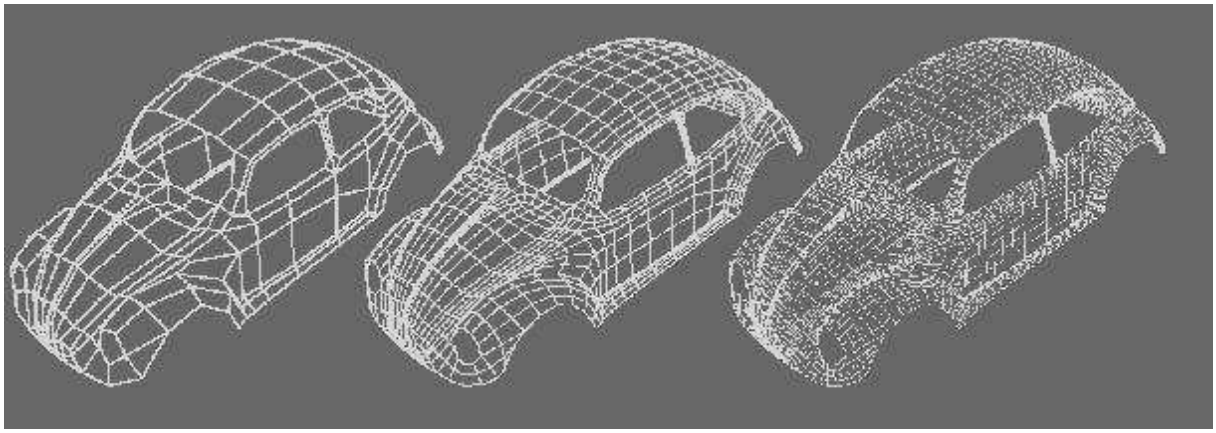

FIG. 1.21 – Exemple de subdivision par l'algorithme de Kobbelt

Le schéma de fonctionnement sur les maillages réguliers est résumé par le schéma suivant :

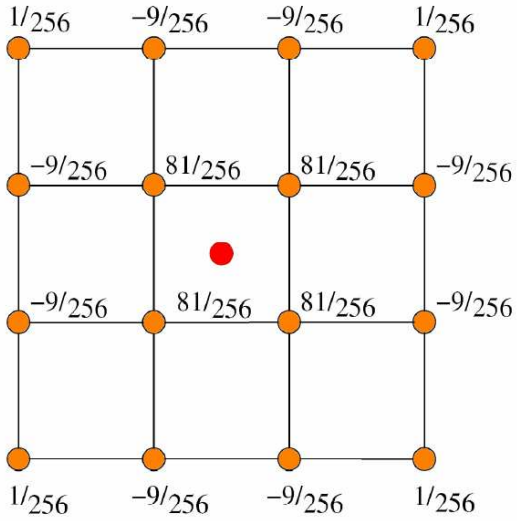

FIG. 1.22 – Schéma de subdivision de Kobbelt (maillage régulier) [extrait de [18]]

Pour les maillages irréguliers ou pour les sommets des bords, on utilise, là aussi, des masques spécifiques :

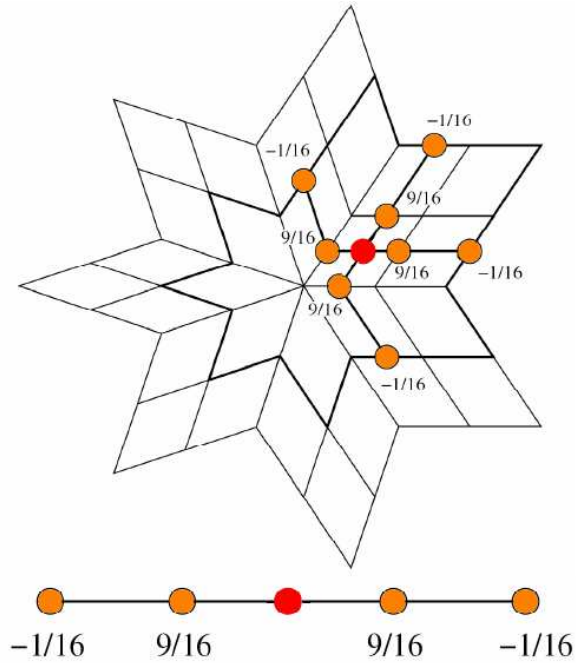

FIG. 1.23 – Schéma de subdivision de Kobbelt (maillage irrégulier ou bords) [extrait de [18]]

Pour un sommet irrégulier ($n \neq 4$), la face voisine ne prend en compte que les milieux des arêtes la composant et les milieux des arêtes suivantes (cf. figure du dessus).

Pour un sommet du bord d'un maillage ($n = 2$), on ne pondère seulement que les points voisins (cf. figure du bas).

Subdivision de Doo-Sabin

C'est un schéma approximant et dual [13] qui s'applique à des maillages quadrangulaires et assure une continuité C^1 pour les points réguliers, C^0 pour les autres. Dans le cas d'un schéma dual, rappelons que l'on ne cherche pas à subdiviser les faces mais à diviser les sommets : à un point du maillage de contrôle on fait hériter plusieurs sommets dans le maillage subdivisé.

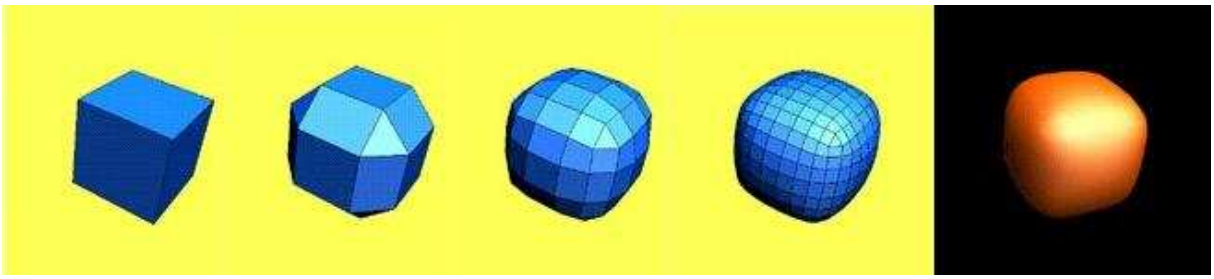

FIG. 1.24 – Exemple de subdivision par l'algorithme de Doo-Sabin

Le schéma de subdivision est le suivant :

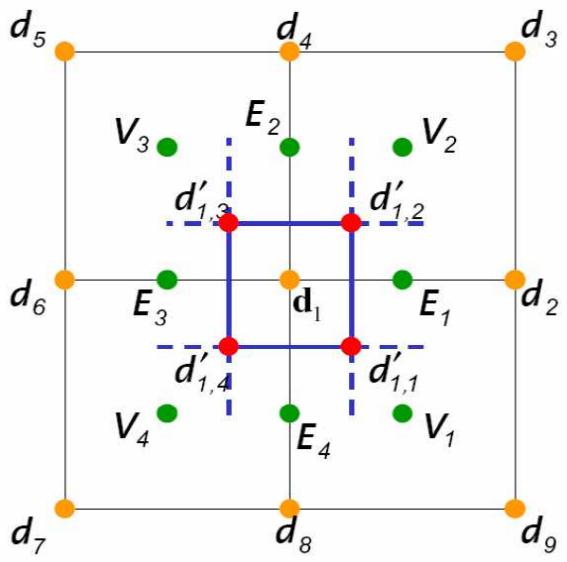

FIG. 1.25 – Schéma de subdivision de Doo-Sabin [extrait de [18]]

Avant de créer les points finaux du maillage subdivisé, on utilise des points de construction (en vert sur la figure) :

- Les points V_i sont les isobarycentres des faces.
- Les points E_i sont les milieux des arêtes reliant un sommet d_i à ses voisins.

Les points du maillage final créés par la subdivision sont les points $d'_{1,j}$ qui héritent de d_1. On les calcule grâce à l'équation suivante :

$$d'_{1,j} = \frac{1}{4}(d_1 + E_j + E_{j-1} + V_j)$$

Subdivision $\sqrt{3}$

C'est un schéma approximant et primal [22] qui s'applique à des maillages triangulaires et assure une continuité C^2 pour les points réguliers, C^1 pour les autres. C'est un schéma intéressant car on opère de la même manière pour tout point de valence $n > 3$, ce qui évite ainsi de gérer trop de cas différents.

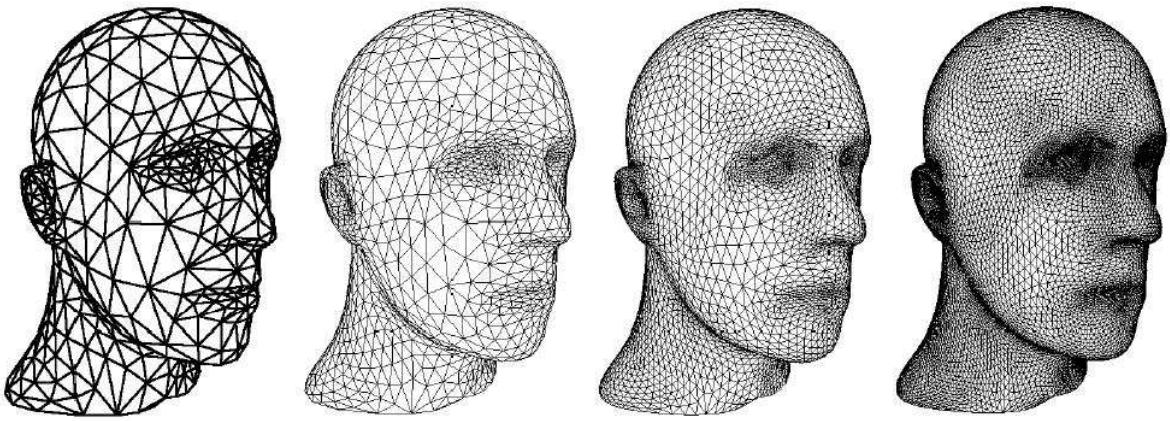

FIG. 1.26 – Exemple de subdivision par l'algorithme $\sqrt{3}$

Le masque utilisé est le suivant :

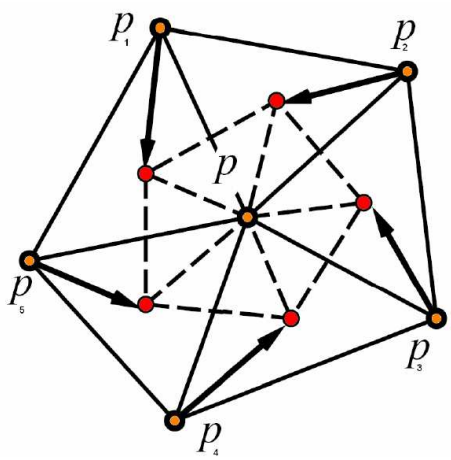

FIG. 1.27 – Schéma de subdivision de l'algorithme $\sqrt{3}$ [extrait de [18]]

Les nouveaux points créés sont les isobarycentres des triangles voisins. Le point p est ensuite déplacé grâce à l'équation suivante :

$$p' = (1 - \alpha_n) \cdot p + \alpha_n \cdot \frac{1}{n} \Sigma_{i=0}^{n-1} p_i$$

où : $\alpha_n = \frac{4 - 2\cos\left(\frac{2\pi}{n}\right)}{9}$

Subdivision de Catmull-Clark

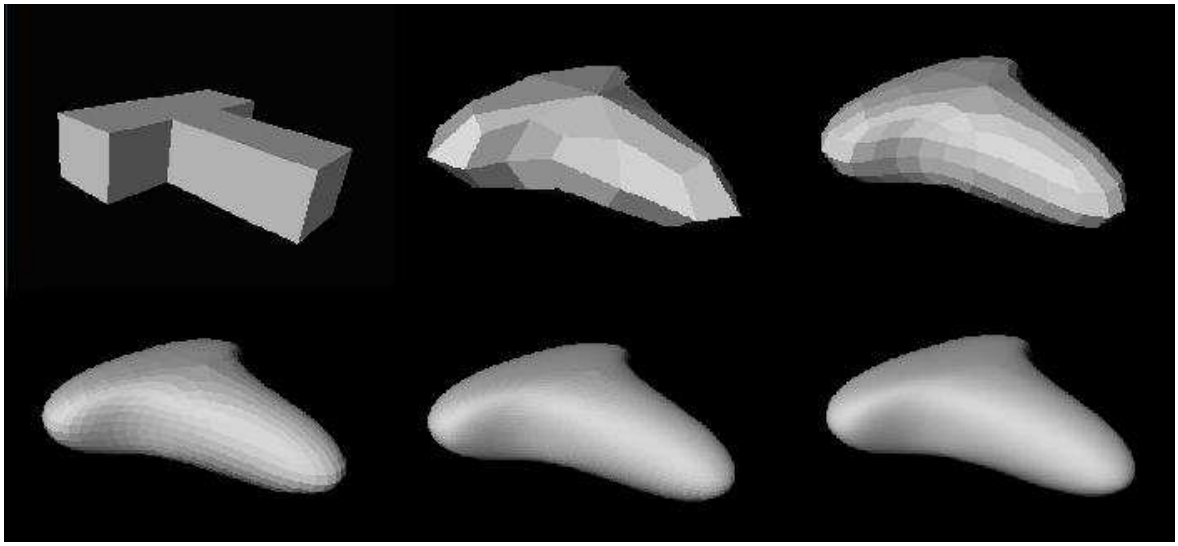

FIG. 1.28 – Exemple de raffinement avec l'algorithme de Catmull-Clark (5 subdivisions successives)

C'est un schéma approximant et primal qui date de 1978 [8]. Il s'applique à des maillages quadrangulaires et permet d'obtenir une continuité C^2 pour les points réguliers, C^1 pour les autres.

Il est massivement employé dans l'industrie notamment pour les images de synthèse de haute qualité (ie avec des résolutions importantes) comme par exemple dans les longs métrages de Pixar. Il est supérieur en qualité mais également en temps de calcul à d'autres schémas comme Butterfly ou Kobbelt ce qui ne permet pas toujours de l'utiliser pour des applications temps réel. En général, pour de telles applications, on privilégie les schémas d'interpolation qui sont moins gourmands en temps de calcul car ils ne sont pas définis de manière récursive. La moins bonne qualité des formes obtenue avec des schémas interpolant n'est pas très visible en temps réel mais est un critère défavorable pour des images très détaillées.

L'implémentation d'un algorithme de Catmull-Clark pour une application temps réel est donc primordiale afin de pouvoir conjuguer les avantages du temps réel et la qualité du rendu et c'est ce qui, en partie, a motivé notre démarche.
Le Catmull-Clark que nous avons implémenté n'est pas l'algorithme original mais se base sur un article de Joe Warren et de Scott Schaefer (Rice University) intitulé *A Factored Approach To Subdivision Surfaces* [30] que nous décrirons plus loin dans une partie spécifique.

La méthode "classique" de subdivision est la suivante :

- On crée des points de face qui sont les isobarycentres des sommets de chaque face originelle.
- On crée des points d'arêtes qui sont les isobarycentres des deux sommets de l'arête et des deux points de face appartenant aux faces adjacentes à cette arête.
- On déplace les anciens sommets di selon l'équation suivante :

$$\frac{S}{n} + \frac{2R}{n} + \frac{d_i(n-3)}{n}$$

où :

- S est l'isobarycentre des nouveaux points de face des faces passant par l'ancien sommet
- R est l'isobarycentre des milieux des arêtes passant par l'ancien sommet
- di est l'ancien sommet
- n est la valence de l'ancien sommet

Ce qui est représenté par le schéma suivant :

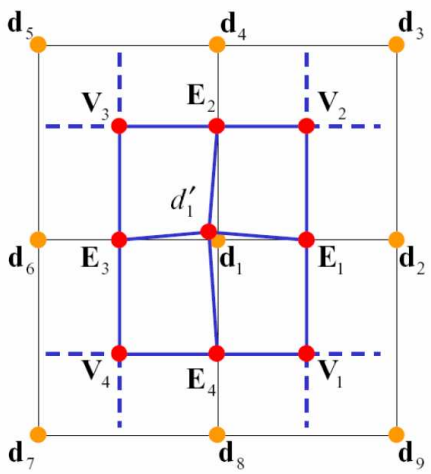

FIG. 1.29 – Schéma de subdivision de Catmull-Clark [extrait de [18]]

Les points V_i sont les points de face, les E_i les points d'arêtes et d'_1 le sommet d_1 après repositionnement.

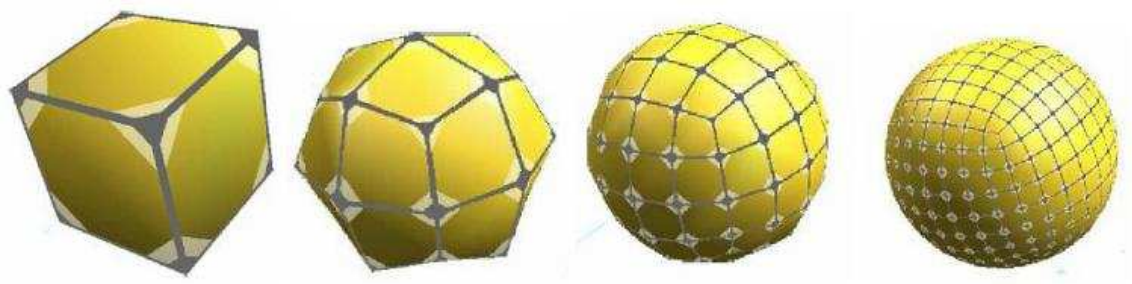

FIG. 1.30 – Exemple de raffinement d'un cube avec l'algorithme de Catmull-Clark : les coins sont arrondis et le processus de subdivision tend à s'approcher d'une sphère

Tableaux comparatifs

Nous avons donc vu que chaque schéma de subdivision a ses propres caractéristiques ce qui donne des résultats visuels qui peuvent être relativement différents selon celui utilisé. Le premier tableau suivant résume les caractéristiques de chaque schéma présenté. Le deuxième tableau présente les différences visuelles notables pour 4 schémas de subdivision : Catmull-Clark, Loop, Doo-Sabin et Butterfly.

Schéma	**Type**	**Primal / Dual**	**Maillage**	**Continuité**
Loop	approximation	primal	triangulaire	C^2 pour les points réguliers, C^1 sinon (valence $\neq 6$)
Butterfly	interpolation	primal	triangulaire	C^1 pour les points réguliers, C^0 sinon (valence $\neq 6$)
Kobbelt	interpolation	primal	quadrangulaire	C^1
Doo-Sabin	approximation	dual	quadrangulaire	C^1 pour les points réguliers, C^0 sinon (valence $\neq 4$)
$\sqrt{3}$	approximation	primal	triangulaire	C^2 pour les points réguliers, C^1 sinon (valence $\neq 6$)
Catmull-Clark	approximation	primal	quadrangulaire	C^2 pour les points réguliers, C^1 sinon (valence $\neq 4$)

FIG. 1.31 – Tableau récapitulatif des propriétés des différents schémas de subdivision

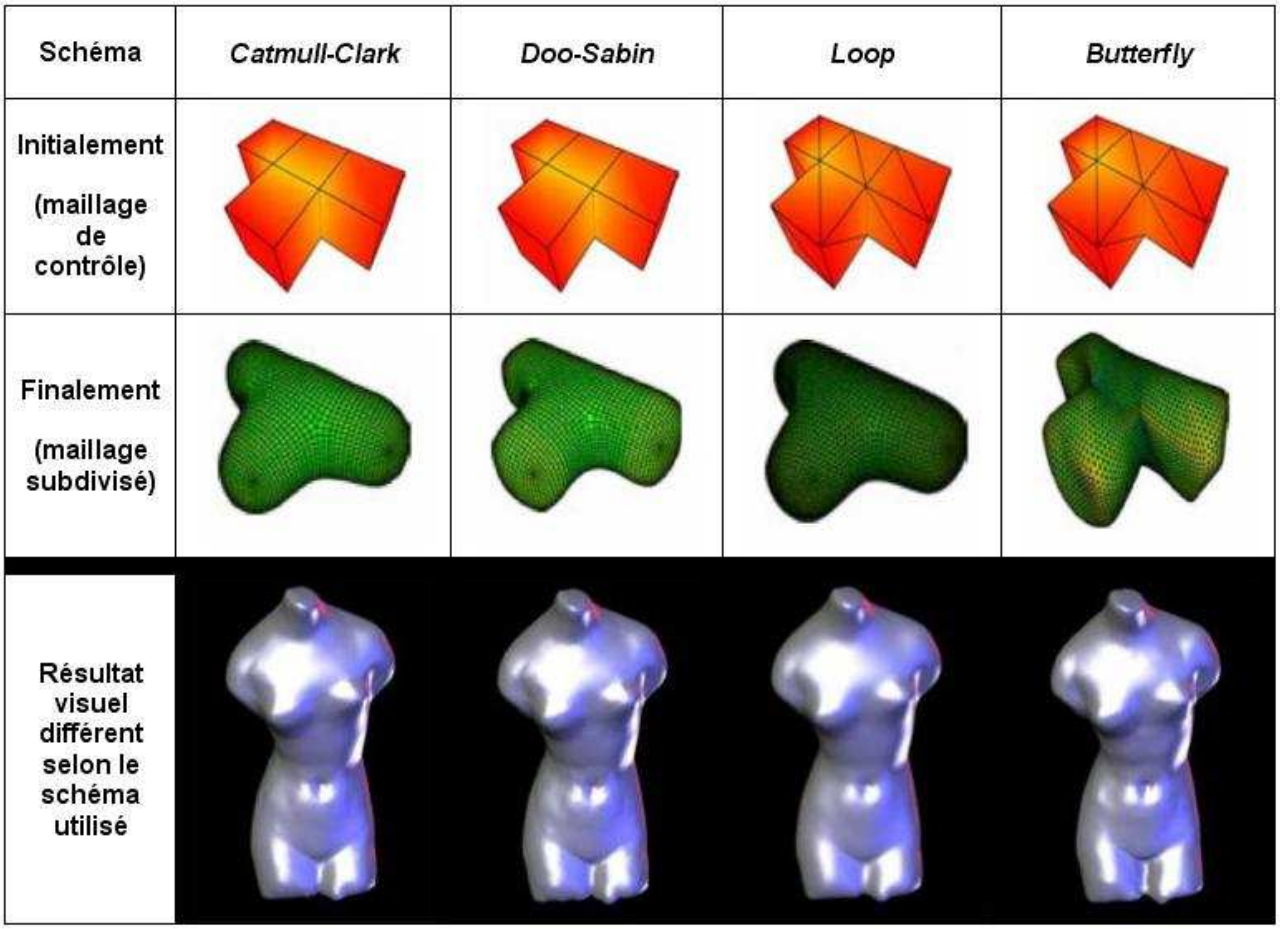

FIG. 1.32 – Différences visuelles observées selon le schéma utilisé

Chapitre 2

Introduction au sujet

La puissance croissante des ordinateurs et notamment celle du hardware graphique permet une visualisation 3D de plus en plus réaliste. Les objets 3D d'une scène sont en général définis par des maillages polygonaux afin d'être traités plus facilement par le pipeline graphique du GPU. C'est la taille de ce maillage (ie le nombre de polygones) qui permet d'obtenir un meilleur rendu en affinant l'image.

Le but de ce projet est d'implémenter un noyau de subdivision sur le GPU afin d'exploiter au maximum les performances de ce dernier pour les calculs sur les nombres flottants et, en même temps, de soulager le processeur central afin qu'il se concentre sur d'autres tâches. Ce critère de performances est crucial car notre application doit se dérouler en temps réel. Le but final est donc d'optimiser chaque fonctionnalité d'affichage afin de pouvoir réaliser cette subdivision en temps réel tout en gardant un taux de rafraîchissement de l'image raisonnable.
Pour cela, il est nécessaire d'utiliser des fonctionnalités récentes d'OpenGL et, en particulier, tout ce qui attrait aux *shaders*, aux VBOs et FBOs.
On s'attend donc à obtenir des résultats supérieurs (en terme de FPS) à ceux d'une subdivision temps réel qui se dérouleraît entièrement sur le CPU puisqu'une partie des tâches nécessaires à la subdivision est réalisée par un processeur spécialisé. Par conséquent, afin de valoriser les résultats attendus, il est aussi nécessaire d'implémenter un noyau de subdivision entièrement sur CPU ce qui nous a conduit à réaliser deux applications afin de mesurer l'efficacité réelle d'une telle implémentation.

Nous présenterons donc, dans la suite de ce rapport, le cahier des charges, les démarches préliminaires à ce projet puis les structures communes aux deux programmes avant de décrire leurs fonctionnalités respectives et de comparer les résultats obtenus.

Chapitre 3

Cahier des Charges

3.1 Besoins fonctionnels

Cette section présente les principales fonctionnalités que nous devions implémenter pour ce projet. Chaque besoin est présenté succintement puis décrit de façon plus détaillée dans la suite du rapport.

3.1.1 *Parser* OBJ

Afin de pouvoir travailler sur des maillages polygonaux préexistants, il nous faut disposer d'un *parser* de fichiers dans lequels sont décrits ces maillages. Le format de fichiers que nous utilisons est le format OBJ. Ce dernier, étant écrit en format ASCII, est facilement éditable manuellement.

3.1.2 Subdivision au niveau du CPU

Notre application doit être capable de réaliser une subdivision de type Catmull-Clark sur un maillage de base ou réseau de contrôle. Toutes les phases de subdivision et de calculs sont réalisées au niveau du CPU. L'idéal serait que notre implémentation soit suffisement rapide pour permettre un affichage temps réel où la subdivision serait appliquée à chaque face. Dans ce dernier cas, la carte graphique n'intervient qu'au niveau de l'affichage de l'objet subdivisé. C'est donc le maillage déjà subdivisé qui lui est fourni.

3.1.3 Subdivision temps réel au niveau du GPU

Notre programme doit aussi pouvoir réaliser une subdivision de type Catmull-Clark sur un maillage de base ou réseau de contrôle à l'aide du GPU. Ainsi, seul le maillage de base sera fourni à la carte graphique cette dernière se chargeant à la fois de la subdivision et de l'affichage de l'objet.

3.1.4 Visualisation

La visualisation doit être épurée et présenter un confort d'utilisation suffisant pour permettre de se concentrer essentiellement sur l'objectif du projet : la comparaison entre deux algorithmes de subdivisions en temps réel. Pour cela il est nécessaire d'utiliser les dernières optimisations de la librairie OpenGL pour bénéficier au maximum des performances de la carte graphique et donc d'une vitesse d'affichage optimale. Dans cette optique, plusieurs extensions sont utilisées dont notamment les VBOs. En ce qui concerne l'interface, des éléments de comparaisons de performances sont affichés en permanence :

- Les FPS : nombre de rafraîchissements de la fenêtre OpenGL en une seconde, permettent de se rendre compte de la rapidité de l'exécution et du rendu de l'algorithme.

- Le niveau de subdivision courant (qui peut être changé en utilisant le menu contextuel à partir du bouton droit de la souris) peut correspondre au maillage de base ou à un des cinq niveaux de subdivision disponibles

- Le nombre de polygones du maillage donne une indication de la résolution de l'objet.

3.2 Besoins non fonctionnels

3.2.1 Rapidité

La rapidité est ici primordiale : le but de notre projet est de réaliser et comparer des méthodes de subdivision en temps réel. A chaque image affichée, on procède à la subdivision du maillage. La vitesse d'affichage ainsi que la nature "temps réel" du programme dépendent donc directement de notre implémentation.

3.2.2 Gestion efficace de la mémoire

Pour les mêmes raisons que celles évoquées précédemment, la gestion de la mémoire est elle aussi cruciale. Si les algorithmes de subdivision sont mal implémentés, ces derniers étant exécutés plusieurs dizaines de fois par seconde, la moindre fuite mémoire peut rapidement devenir désastreuse.

3.2.3 Interface succinte

Le but ne notre application étant avant tout de tester différents algorithmes et leurs performances, la simplicité de l'interface s'impose pour deux raisons :

- En termes de performances, cette dernière ne doit pas avoir d'influence sur les résultats obtenus.
- En terme de besoins, très peu d'interactions sont requises avec l'utilisateur : seul le changement de niveau de subdivision et la navigation dans la fenêtre de visualisation sont nécessaires.

3.2.4 Outils de développement

Notre application sera développée en C++ sous environnement Linux. L'API graphique utilisée sera OpenGL version 1.5. La partie concernant la programmation de carte graphique utilisera le langage de *shader* GLSL. La configuration des PC utilisés sera la suivante :

- Intel Pentium4 2.4GHz
- 512Mo RAM
- Carte graphique NVidia Geforce 6600GT 256Mo avec driver v81.78

Chapitre 4

Démarches préliminaires

4.1 Render to texture et Render to vertex array

4.1.1 Render to texture

La plupart des résultats GPGPU [19] [25] est le fruit d'une approche *render to texture* (RTT). Le rendu dans un *buffer* hors-écran ensuite utilisé comme texture permet d'obtenir des effets difficilement appréhendables par une autre technique comme les miroirs, les ombres. Le *render to texture* est utilisée dans l'application pour effectuer un rendu multi-passes via l'utilisation d'un langage de *shaders*.

La technique de *render to texture* consiste simplement à considérer l'affichage à l'écran (*framebuffer*) comme une texture pour pouvoir, par la suite, utiliser celle-ci dans une deuxième phase. Il existe plusieurs approches pour implémenter un *render to texture*. Toujours dans un souci de performances, les techniques reposant principalement sur le CPU ont été très vite écartées au profit de celles n'utilisant que le GPU.

Bien que non utilisées dans le programme, plusieurs approches basés sur le CPU ont été implémentées lors de la mise en place des différents éléments du programme. Un bref descriptif est présenté ci-dessous [29].

Les méthodes basées CPU fonctionnent selon le schéma suivant :

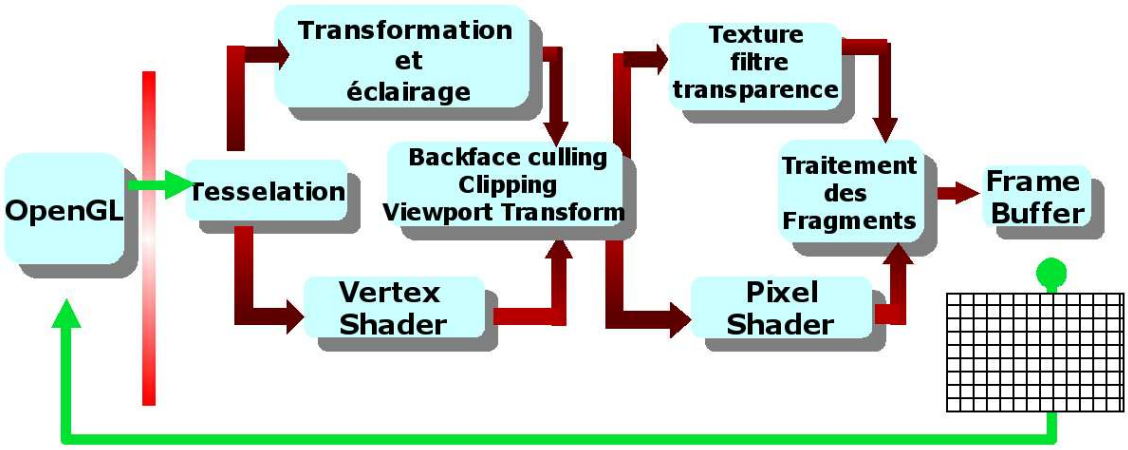

FIG. 4.1 – pipeline Render to texture [29]

On distingue trois méthodes principales :

- glReadPixels() et glTexImage*()
- glCopyTexImage*()
- glCopyTexSubImage*()

Toutes présentent l'inconvénient de nécessiter des transferts vers le CPU et sont donc relativement lentes et peuvent provoquer une perte de précision.
Les approches basées GPU sont, elles, beaucoup plus rapides [31]. Deux techniques sont présentent dans les extensions d'OpenGL : les *pbuffers* et les *Frame Buffer Object* (FBO) [17] [4]. Les *Frame Buffer Object* permettent un *render to texture* flexible et efficace et remplacent à l'heure actuelle les *pbuffers* qui sont beaucoup plus complexes et de mise en œuvre moins souple. Un autre avantage des FBOs est le fait qu'ils soient multi-plateformes contrairement aux *pbuffers*. Néanmoins, les FBOs n'ont pu être utilisés dans un premier temps du fait d'un problème avec les pilotes de la carte graphique. En effet, les FBOs sont une extension très récente sur les cartes NVidia et nécessitent des pilotes à jour. Ainsi une version utilisant les pbuffers à été envisagée avant que les pilotes corrects ne soient mis à jour. Cette mise à jour n'a eu lieu qu'une dizaine de jours avant le remise de ce rapport.

4.1.2 Utilisation des Frame Buffer Object

Les FBOs permettent le rendu directement dans une texture et, de ce fait, sont bien plus efficaces que les méthodes comme *glCopyTexImage2D()*. La texture ainsi générée peut facilement être utilisée pour une nouvelle passe de rendu. Le schéma général des FBOs est le suivant :

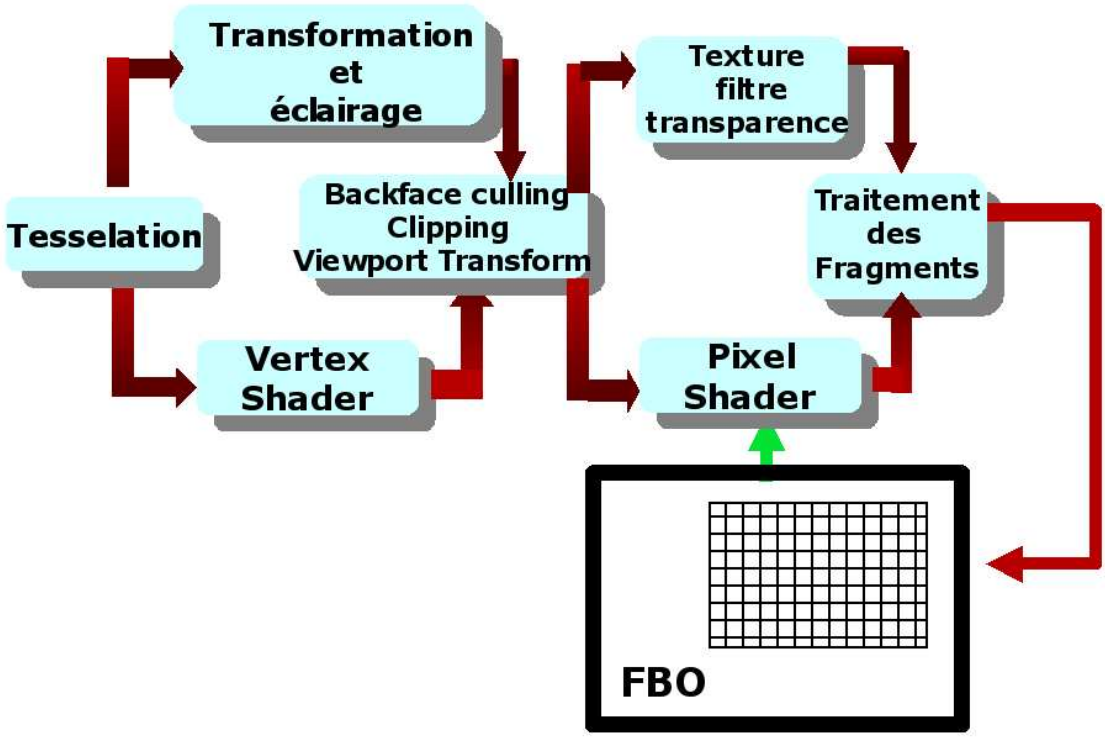

FIG. 4.2 – pipeline d'un render to texture avec FBO [29]

Du point de vue fonctionnel, l'implémentation d'un FBO s'apparente à celle d'un VBO décrite plus loin. Les étapes de son initialisation sont les suivantes :

Le *frame buffer object* est généré de manière similaire à une texture avec :

```
glGenFramebuffersEXT(1, &fbo)
```

La texture *tex* associée au FBO est générée avec :

```
glGenTextures(1, &tex)
```

Puis le *buffer* courant est désigné par l'appel d'une fonction de *binding* :

```
glBindFramebufferEXT (GL_FRAMEBUFFER_EXT, fbo)
```

Désormais, les opérations se dérouleront sur le FBO. La texture *tex* qui va stocker l'image du *framebuffer* est initialisée :

```
glBindTexture(GL_TEXTURE_RECTANGLE_NV, tex)
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_RGBA, WIDTH,HEIGHT,
             0, GL_RGBA, GL_UNSIGNED_BYTE, NULL)
```

Finalement, la texture est liée au FBO courant avec :

```
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_RECTANGLE_NV, tex, 0)
```

Il est possible de désactiver le FBO pour permettre l'affichage dans le *framebuffer* en utilisant :

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

Une fois cette phase d'initialisation terminée, les FBOs peuvent directement être utilisés pour effectuer le *render to texture*. Combinés à l'utilisation des *shaders*, principalement du *fragment shader*, le rendu multi-passes sera assuré totalement sur le GPU et donc bien plus rapidement que par les méthodes évoquées précédemment. Pour ce faire, à chaque nouvelle passe la texture associée au FBO courant est plaquée sur un polygone. Elle subie alors les modifications, filtrages ou toute opération spécifique du *shader*, puis est de nouveau rendue sur le FBO pour une nouvelle passe éventuelle.

4.1.3 Render to vertex array avec les Pixels Buffer Object

Les FBOs nous permettent le rendu d'une texture en effectuant plusieurs passes de traitement. Seulement, il est intéressant et même souvent nécessaire d'utiliser la texture finale pour former un nouveau maillage. Chaque pixel **RGBA** correspondant à un sommet de coordonnées **XYZW**. Cette opération peut s'effectuer en utilisant simplement la

fonction d'OpenGL *glReadPixels()* avec des paramètres corrects. En utilisant uniquement cette fonction, le *buffer* qui sera ainsi construit pourra être réutilisé pour la construction puis l'affichage d'un VBO. Malheureusement, cette méthode a comme inconvénient majeur d'effectuer des opérations sur le CPU et donc d'utiliser le bus de transfert.

Pour palier à ce problème, l'extension des *pixels buffers object* (PBO) [3] va permettre un *render to vertex array* (RTVA) [5] en récupérant la texture du FBO puis en "*castant*" directement un PBO en VBO pour l'affichage du maillage. Cette extension n'apporte pas de nouvelles fonctionnalités aux *buffers object* eux-mêmes. Elle ajoute simplement deux autres cibles (paramètre *target* de la fonction *glBindBufferARB()*) :
PIXEL_PACK_BUFFER et PIXEL_UNPACK_BUFFER pouvant servir lors de l'opération de *binding*. Lorsqu'un *buffer object* est initialisé avec PIXEL_PACK_BUFFER, les commandes comme *glReadPixels()* vont écrire leur données dans le *buffer object* (en les lisant depuis le *framebuffer*). Avec PIXEL_UNPACK_BUFFER, les commandes comme *glDrawPixels()* vont lire leur données à partir du *buffer object* (puis les dessiner dans le *framebuffer*).

Ainsi, pour effectuer le passage *render to vertex array* une fois que l'image est rendue, il est possible stocker cette image dans le *buffer object* via *glReadPixels()*. Ensuite, il suffit de réutiliser ce *buffer* en tant que nouvelle source de données à l'aide des VBOs.

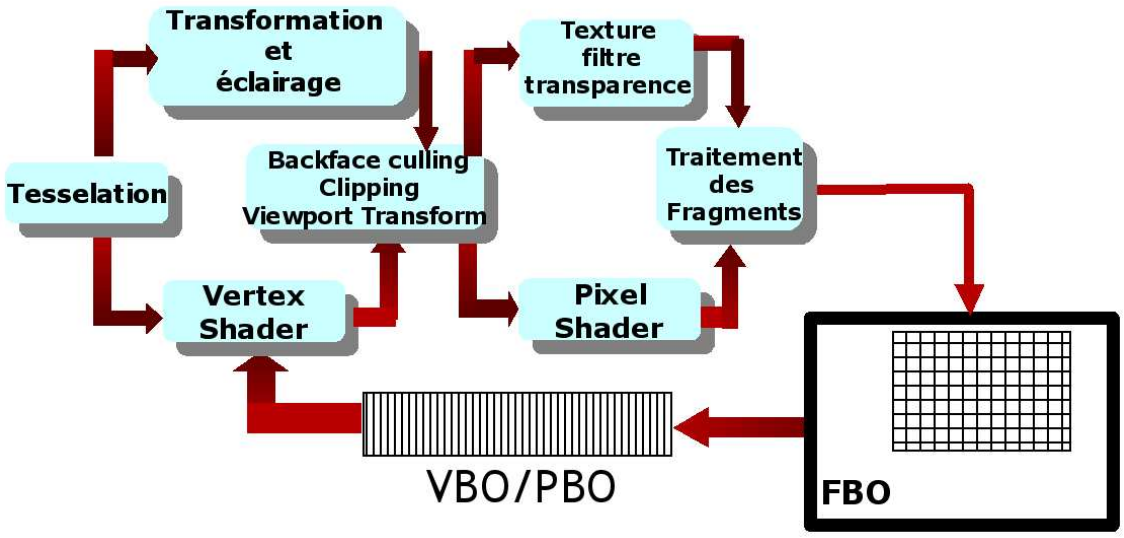

FIG. 4.3 – pipeline Render to Vertex Array [29]

Voici un exemple simple de programme effectuant le *render to vertex array*.

Création du *buffer object* :

```
glGenBuffers(1, bufferObject);
glBindBufferARB(PIXEL_PACK_BUFFER_EXT, bufferObject);
glBufferDataARB(PIXEL_PACK_BUFFER_EXT, numberVertices*4, NULL, GL_DYNAMIC_DRAW_ARB);
```

Affichage d'un polygone texturé dans le *framebuffer*. En combinant cette étape avec l'utilisation des FBOs et les *shaders*, on effectue du rendu multi-passes.

```
RenderToTexture() ;
```

Remplissage du *buffer object* depuis le *framebuffer*

```
glReadPixels(0, 0, numberVertices*4, 1, GL_RGBA, GL_FLOAT, BUFFER_OFFSET(0));
```

Passage du PBO au VBO : la texture est interprétée comme une suite de coordonnées de sommets où chaque pixel (RGBA) est un sommet (XYZW).

```
glBindBuffer(GL_ARRAY_BUFFER, bufferObject);
```

Puis, pour afficher directement le nouveau maillage :

```
glEnableClientState(VERTEX_ARRAY);
glVertexPointer(4, FLOAT, 0, BUFFER_OFFSET(0));
glDrawArrays(GL_TRIANGLES, 0, numberVertices);
```

Ici, on dessine des triangles en considérant chaque sommet comme la suite de quatre FLOAT consécutifs dans le *Vertex Array*.

4.2 Utilisation des VBOs

Un *vertex buffer object* est un outil puissant qui permet de stocker les données dans la mémoire haute performance : celle de la carte graphique. Il nécessite pour cela les extensions récentes d'OpenGL et est inclus dans OpenGL 1.5 [26].

4.2.1 Intérêt des VBOs

Les VBOs sont utilisés ici pour bénéficier des meilleures performances possibles des cartes graphiques au niveau de l'affichage et pour que l'application OpenGL bénéficie de leur flexibilité [2]. En effet, une comparaison entre les différentes techniques d'affichage montre clairement la grande efficacité des techniques utilisant la mémoire graphique comme les *display list* et les VBOs lorsque le nombre de polygones devient important.

4.2.2 Comparaison des méthodes d'affichage d'OpenGL

OpenGL fournit traditionnellement deux approches principales pour afficher les données géométriques : le mode immédiat et les *display list* [6].

Lorsqu'on utilise le mode immédiat, l'application envoie simplement les données au GPU à chaque *frame* ce qui est avantageux dans les situations comme la modélisation et l'animation, c'est-à-dire lorsque la géométrie est fréquemment créée ou modifiée. C'est le mode d'affichage le plus simple à utiliser puisqu'il suffit de faire appel aux fonctions *glVertex*()* pour placer un point, *glNormal*()* pour lui associer une normale, *glColor*()* pour donner une couleur au vertex et *glTexCoord*()* pour lui donner des coordonnées de texture. Cependant, l'inconvénient principal de cette méthode est sa faible vitesse d'affichage du fait des transferts de données en tant qu'élément individuel et des goulots d'étranglement sur le bus du CPU.

L'approche avec *display list* quant à elle permet au GPU de tirer les données directement de la mémoire avec le transfert DMA. Ainsi, les *display list* permettent de transférer en une seule fois un maximum de données. Malgré tout, les *display list* présentent quelques désavantages. Dans certaines situations, les modifications des données géométriques requièrent la création d'une nouvelle *display list*. Selon la fréquence à laquelle la géométrie est mise à jour, les performances seront nuancées par la complexité des opérations de création et de suppression des listes. De plus, le GPU utilise les *display list* à travers une copie sauvegardée par le serveur OpenGL. Cela crée une redondance des données comparées au mode immédiat et peut provoquer des erreurs lorsque l'espace mémoire est insuffisant.

En alternative aux *display list*, OpenGL implémente également les *vertex array*. Ils ont

pour but de grouper et de traîter les vertex et les attributs en un bloc. Ce mode permet aussi d'indexer l'affichage grâce à un tableau d'indices : ceci évite d'envoyer plusieurs fois les même points à la carte graphique. Les *vertex array* autorisent les données de géométrie et de couleur à être entre-mélées : on parle alors d'*interleaved vertex array*.

4.2.3 Tableau récapitulatif

Les VBOs permettent donc une amélioration des performances d'OpenGL en combinant les bénéfices du mode immédiat avec ceux des *display list* et des *vertex array* tout en évitant leurs limitations. Ils admettent ainsi que les données soient groupées et stockées efficacement à la manière des *vertex array* pour optimiser le transfert de données. Ils fournissent également aux applications la flexibilité autorisant la modification de la géométrie sans provoquer de coût supplémentaire dû à la validation.

Méthode	**Stockage**	**Dynamique / Statique**	**Vitesse d'affichage**[1]
Immediat mode	-	Dynamique	*
Display list	Mémoire graphique[2]	Statique	****
Vertex Array	RAM	Dynamique	**
Interleaved Vertex Array	RAM	Dynamique	**
Vertex Buffer Object	Mémoire graphique	Dynamique	****

FIG. 4.4 – Tableau récapitulatif des méthodes d'affichage OpenGL

[1] comparaison des FPS sur une GeForce 6600 GT pour environ 10 000 polygônes
[2] en cache selon les *drivers*

4.3 Utilisation des *shaders*

Les *shaders* sont très flexibles et vont nous permettre d'effectuer des traitements en utilisant toute la puissance du GPU. Les données en entrée (le maillage de base) doivent être converties en texture afin d'être manipulables par le GPU. Ensuite, un *pixel shader* sera appliqué sur la texture et nous obtiendrons en sortie une texture modifiée qui correspond à notre maillage subdivisé.

Notre texture sera créée dans le programme OpenGL. Ensuite, nous allons récupérer l'identifiant de la variable dans le shader avec la fonction [27] :

```
GLint glGetUniformLocationARB(GLhandleARB program, const GLcharARB *name);
```

Une unité de texture sera choisie par l'appel à :

```
void glActiveTexture(GLenum texture);
```

à laquelle on associe la texture précedemment créée dans le programme OpenGL avec :

```
void glBindTexture (GLenum target, GLuint texture);
```

Enfin, on initialise les données du sampler[1] avec :

```
void glUniform1i(GLint location, GLint value);
```

[1] type de données GLSL représentant une texture

Chapitre 5

Algorithmes et structures de données utilisés

Dans cette partie, nous allons décrire, d'une façon non exhaustive, l'architecture de notre programme et les principales structures de données utilisées. En réalité, nous avons développé non pas une seule mais deux applications : une pour la subdivision sur le CPU, l'autre pour la subdivision sur GPU.
Ces deux programmes se différencient donc dans leur fonctionnement et leur utilisation mais ils s'appuient sur les mêmes structures de base.

5.1 Structures de base

Les deux applications ont donc des structures communes à savoir (diagramme de classes disponible en annexe) :

- une classe **Vertex** pour stocker en mémoire la position des sommets
- une classe **Face** qui permet de définir une face grâce à des références vers les sommets de cette dernière
- une classe **Model** qui définit un maillage polygonal avec une liste de Vertex et une liste de Face.

Pour stocker tout ce qui est sous la forme d'une liste (liste de Vertex, de Face, de Model...), nous avons décidé d'utiliser le container **vector** de la STL qui permet de stocker efficacement ces données et surtout permet un accès à ces éléments beaucoup plus rapide qu'une structure personnelle.

5.1.1 Classe *Vertex*

Cette classe permet de définir un point avec ses coordonnées dans l'espace R3. On associe également à chaque point une valence qui est utilisée lors de la subdivision de Catmull-Clark sur CPU.

Nous avons, de plus, redéfini la plupart des opérateurs mathématiques afin de faciliter les opérations (addition, soustraction, multiplication...) sur les objets Vertex.
Pour la subdivision sur GPU, nous avons du également associer à chaque Vertex une liste de faces adjacentes (de type *vector<Face*>*). En effet, comme il est expliqué plus loin (cf. *Algorithme de subdivision sur GPU*), avant de faire la subdivision au niveau du GPU, nous devons tout d'abord subdiviser une première fois le maillage de contrôle puis découper ensuite le maillage subdivisé obtenu en morceaux appelés *Frag Mesh*.

Cette découpe, effectuée en spirale, nécessite de connaître les faces adjacentes à une face, ces dernières étant calculées grâce aux faces adjacentes de chaque Vertex.

5.1.2 Classe *Face*

Cette classe permet de stocker en mémoire les données relatives à une face à savoir la liste des sommets du polygone (*vector<int>*). Cette liste ne contient pas directement les objets Vertex associés à la face mais des indices vers la liste des sommets du maillage. Afin d'avoir accès aux données des Vertex, il faut donc une structure qui stocke cette liste de sommets et la liste des faces du maillage.

Comme pour la classe Vertex, nous avons du associer à chaque face une liste de faces adjacentes afin de pouvoir calculer les *Frag Mesh* pour la subdivision sur GPU.

5.1.3 Classe *Model*

Cette classe permet de définir un maillage en mémoire. Elle contient comme attributs une liste de Vertex (*vector< Vertex*>*), une liste de normales (*vector< Vertex*>*), une liste de coordonnées de textures (*vector< Vertex*>*) et une liste de Face (*vector<Face*>*).

Nous avons utilisé des *vector* de pointeurs car les objets Vertex et Face étant souvent modifiés par la subdivision temps réel, cela nous permet de ne modifier que les attributs de ces objets et non de recréer de nouveaux objets.

Les listes de normales et de coordonnées de textures ne sont pas toujours utilisées. Cela dépend de la présence de telles informations dans le fichier décrivant le maillage de

contrôle.

La classe Model contient également le *parser* OBJ décrit plus bas qui permet de stocker toutes les données d'un maillage de contrôle décrit dans un fichier *.obj*.

Comme pour les classes Vertex et Face, la classe Model a été modifiée pour l'application sur GPU. Elle contient, en plus, des fonctions de calcul d'adjacence de faces (pour les sommets et pour les faces).

5.2 *Parser* OBJ

La visualisation de maillages polygonaux nécessite de disposer d'objets 3D définis dans des fichiers qu'il faut ensuite parser afin de pouvoir stocker le ou les maillages en mémoire. Plusieurs types de fichiers existent pour cela comme par exemple les fichiers *VRML* (*.wrl*), *PGN* (*.pgn*) ou *OBJ* (*.obj*). La syntaxe de ces fichiers, bien que différente, est basée sur les mêmes composantes à savoir l'ensemble des sommets du maillage avec leurs coordonnées et l'ensemble des faces avec des références vers les sommets qui la définissent. Suivant le type de fichier ou la finesse des informations sauvegardées, on dispose également de coordonnées de normales et de textures.

Notre application étant surtout utile pour comparer les performances entre une méthode de subdivision en temps réel s'effectuant sur le CPU et une autre s'effectuant sur le GPU, il nous était simplement demander de développer un *parser* de fichiers OBJ.

Ainsi, nous n'avons pas défini de classe générique telle que *FiltreFichierEntrée* dont nous aurions fait dériver plusieurs classes, chacune définissant un *parser* spécifique à un type de fichier (par exemple : *FiltreOBJ*, *FiltreWRL*, *FiltrePGN*,...). Notre *parser* est donc simplement défini dans une fonction *readFileOBJ* de la classe Model. Elle prend en entrée le nom du fichier *.obj* à parser et construit le maillage en mémoire.

```
# Exemple de fichier OBJ : un cube

v -1.0 -1.0 -1.0
v -1.0 1.0 -1.0
v 1.0 1.0 -1.0
v 1.0 -1.0 -1.0
v -1.0 -1.0 1.0
v -1.0 1.0 1.0
v 1.0 1.0 1.0
v 1.0 -1.0 1.0
# 8 vertices

vt  0.000000 0.000000 0.000000
vt  1.000000 0.000000 0.000000
vt  0.000000 1.000000 0.000000
vt  1.000000 1.000000 0.000000
vt  0.000000 0.000000 0.000000
vt  1.000000 0.000000 0.000000
vt  0.000000 1.000000 0.000000
vt  1.000000 1.000000 0.000000
vt  0.000000 0.000000 0.000000
vt  1.000000 0.000000 0.000000
vt  0.000000 1.000000 0.000000
vt  1.000000 1.000000 0.000000
# 12 texture vertices

vn  0.000000 0.000000 -1.570796
vn  0.000000 0.000000 -1.570796
vn  0.000000 0.000000 -1.570796
vn  0.000000 0.000000 -1.570796
vn  0.000000 -0.000000 1.570796
vn  0.000000 -0.000000 1.570796
vn  0.000000 -0.000000 1.570796
vn  0.000000 -0.000000 1.570796
# 8 vertex normals

f 1/11/4 2/9/2 3/10/1 4/12/3
f 6/12/8 5/11/7 8/9/5 7/10/6
f 2/8/6 6/7/5 7/5/1 3/6/2
f 5/4/8 1/3/6 4/1/2 8/2/4
f 4/8/7 3/7/8 7/5/4 8/6/3
f 2/4/5 1/3/7 5/1/3 6/2/1
# 6 faces
```

FIG. 5.1 – Exemple de fichier OBJ : l'objet décrit est un cube centré en 0

Pour cela, on analyse chaque ligne du fichier. On stocke ainsi tous les sommets en mémoire grâce à un *vector* de pointeurs sur des objets Vertex. On fait de même pour les coordonnées des normales et pour celles des textures (si elles sont présentes dans le fichier). Pour les faces, plusieurs cas sont possibles selon que l'on dispose de ces informa-

tions ou non. Ainsi, une ligne du fichier définissant une face peut être écrite selon l'une des formes suivantes :

f	v/vt/vn	v/vt/vn	v/vt/vn	v/vt/vn ...
f	v//vn	v//vn	v//vn	v//vn ...
f	v	v	v	v

où :

- v désigne l'indice du vertex
- vt désigne l'indice du vertex-texture
- vn désigne l'indice du vertex-normal

Pour ne pas complexifier le *parser*, nous ne parsons que des fichiers définissant des faces en quadrilatère car notre application, étant basée sur la subdivision de Catmull-Clark, ne s'applique qu'à des maillages quadrangulaires. Ceci limite, certes, le nombre de fichiers que nous pouvons parser mais il n'était de toute façon pas utile de stocker des maillages polygonaux quelconques que nous n'aurons pas pu subdiviser par la suite. Les faces sont donc stockées en mémoire dans un *vector* de pointeurs sur des objets Face qui contiennent chacun un *vector* d'indices vers les positions des sommets la définissant dans le *vector* de pointeurs de Vertex.

Une fois le fichier parsé, le maillage de contrôle est donc défini et l'objet 3D peut donc être visualisé et subdivisé.

Afin d'accentuer le fait que notre application subdivise un objet en temps réel, nous avons donc décidé d'animer nos objets 3D. Pour cela, nous parsons plusieurs fichiers *.obj* qui correspondent chacun à une position de l'objet à un moment donné. Ainsi, les maillages de contrôle de chaque image de l'animation sont stockés dans un *vector* de pointeurs sur des objets Model. A chaque appel à la fonction *display*, on récupère un Model différent que nous subdivisons.

5.3 Structure des VBOs

L'idée de base des VBOs est de fournir des zones mémoires (*buffers*) accessibles pas le biais d'identifiants. Un *buffer* devient actif par une opération de *binding* de manière similaire à d'autres procédés OpenGL comme les *display list* et les textures. Ils définissent le type d'usage des *buffers*. Cela permet, entre autres, aux *drivers* graphiques d'optimiser la gestion de la mémoire interne et de choisir le type de mémoire le plus adapté (mémoire cache, mémoire graphique) dans laquelle les *buffers* seront stockés.

5.3.1 Présentation des fonctions

L'opération de *binding* convertit chaque pointeur de la fonction *client-state* en *offsets* relatif au *buffer* courant. En d'autre terme, cette opération change une fonction *client-state* en une fonction *server-state*.
Si les extensions sont présentes [1], on peut générer les *buffers ARB* voulus. Il suffit de faire appel à la fonction *glGenBuffersARB()*, de sélectionner le *buffer* à remplir avec *glBindBufferARB()* puis de remplir sa mémoire avec *glBufferDataARB()*. Ceci copie automatiquement les informations des *buffers* de la *RAM* vers la mémoire vidéo.

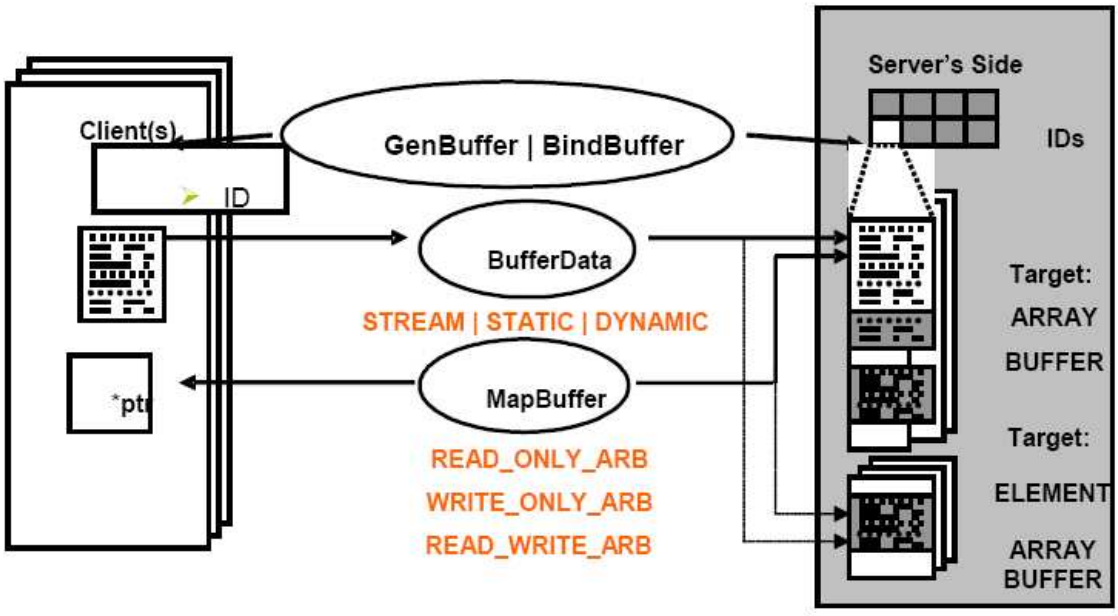

FIG. 5.2 – Description des VBOs [2]

Les fonctions clés suivantes sont utilisées [2] [15] :

- ***glGenBuffersARB() :*** retourne les identifiants des *buffer objects* non utilisés. Requière deux paramètres : le nombre de *buffers* à générer et l'adresse d'une variable GLuint où stocker l'identifiant ID.

void glGenBuffersARB(GLsizei n, GLuint ids)*

- ***glBindBufferARB() :*** une fois que l'ID a été générée, il faut pouvoir lier le VBO avec l'ID correspondant avant de l'utiliser. *glBindBufferARB()* prend deux paramètres : *target* et *ID*.

void glBindBufferARB(GLenum target, GLuint id)

  target indique si le VBO stocke les données du tableau de vertex ou les indices des polygones : GL_ARRAY_BUFFER_ARB ou GL_ELEMENT_ARRAY_BUFFER_ARB. A noter que le paramètre *target* permet au VBO de considérer le lieu de stockage le plus efficace des *buffers*, par exemple, préconise le stockage des indices dans l'AGP ou la mémoire système, et les vertices dans la mémoire vidéo.

- ***glBufferData() :*** contrôle la taille du *buffer* de donnée, la manière d'utiliser ses informations et autorise la copie dans un *buffer*.

void glBufferDataARB(GLenum target, GLsizei size, const void data, GLenum usage)*

  Une fois encore le premier paramètre *target* peut être GL_ARRAY_BUFFER_ARB ou GL_ELEMENT_ARRAY_BUFFER_ARB. *size* est la taille des données à transférer. Le troisième paramètre est un pointeur sur le tableau de données sources. Le dernier paramètre *usage* indique au VBO comment le *buffer* doit être utilisé : *static*, *dynamic*, *stream*, ...

Pour ce dernier paramètre, neuf valeurs sont autorisées :

GL_STATIC_DRAW_ARB
GL_STATIC_READ_ARB
GL_STATIC_COPY_ARB
GL_DYNAMIC_DRAW_ARB
GL_DYNAMIC_READ_ARB
GL_DYNAMIC_COPY_ARB
GL_STREAM_DRAW_ARB
GL_STREAM_READ_ARB
GL_STREAM_COPY_ARB

STATIC signifie que les données dans le VBO ne changent pas. DYNAMIC que les données changent fréquemment et STREAM qu'elles changent à chaque image. DRAW signifie que les données sont envoyées au GPU pour l'affichage (de l'application vers le rendu GL), READ qu'elles seront lues pas le client (de GL vers l'application) et COPY qu'elles sont utilisées à la fois pour l'affichage et pour la lecture.

Pour les VBOs seul DRAW est utile, COPY et READ n'ont de sens qu'avec l'utilisation d'un *pixel/frame buffer object* (PFO, FBO).

– ***glDeleteBuffersARB() :*** les VBOs sont supprimés quand ils ne sont plus utilisés.

void glDeleteBuffersARB(GLsizei n, const GLuint ids)*

5.3.2 Mise en œuvre dans l'application

Plusieurs structures de VBO ont été testées pour parvenir à la plus performante. L'élément de comparaison le plus évident étant le taux de *FPS*, il apparaît crucial de choisir la méthode de rendu la plus appropriée. La démarche pour atteindre cette version optimum est expliquée dans les paragraphes suivants.

Première version

La première version du programme utilisait une structure avec trois VBOs, un pour les coordonnées, un pour les normales et un pour les couleurs. Lors de l'initialisation, il suffisait alors d'appeler trois fois les fonctions *glGenBuffersARB*, *glBindBufferARB*, *glBufferDataARB* avec le bon identifiant ID du VBO. Les trois *buffers* étant de tailles identiques, les appels aux fonctions s'effectuaient simplement avec :

```
glGenBuffersARB( 1, &vbo[0] );
glBindBufferARB( GL_ARRAY_BUFFER_ARB, vbo[0] );
glBufferDataARB( GL_ARRAY_BUFFER_ARB, sizeof(GLfloat) * size_of_buffer,
                 buffer[0], GL_STATIC_DRAW_ARB );
glVertexPointer( 3, GL_FLOAT, sizeof(GLfloat), 0 );
```

où buffer[0] est le tableau de coordonnées, buffer[1] celui des normales et buffer[2] celui des couleurs.
Ainsi pour spécifier le *buffer* des normales et celui des couleurs, la fonction *glVertexPointer()* est remplacer par les fonctions *glColorPointer()* et *glNormalPointer()*.

Lors de l'affichage, une fois les trois *buffers* activés par les différents appels de fonction *glEnableClientState()*,

```
glEnableClientState( GL_COLOR_ARRAY );
glEnableClientState( GL_NORMAL_ARRAY );
glEnableClientState( GL_VERTEX_ARRAY );
```

la commande d'affichage était :

```
glBegin (GL_QUADS);
  for(int i = 0 ; i < size_of_buffer ; i++)
     glArrayElement (i);
glEnd() ;
```

Il fallait ensuite désactiver les *buffers* avec :

```
glDisableClientState( GL_COLOR_ARRAY );
glDisableClientState( GL_NORMAL_ARRAY );
glDisableClientState( GL_VERTEX_ARRAY );
```

Très vite, cette première approche c'est avérée inefficace. En effet, les performances obtenues étaient très en dessous de celles annoncées, notamment en comparaison avec les *display lists*. Les points négatifs sont malgré tout parfaitement identifiables. La version suivante les corrige pour la plupart. La source la plus évidente de perte de performances est l'utilisation d'un trop grand nombre de VBOs. En effet, le GPU doit perpétuellement passer d'un VBO à l'autre occasionnant ainsi une perte importante de temps d'affichage. Ensuite, le *buffer* stockant la couleur de chaque sommet n'apparaît pas capital et sera abandonné. Enfin, l'utilisation d'un *buffer* stockant les indices des sommets pour chaque face permettra de réduire la taille du tableau de sommet et d'en supprimer les redondances.

Deuxième version

Dans cette deuxième version, trois *buffers* sont encore nécessaires mais, contrairement à la première version, on compte deux *buffers* GL_ARRAY_BUFFER_ARB pour les coordonnées des sommets et les normales et un *buffer* GL_ELEMENT_ARRAY_BUFFER_ARB utilisé pour les indices. Au niveau de l'initialisation, il suffit juste de préciser lors de l'opération de *binding* s'il s'agit d'un *index buffer* ou d'un *vertex buffer*.

```
glBindBufferARB( GL_ELEMENT_ARRAY_BUFFER_ARB, IndexBuffer );
```

La conséquence principale de l'utilisation d'un buffer d'indices est visible dans la fonction d'affichage par l'appel de la fonction :

```
glDrawElements( GL_QUADS, nb_faces*4, GL_UNSIGNED_INT,  0);
```

Ainsi les polygones seront dessinés en considérant les quatre indices consécutifs de l'*index buffer* qui référence les sommets du *vertex buffer*. Les résultats obtenus se révèlent bien supérieurs à ceux de la première version. Les performances peuvent néanmoins être accrues en n'utilisant plus que deux VBOs : un pour les indices et un contenant à la fois les coordonnées des sommets et les normales.

Troisième version

La version finale repose essentiellement sur la deuxième version c'est-à-dire en utilisant un *index buffer*. Pour éviter des changements trop fréquents de VBO, les tableaux de coordonnées et de normales sont fusionnés. Il faut alors préciser lors de l'initialisation des VBOs le décalage à utiliser pour différencier les normales des coordonnées. Ici le tableau de normales est simplement concaténé à celui des sommets, ainsi le décalage (ou *offset*) est égal à la taille du tableau de sommets. Dans l'entête du fichier, on définit la macro BUFFER_OFFSET qui sert à créer un *offset* pour la lecture des informations dans la structure. Elle a la forme suivante :

```
#define BUFFER_OFFSET(i) ((char *)NULL + (i))
```

On peut dès lors l'utiliser de la manière suivante lors de l'initialisation :

```
glVertexPointer( 3, GL_FLOAT, sizeof(VERTEX), 0 );
glNormalPointer( GL_FLOAT, sizeof(GL_float), BUFFER_OFFSET(sizeof(GLfloat) *
```

La fonction d'affichage reste la même que celle de la deuxième version par l'appel de la fonction *glDrawElements()*.

5.4 Algorithme de subdivison de Warren/Schaeffer

Le but de cet algorithme est de réaliser une subdivision de type Catmull-Clark sur un maillage polygonal. Dans le cadre de notre sujet, nous avons réduit son utilisation à des faces comportant quatre sommets. Pour ce faire, nous utilisons la méthode mise en place par Joe D. Warren et Scott Schaeffer [30]. Celle-ci se compose de deux phases :

- Une phase de subdivision simple où chaque face est découpée, divisée en plusieurs faces.
- Une phase de moyennage, où les points du nouveau maillage sont déplacés afin d'obtenir un aspect lissé.

5.4.1 Subdivision linéaire du maillage

Avant d'exécuter cet algorithme, le maillage est structuré de la manière suivante : nous avons d'un côté une liste de tous les sommets, chacun ayant ses caractéristiques propres (position, valence...) et d'autre part nous avons l'ensemble des faces du maillage, chacune définie par une liste d'indices faisant référence aux sommets qui la compose dans la liste des sommets citée ci-dessus. Cette première phase de l'agorithme consiste en fait à diviser chaque quad en quatre quads. La difficulté repose donc sur le fait que lorsqu'une des arêtes d'un quad est divisée en deux arêtes, le nouveau point créé ne doit pas être recréé lors de la subdivision de la face voisine. Pour palier à ce problème, l'utilisation d'une table de hachage est alors nécessaire. Durant l'implémentation de cette partie, nous avons d'abord utilisé une table de hachage simplifiée, composée de deux vecteurs, l'un comportant les clés et l'autre les valeurs associées. Mais les performances étaient bien en deçà de nos attentes. Nous avons donc utilisé les *map* de la STL. L'amélioration en terme de performances fut alors très importante.

L'algorithme de subdivision linéaire est le suivant :

pour chaque polygone composé des indices $\{s_1,\ s_2,\ s_3,\ s_4\}$ on vérifie si le sommet sur l'arête $\{s_i,\ s_{i+1}\}$ est dans la table de hachage. Si tel est le cas, on utilise l'indice stocké dans la table. Sinon, on insère un nouveau sommet au maillage et on ajoute son indice à la table de hachage avec la clé $\{v_i,\ v_{i+1}\}$ (on appellera ce nouveau sommet e_i). On ajoute un sommet, c, situé au centre du polygone, dans la liste des sommets.
Enfin on crée le nouveau polygone formé des sommets $\{e_i,\ v_i,\ e_{i+1},\ c\}$. On obtient donc une nouvelle liste de sommets et un nouvel ensemble de faces.

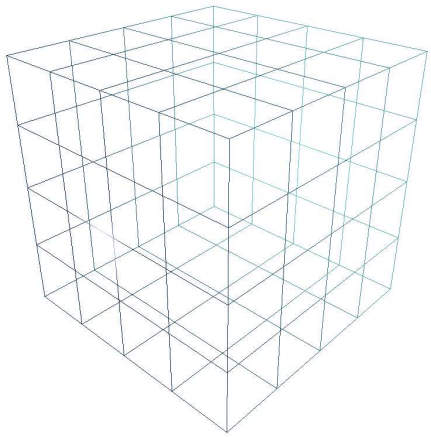

FIG. 5.3 – Cube après deux passes de subdivision linéaire.

5.4.2 Lissage du maillage

On initialise une nouvelle liste de points, de la même taille que celle obtenue à l'issue de la phase précédente, mais dont les coordonnées sont à $\{0, 0, 0\}$. Pour chaque quad q, on calcule le centroïde de q et on ajoute sa valeur aux quatre entrées indexées par les sommets de q. Après l'avoir fait pour toutes les faces du maillage, on divise chaque entrée par la valence du point associé.

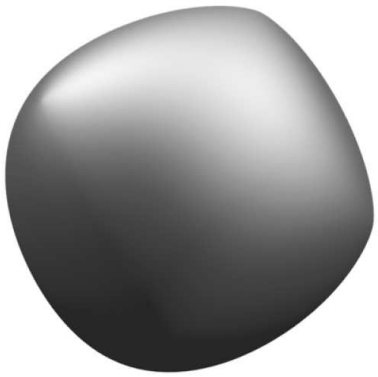

FIG. 5.4 – Subdivision d'un cube avant correction du lissage (extrait de [30]).

A ce stade le maillage est subdivisé, lissé et la surface obtenue est C^2 excepté autour des points extraordinaires. En effet, comme on peut le voir sur le cube subdivisé, alors que la surface parait bien lissée sur sa majeure partie, on remarque de fortes variations de son apparence au niveau des points de valence 3, où la surface n'est que C^1. Pour remédier à ce problème de continuité, on applique une dernière passe de correction. Soit

p un sommet après la phase de subdivision linéaire et p' sa position après la passe de lissage. On repositionne les sommets du maillage à la position suivante :

$$p' + w(n)(p - p')$$

où n est la valence du point et $w(n)$ une fonction de correction. Ici, nous ne traitons que des quads : la fonction de correction est ainsi $w(n) = \frac{4}{n}$. En utilisant cette rectification, nous obtenons une surface exempte des discontinuités sus-citées.

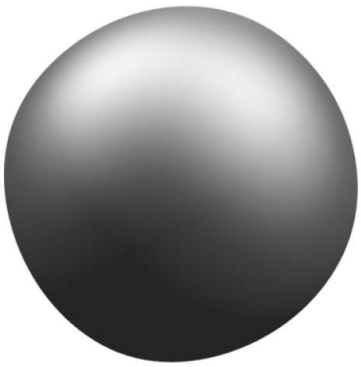

FIG. 5.5 – Subdivision d'un cube après correction (extrait de [30]).

5.5 Algorithme de subdivison sur GPU

Nous nous sommes basés sur le noyau de subdivision proposé par Le-Jeng Shiue, Ian Jones et Jorg Peters présenté lors du Siggraph 2005. Cet algorithme se fait en deux grandes phases : une première étape est précalculée par le CPU, le reste de l'algorithme étant ensuite effectué sur le GPU en temps réel, à chaque image affichée.

5.5.1 Précalculs effectués sur le CPU

Une ensemble de tâches est d'abord réalisée par le CPU pour préparer la subdivision par le GPU. La première consiste à effectuer un première passe de subdivision au maillage de contrôle, ceci afin d'isoler les points irréguliers par une ceinture de sommets réguliers. Le maillage obtenu est alors découpé en morceaux ou *fragment meshes* (cf. figure 5.8). Chaque *fragment mesh* est un ensemble de faces composé d'un point du maillage initial entouré de deux couronnes de sommets.

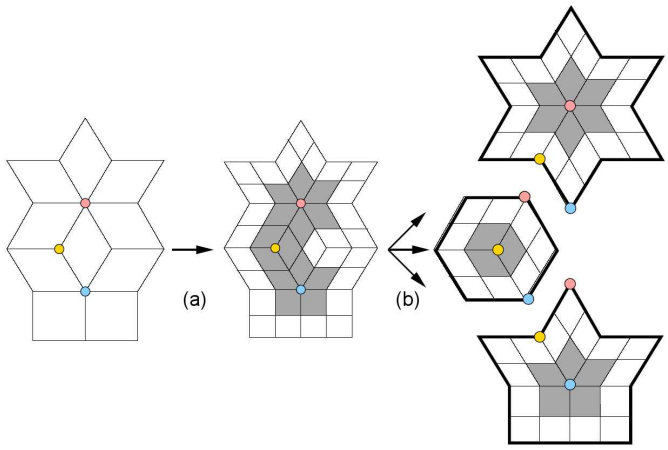

FIG. 5.6 – Découpage en *fragment meshes* (extrait de [28]).

Les sommets d'un fragment mesh sont alors stockés dans vecteur unidimensionnel (cf. figure 5.7). Ce dernier doit être construit par un parcours en spirale du *frag mesh* depuis son centre jusqu'à son dernier point de la deuxième couronne. Pour réaliser cela, nous nous sommes servis d'une propriété simple des maillages polygonaux : les faces sont orientées et cette orientation est définie par l'ordre des indices des points de la face. En partant du sommet central, on ajoute chaque point du *frag mesh* rencontré et ce tant qu'il n'est pas déjà dans la texture. Si tel est le cas, on passe à la face suivante, à savoir celle située de l'autre côté de l'arête formée par ce point non ajouté et par le dernier point ajouté au vecteur.

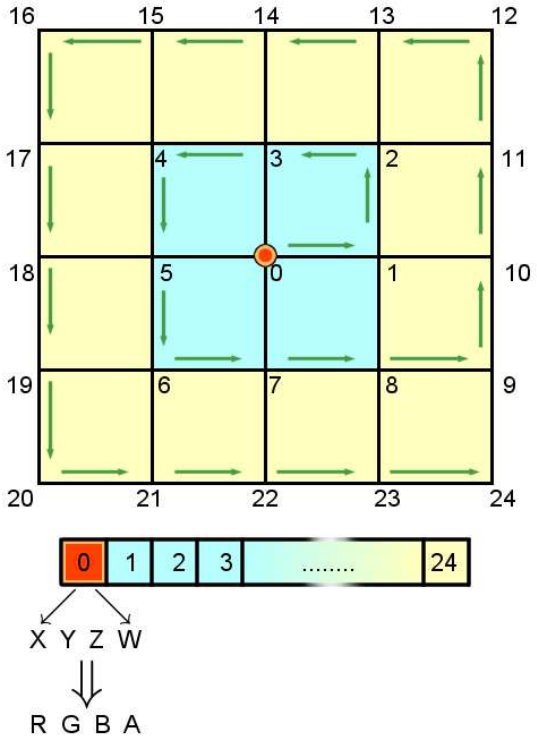

FIG. 5.7 – Exemple de construction d'une *patch-texture* pour un sommet de valence 4.

Le vecteur ainsi créé peut alors être vu comme une texture 1D, appelée *patch-texture*, où chaque coordonnée RGBA du pixel représente en fait les coordonnées x, y, z, w d'un point du *frag mesh*. Dans ce cas, les coordonnées de texture correspondent à l'indice pour accéder à un sommet.

5.5.2 Noyau de subdivision sur GPU

Le but ici va être de générer les sommets du *frag mesh* au niveau de subdivision suivant et donc la *patch texture* lui étant associée.
Pour avoir accès au *shader*, on crée un un quad de la taille du *viewport*, ce dernier ayant pour largeur la taille de la *patch-texture* à générer et pour hauteur 1 pixel. Ceci de manière à avoir un texel par pixel. A l'issue de chaque passe de raffinement, le *viewport* est ainsi élargi à la taille de la nouvelle *patch-texture* calculée. Cette taille, s, peut être connue car elle ne dépend que de la valence du sommet central du *fragment mesh* et du niveau de subdivision :

$$s_{n,d} = nq(q-1)+1 \text{ pixels, avec } q = 2^{d-1}+2$$

Pour chaque pixel, le *fragment shader* récupère les indices nécessaires au calcul du nouveau pixel grâce à la *lookup table*. Cette dernière est en fait une texture RGBA de 3 lignes de haut et de même longueur que la *patch-texture* que l'on souhaite calculer. On retrouve dans cette table dans chaque colonne i les indices des points de la patch-texture au niveau d nécessaires au calcul du pixel i de la patch texture au niveau $d+1$. On peut par ailleurs noter deux points importants concernant la *lookup table* :

- elle est précalculée une fois pour toutes par le CPU pour le niveau maximum de subdivision souhaité, les *lookup tables* des niveaux inferieurs étant incluses dans cette dernière.

- elle est spécifique à la valence du *frag mesh* : il faut donc une *lookup table* par valence.

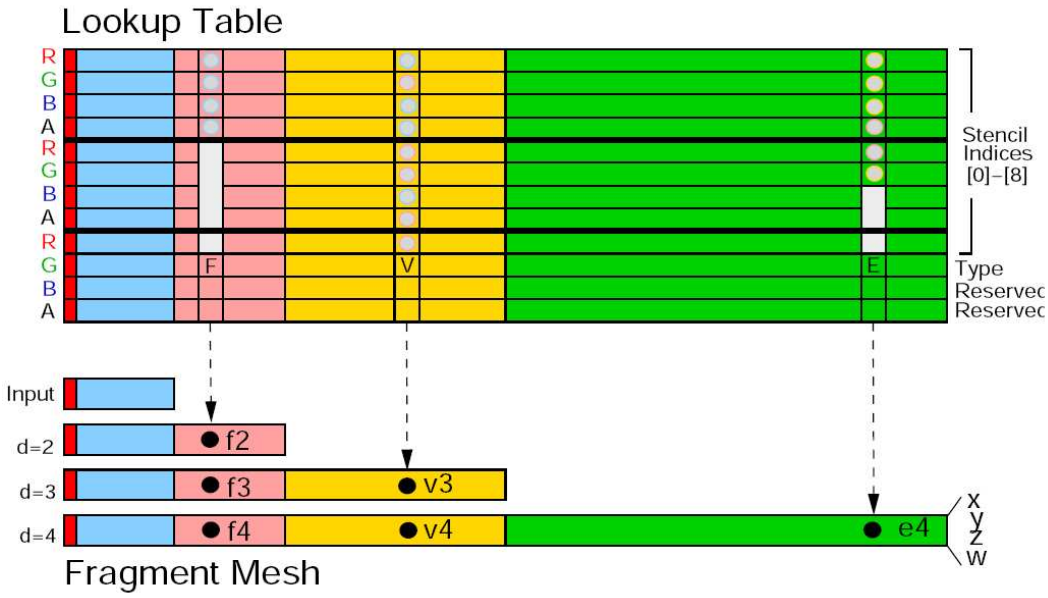

FIG. 5.8 – Fonctionnement de la *lookup table* (extrait de [28]).

On passe donc en entrée au *fragment shader* cette *lookup table* ainsi que notre *patch-texture* et on peut ainsi générer un nouveau pixel, autrement dit un nouveau point pour notre maillage. Voici le *fragment program* exécuté par le *fragment shader* pour créer ce nouveau pixel :

```
// The input patch-texture
uniform sampler2D InputPatch ;

// The lookup table texture
uniform sampler2D LookUp ;

// Texture coordinates toaccess lookup table
varying vec2 LookupTC ;

// Subdivision stencil type

#define FACE_NODE 1
#define EDGE_NODE 2
#define VERTEX_NODE 4

// Transforation from index to texture coordinate
#define IDX2TC ( 1 . 0 / 2 0 4 8 . 0 )
void main ( void ) {   // Collect  the lookup table entry
vec4 rgba1 = texture2D ( LookUp , vec2 ( LookupTC.s , 0 . 0 / 3 . 0 ) ) * IDX2TC ;
vec4 rgba2 = texture2D ( LookUp , vec2 ( LookupTC.s , 1 . 0 / 3 . 0 ) ) * IDX2TC ;
vec4 rgba3 = textur e2D ( LookUp , vec2 ( LookupTC.s , 2 . 0 / 3 . 0 ) ) ;
int type = int ( rgba3.g ) ;
rgba3 = rgba3 *IDX2TC ;
```

```
// Collect the stencil nodes in the input patch-texture

vec4 S[9] ;
S[0] = texture2D ( InputPatch , vec2 ( rgba1.r, 0 ) ) ;
S[1] = texture2D ( InputPatch , vec2 ( rgba1.g, 0 ) ) ;
S[2] = texture2D ( InputPatch , vec2 ( rgba1.b, 0 ) ) ;
S[3] = texture2D ( InputPatch , vec2 ( rgba1.a, 0 ) ) ;
S[4] = texture2D ( InputPatch , vec2 ( rgba2.r, 0 ) ) ;
S[5] = texture2D ( InputPatch , vec2 ( rgba2.g, 0 ) ) ;
S[6] = texture2D ( InputPatch , vec2 ( rgba2.b, 0 ) ) ;
S[7] = texture2D ( InputPatch , vec2 ( rgba2.a, 0 ) ) ;
S[8] = texture2D ( InputPatch , vec2 ( rgba3.r, 0 ) ) ;

// Compute the position using the vertex-stencil
vec4 v e r t P o s = S[0] * 9 . 0 / 1 6 . 0 +
( ( S[1]+ S[2] ) + ( S[3]+ S[4] ) ) * 3 . 0 / 3 2 . 0 +
( ( S[5]+ S[7] ) + ( S[8]+ S[6] ) ) / 6 4 . 0 ;

// Compute the position using the edge-stencil
vec4 edgePos = ( S[0]+ S[1] ) * 3 . 0 / 8 . 0 +
( ( S[2]+ S[4] ) + ( S[3]+ S[5] ) ) / 1 6 . 0 ;

// Compute the position using the face-stencil
vec4 facePos = ( ( S[0]+ S[2] ) + ( S[1]+ S[3] ) ) / 4 . 0 ;

// Assign the valid position by numerical masking
gl FragColor =vertPos * float ( type == VERTEX_NODE) +
    edgePos * float ( type == EDGE_NODE) +
    facePos * float ( type == FACE_NODE ) ;
}
```

Une nouvelle *patch-texture* correspondant à une nouvelle subdivision du *frag mesh* initial est ainsi créée. Cette dernière peut alors être replacée en entrée du *fragment shader* pour la subdivision suivante. A noter qu'à chaque subivision sur GPU, les sommets de la bordure exterieure du *frag mesh* sont supprimés : plus on subdivise, plus le maillage obtenu tend vers la première couronne.

La *patch-texture* finalement obtenue contient les coordonnées des points du *fragment mesh* subdivisé au niveau souhaité. Il ne reste donc plus qu'à afficher les quads formés par ces nouveaux sommets pour voir apparaître notre maillage subdivisé. Bien évidemment ce procédé doit être appliqué à l'ensemble des *frag meshes* pour obtenir une subdivision complète de l'objet.

Chapitre 6

Présentation de l'application

Dans ce chapitre, nous allons décrire les principes et les fonctionnalités de notre application de subdivision temps réel de Catmull-Clark sur CPU.
Les structures de base ayant été décrites précédemment, nous ne reviendrons pas dessus dans cette partie.

6.1 Principe général du programme

Le principe général de notre programme est le suivant :

- parser un ou plusieurs fichiers OBJ (si on souhaite visualiser une animation) afin de stocker le(s) maillage(s) de contrôle en mémoire.

- visualiser le maillage subdivisé en temps réel (par le CPU) ie que le maillage de contrôle est subdivisé à chaque *frame* de l'application. Le niveau de subdivision est variable et défini par l'utilisateur à l'aide d'un menu décrit ci-dessous.
  Si on est dans le cas d'une animation, on change également de maillage de contrôle (représentant la position de l'objet à un moment donné) à chaque *frame*.

6.2 Fonctionnalités du programme

Afin d'interagir avec l'utilisateur, nous avons développé certaines fonctionnalités pour notre application.

6.2.1 Changer le niveau de subdivision

Comme expliqué précédemment, le niveau de subdivision est variable et défini par l'utilisateur.
On peut ainsi visualiser, si le taux de FPS le permet, un maillage subdivisé de 1 à 5 fois mais aussi le maillage de contrôle.

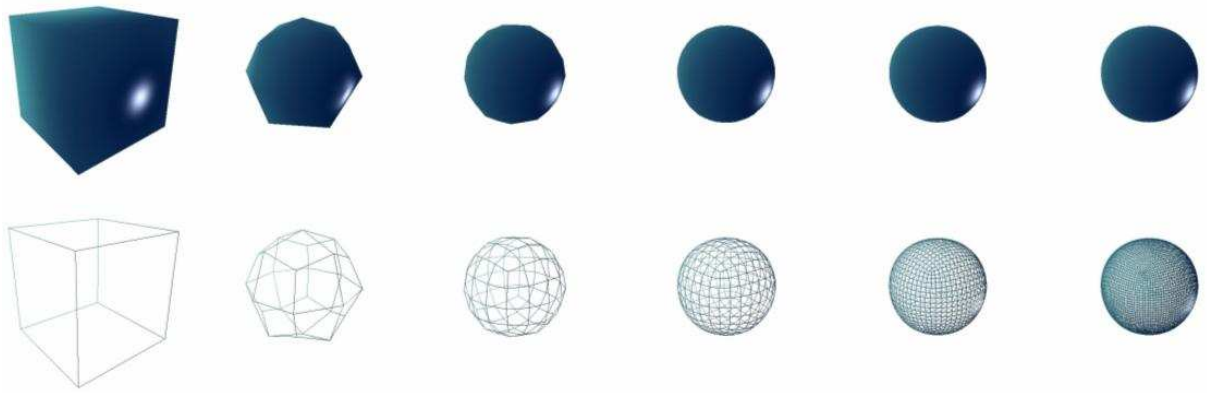

FIG. 6.1 – Exemple d'un cube subdivisé : de gauche à droite, le maillage de contrôle puis les 5 subdivisions successives

6.2.2 Changer le mode de visualisation

On dispose de 3 modes de visualisation : classique (objet plein), mode fil de fer ou visualisation uniquement des sommets.
La visualisation en mode fil de fer a tendance à faire chuter le nombre de FPS mais est la plus intéressante car elle permet de visualiser les effets de la subdivision.

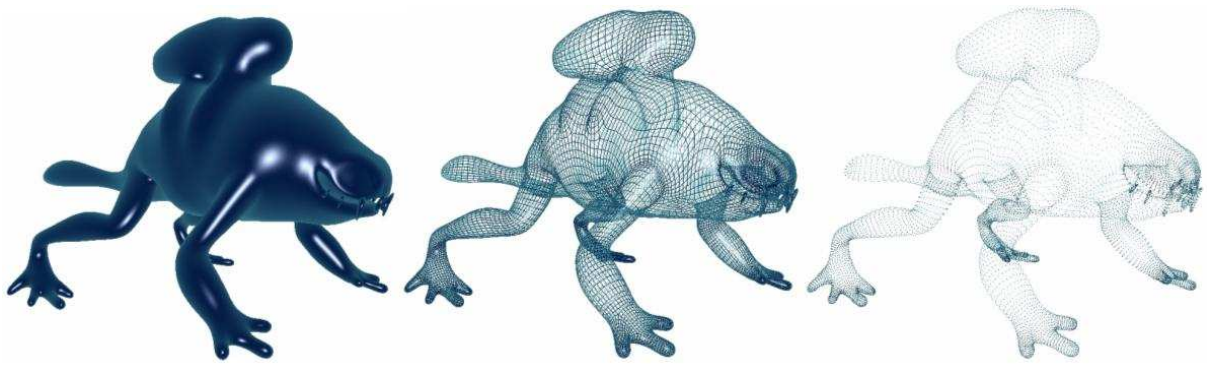

FIG. 6.2 – Modes de visualisation disponibles : objet plein, wireframe et points (modèle OBJ extrait de [10])

6.2.3 Utiliser des normales ou non

On peut utiliser des normales si elles ont été définies dans le fichier de départ ou visualiser l'objet sans normales. L'utilisation de ces dernières permet un meilleur rendu visuel (calcul d'éclairage notamment) mais le mode sans normales permet de ne pas les stocker dans le VBO qui contient par conséquent beaucoup moins d'informations.
On obtient donc un meilleur taux de FPS sans normales. Cependant, ceci est surtout le cas pour le maillage de contrôle ou pour la première subdivision. En effet, lorsque le nombre de polygones augmente, le taux de FPS devient relativement faible et le facteur de ralentissement n'est plus la taille du VBO mais la subdivision en temps réel. Le taux de rafraîchissement de l'image n'est donc, dans ce cas, plus influencé par l'utilisation des normales.

FIG. 6.3 – Exemple de visualisation avec ou sans normales : à gauche, les normales sont utilisées, à droite non (modèle OBJ extrait de [10])

6.2.4 Animer ou stopper l'animation d'un objet

On peut stopper ou relancer l'animation d'un objet si on dispose de plusieurs maillages de contrôle décrits dans plusieurs fichiers OBJ et définissant chacun la position de l'objet à un moment donné.
Pour le maillage de contrôle et pour les premières subdivisions, il se peut que visualiser un objet animé ne rende pas un résultat visuel très agréable car le nombre de FPS restant relativement élevé, les images sont affichées très rapidement. A contrario, pour un nombre élevé de subdivisions, le taux de FPS étant relativement faible, l'animation n'est plus fluide et le rendu s'en trouve altéré.

6.2.5 Changer le *shader* utilisé pour la visualisation

Lors de nos recherches sur le langage des *shaders*, nous avons du apprendre à les incorporer dans un code OpenGL afin de pouvoir les utiliser notamment dans l'application de subdivision sur le GPU.
Nous avons donc recherché quelques *shaders* que nous avons modifiés puis incorporés à notre application afin d'améliorer le rendu visuel grâce notamment à des calculs d'éclairage. Nous disposons ainsi de 3 *shaders* différents permettant tout autant de modes d'éclairage à savoir un *gooch shading*, un *toon shading* et un *per-pixel lighting*.
Chaque *shader* à un temps de calcul d'éclairage qui lui est propre ce qui a pour conséquence de faire varier le taux de FPS selon le *shader* utilisé.

FIG. 6.4 – *Shaders* utilisés pour la visualisation. De gauche à droite : *Gooch-Shading*, *Toon-Shading* et *Per-Pixel Lighting* (modèle OBJ extrait de [10])

6.2.6 Faire une capture d'image (*screenshot*) de l'application

On peut en faire autant que l'on veut lors de la même utilisation mais il faut savoir qu'entre chaque utilisation les fichiers de sortie seront écrasés. Les images générées sont au format *PPM* qui est l'un des formats graphiques les plus simples mais qui a le désavantage d'être lourd en terme de place mémoire (les fichiers peuvent faire plusieurs Mo !).
Cette fonctionnalité, non nécessaire, nous a été surtout utile pour faire des *screenshots* de notre application très facilement. Nous avons décidé de la garder puisque nous l'avions implémentée mais son utilisation fait d'autant plus ralentir l'application que la fenêtre de visualisation est grande.

Toutes ces fonctionnalités sont accessibles par un menu grâce au bouton droit de la souris mais aussi avec des raccourcis clavier.

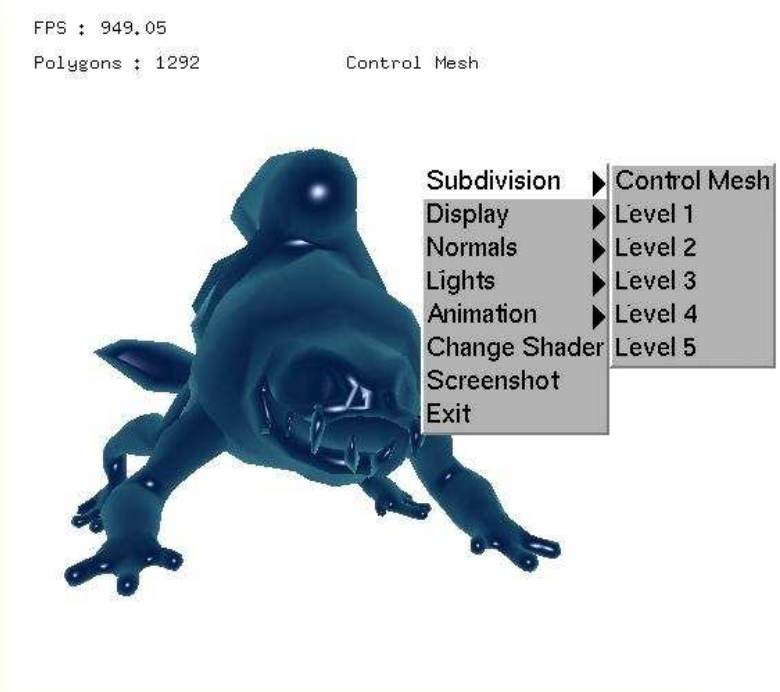

FIG. 6.5 – Menu de l'application de subdivision temps réel sur CPU accessible par le bouton droit de la souris (modèle OBJ extrait de [10])

D'autres interactions sont également possibles comme tourner ou translater l'objet. Les translations sur le plan (Oxy) s'effectuent grâce aux touches Z-Q-S-D.

La translation selon l'axe (Oz) et les rotations s'effectuent respectivement en maintenant le bouton du milieu ou le bouton gauche de la souris enfoncé.

Chapitre 7

Résultats obtenus

7.1 Résultats obtenus sur CPU

Cette partie est consacrée aux résultats obtenus lorsque la subdivision est effectuée entièrement sur le CPU. Les tests effectués sont résumés dans le tableau ci-dessous (cf. figure 7.2). Ils ont été réalisés sur 3 modèles 3D choisis pour leur variété de complexité :

- un cube : forme simple composée de 8 sommets et de 6 faces
- un noeud : de complexité moyenne avec 96 polygones (primitive de base de 3DSMAX)
- une grenouille : objet plus élaboré composé de 1292 polygones (modèle OBJ extrait de [10])

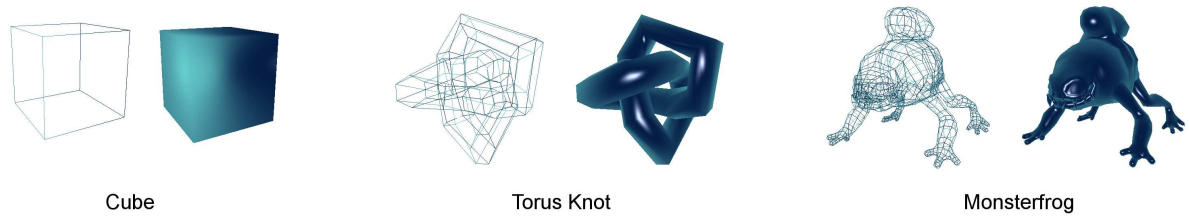

FIG. 7.1 – Modèles 3D utilisés pour les comparaisons

Comme on peut le voir sur le graphique 7.3, les performances chutent fortement avec le niveau de subdivision. On peut remarquer qu'à chaque subdivision le nombre de polygones est multiplié par 4 et le nombre d'images par seconde divisé par 4. Ici, le facteur limitant n'est pas l'affichage mais la subdivision en temps-réel. Par exemple, le *control mesh* de la *Torus Knot* composé de 96 polygones est affiché à plus de 1200 FPS alors que

	Control Mesh	Niveau 1	Niveau 2	Niveau 3	Niveau 4	Niveau 5
Cube	3271,3 6	960,1 24	276,2 96	68,7 384	16,7 1536	3,6 6144
Torus Knot	1529,4 96	90,4 384	17,2 1536	3,7 6144	0,9 24576	0,2 98304
Monsterfrog	1257,7 1292	5,8 5168	1,1 20672	0,3 82688	0,07 330752	0,02 1323008

FIG. 7.2 – Tableau récapitulatif des performances obtenues avec différents objets et pour différents niveaux de subdivision sur CPU (les FPS sont indiquées en vert et le nombre de polygones en bleu).

le cube subdivisé au niveau 2 ne l'est qu'à 276. Ceci est dû au fait que pour le maillage de contrôle, on n'effectue pas de subdivision en temps réel contrairement aux maillages raffinés. C'est donc bien cette dernière qui fait chuter sensiblement le *framerate* [1]. De plus, on peut remarquer que pour les maillages de contrôle de la *Torus Knot* et de la *Monsterfrog*, le nombre de FPS est du même ordre de grandeur (1257,7 pour la *Monsterfrog*, 1529,4 pour la *Torus Knot*) alors que le nombre de polygones des deux objets est très différent : 13 fois plus de polygones pour la *Monsterfrog* (1292 contre 96). Ceci accentue encore le rôle joué par la subdivision en temps réel sur la baisse des performances.

On en conclut donc que la subdivision de surfaces en temps réel effectuée sur le CPU ne permet pas de garantir un taux de FPS raisonnable pour un niveau de subdivision élevé. En effet, bien que le schéma de Catmull-Clark que nous avons implémenté est relativement simple et optimisé (utilisation de la structure de données *hashmap* de la STL, schéma de Catmull-Clark non adaptatif...), les performances obtenues après plusieurs subdivisions successives ne satisfont pas le critère du temps réel.

[1] nombre d'images par seconde

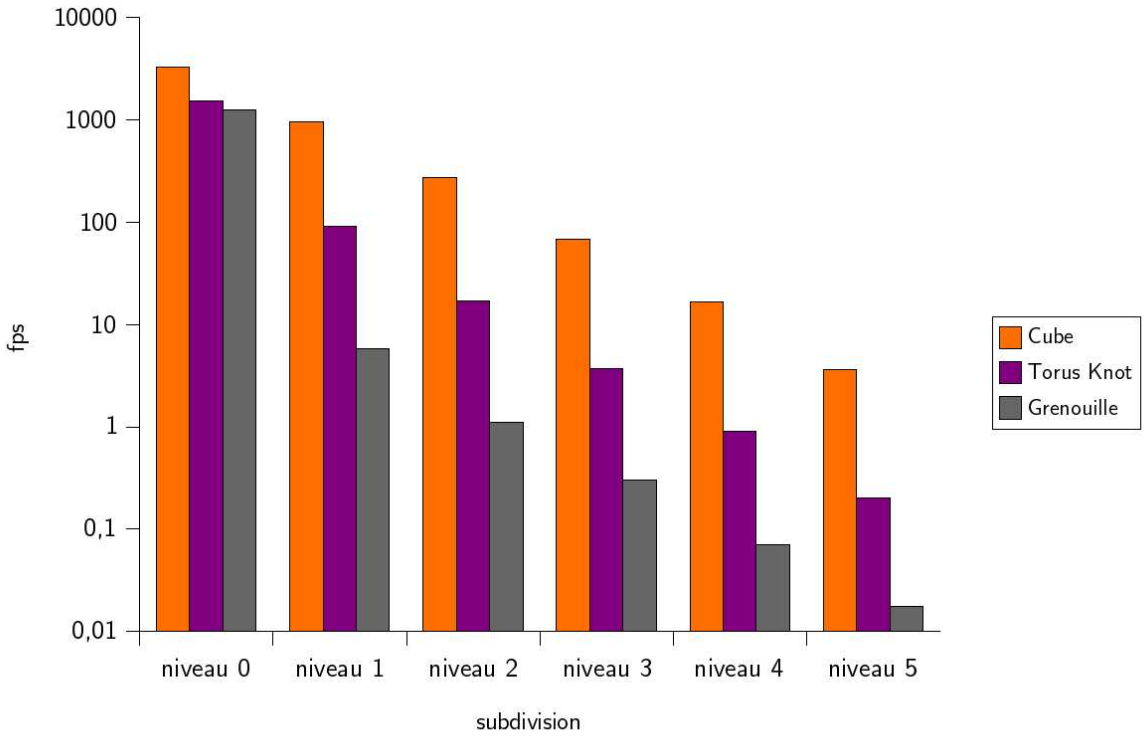

FIG. 7.3 – Graphique des performances obtenues sur CPU.

7.2 Résultats obtenus sur GPU et analyse de ces derniers

Nous allons maintenant essayer comparer les performances obtenues sur GPU et CPU. Essayer car malheureusement, nous n'avons pas pu compléter la partie sur GPU avant la remise de ce rapport, et ce pour plusieurs raisons :

- complexité du papier sur lequel s'appuie notre implémentation [28],
- installation sur une unique machine des *drivers* nécessaires à cette implémentation 10 jours avant la *deadline*.

Le temps nous manquant, nous nous sommes fixé comme objectif non pas de réaliser la subdivision sur l'ensemble d'un objet mais sur un unique *fragmesh* pour un niveau de subdivision 1. Dans le cadre de cette démarche nous avons utilisé un objet simple : une grille composée de 16 quads, le sommet central étant donc de valence 4 A.1. Cela nous

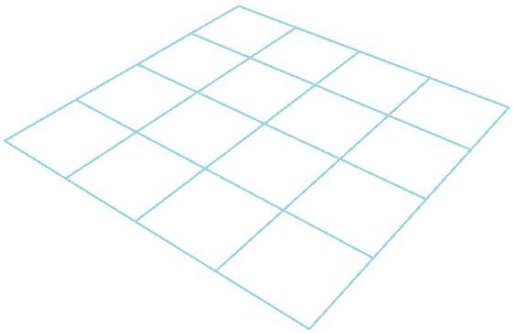

FIG. 7.4 – Objet utilisé pour notre test sur un *frag mesh*.

	Nb. de bolygones avant subdivision	**Nb. de polygones après subdivision**	**FPS**
CPU	16	36	743.5
GPU	9	36	1410.2

FIG. 7.5 – Performances obtenues sur GPU. Le passage de 16 à 36 et non 64 polygones est dû à l'élimination des sommets en bordure du *mesh*.

permet d'obtenir directement à partir du point central de cette géométrie un *fragmesh* avec ses deux couronnes de quads comme indiqué précédemment 5.8.

Les résultats obtenus sont présentés dans le tableau 7.5. Comme on peut le voir, sur une géométrie très simple le taux de FPS est pratiquement doublé. Ce résultat est d'autant plus encourageant que, comme nous l'avons montré lors du test sur le CPU, c'est bien la subdivision et non l'affichage qui limite le *framerate*. De plus, le niveau de subdivision utilisé ici n'est que 1. Enfin, on peut noter que dans les travaux de Le-Jeng Shiue, Ian Jones et Jorg Peters [28], l'écart de perfomances atteignait un facteur 20 sur un maillage de base à 40 polygones et sur une configuration où par rappport à celle que nous utilisons, la différence de puissance entre CPU et GPU était moins importante (processeur équivalent et carte graphique moins puissante que la notre).

On peut donc raisonnablement penser qu'avec des objets plus complexes et un niveau de raffinement plus élevé, la différence de performances entre subdivision en temps-réel sur CPU et GPU serait bien plus importante que celle obtenue ici.

Chapitre 8

Bilan

La subdivision de surfaces est un procédé de modélisation permettant d'obtenir des objets lissés à faible coût. Cependant, l'utilisation en temps réel de telles techniques pose encore des problèmes d'implémentation comme on l'a vu dans ce rapport. En effet, il est nécessaire de disposer de méthodes de calculs et d'affichage optimisées afin de cadrer aux exigences du temps réel notamment en terme de FPS.
De ce fait, de nouvelles techniques de programmation tendent à tirer profit de la puissance du GPU comparée à celle du CPU pour certains types de calculs spécifiques. Ainsi, il peut être judicieux de tirer partie de son architecture en pipeline pour des algorithmes pouvant être parallélisés comme c'est le cas de celui que nous devions implémenter.
A l'heure actuelle, les techniques de subdivisions de surfaces qui utilisent principalement le CPU parviennent malgré tout à obtenir des performances honorables sur des objets de complexité moyenne et ce en temps réel.
Cependant, le fait d'effectuer certains traitements sur le GPU et non plus sur le processeur central permet d'effectuer cette même subdivision sur des modèles beaucoup plus complexes en augmentant le *framerate*.

En ce qui concerne notre projet, la première partie correspondant à l'algorithme de subdivision sur CPU a été réalisée en accordant un grand soin à l'optimisation aussi bien celle concernant l'algorithme de Catmull-Clark en lui-même que la partie visualisation. Cette démarche a permis de mettre en évidence le travail important du CPU dans l'algorithme avec des performances qui décroissent en fonction du nombre de polygones du modèle de base et du niveau de subdivision appliqué.
La deuxième partie, utilisant principalement le GPU, n'a, elle, pu être menée totalement à son terme. Il n'est ainsi pas possible de charger un modèle quelconque étant donné que seule une *LookUp table* de niveau 1 de profondeur et s'appliquant à des sommets de valence 4 a été prévue. Ceci s'explique par les difficultés rencontrées tant au niveau de la compréhension de l'article que des problèmes liés à l'implémentation et la mise en œuvre de structures récentes et donc parfois faiblement documentées.

Malgré tout, les résultats obtenus semblent prometteurs et les modifications à apporter facilement identifiables. En effet, l'essentiel des composants nécessaires a été étudié et mis en place.

Par manque de temps, nous n'avons donc pas pu mener à bien ce projet. Cependant, de nombreuses fonctionnalités ont été étudiées et implémentées notamment les dernières extensions des cartes graphiques. En effet, le langage des *shaders*, les notions de *render-to-texture* avec FBO, *render-to-vertex-array* avec PBO et les VBOs sont issus de mises à jour très récentes des pilotes de cartes graphiques. Ainsi, pour pouvoir les utiliser, il est absolument nécessaire de posséder la dernière version disponible. Celle-ci n'ayant été fourni qu'en toute fin du projet, certaines parties de celui-ci n'ont pues être totalement optimisées. En nous familiarisant avec les techniques d'implémentation spécifiques au GPGPU comme le rendu multi-passes ou le *render-to-texture*, nous avons découvert de nouvelles approches utilisées dans les travaux de recherches les plus récents et vouées à investir massivement les applications commerciales dans un futur proche.

Des extensions à ce projet sont bien évidemment possibles. Tout d'abord, en ce qui concerne la généricité des modèles utilisés. En effet, il est possible de créer une structure qui s'adapte à la valence des sommets et au schéma utilisé (Catmull-Clark, Loop, Doo-Sabin, Butterfly, $\sqrt{3}$, Kobbelt, Midedge...). De plus, la visualisation peut être aussi améliorée grâce à des techniques comme le *displacement mapping* ou le *normal mapping* qui permettent d'intégrer des détails à la surface lors du rendu. Enfin, l'application peut être complétée avec un *parser* de fichiers plus étendu permettant de charger (ou de sauvegarder) des modèles 3D dans des fichiers de formats divers (*.obj*, *.wrl*...) et ce à travers d'une interface plus élaborée (en QT notamment).

Annexe A

Diagramme de classes

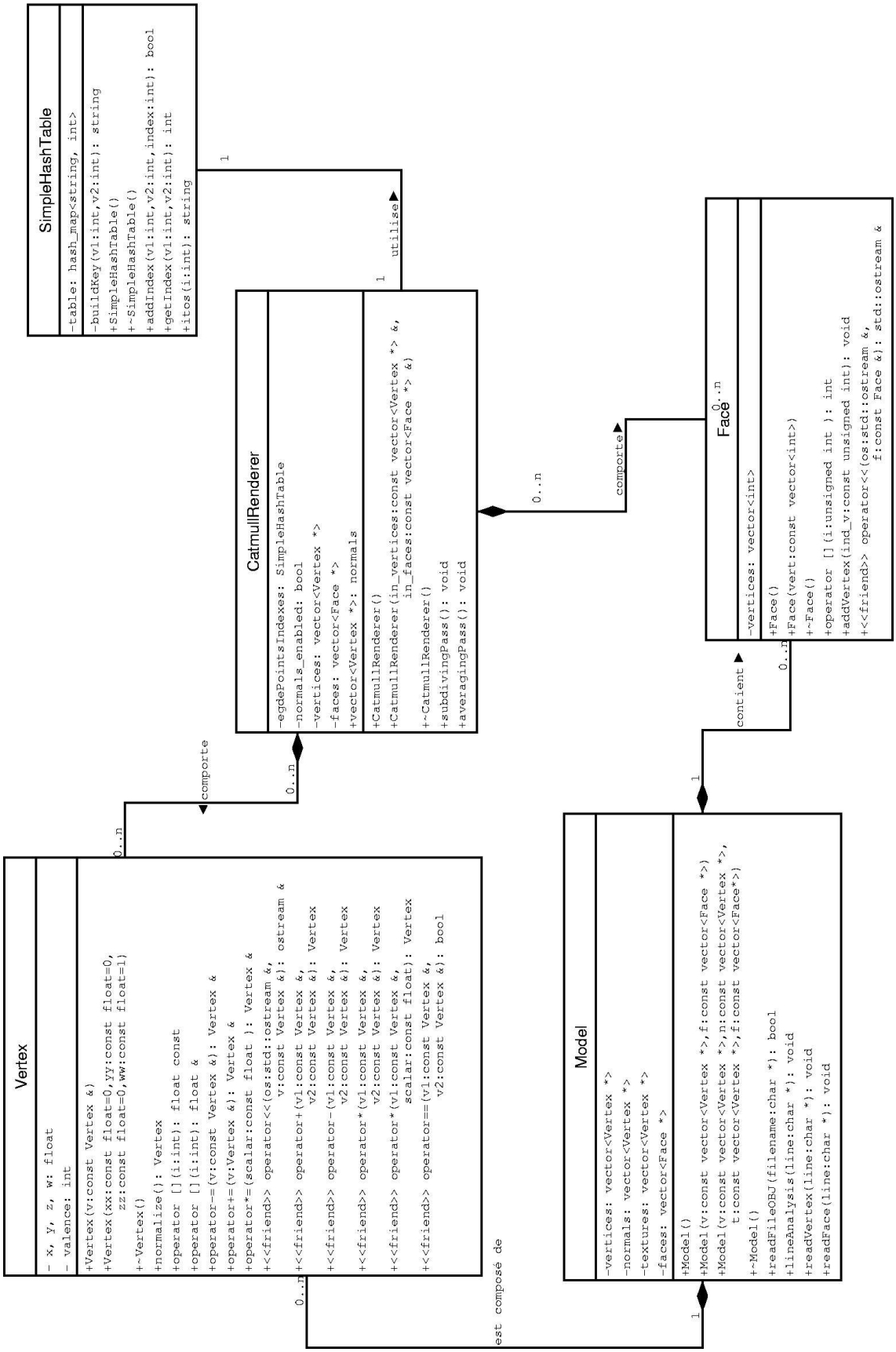

FIG. A.1 – Diagramme de classes

Annexe B

A Realtime GPU Subdivision Kernel

Cette annexe correspond au papier de Le-Jeng Shiue, Ian Jones et Jörg Peters (University of Florida) qui a servi de base à notre projet.

Bibliographie

[1] Arb_vertex_buffer_object. Technical report, NVIDIA Corporation, oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt, January 2003.

[2] Using vertex buffer object (vbos). Technical report, NVIDIA Corporation, www.nvidia.com/object/using_VBOs.html, October 2003.

[3] Ext_pixel_buffer_object. Technical report, NVIDIA Corporation, www.nvidia.com/dev_content/nvopenglspecs/ GL_EXT_pixel_buffer_object.txt, March 2004.

[4] Ext_framebuffer_object. Technical report, NVIDIA Corporation, www.opengl.org/documentation/ extensions/EXT_framebuffer_object.txt, January 2005.

[5] Technical brief : Textures downloads and readbacks using pixel buffer object in opengl. Technical report, NVIDIA Corporation, download.nvidia.com/developer/Papers/2005/Fast_Texture_Transfers/, August 2005.

[6] Song Ho Ahn. Opengl vertex buffer object (vbo), 2005.

[7] Tamy Boubekeur. Projet de fin d'études master 2 mm : Subdivision de surface sur gpu, 2006.

[8] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6) :350–355, November 1978.

[9] G. M. Chaikin. An algorithm for high-speed curve generation. *Computer Graphics and Image Processing*, 3 :346–349, 1974.

[10] Nvidia Corporation, editor. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping. 2005.

[11] Porquet Damien. Pixel shader, 2005. www.msi.unilim.fr/p̃orquet/memoire/node76.html.

[12] Porquet Damien. Vertex shader, 2005. www.msi.unilim.fr/p̃orquet/memoire/node75.html.

[13] D. W. H. Doo and M. A. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6) :356–360, November 1978.

[14] N. Dyn, D. Levin, R. Sackler, and B. Sackler. A butterfly subdivision scheme for surface interpolation with tension control, 1988.

[15] Paul Frazee. Nehe lesson 45 : Vbo, 2003.

[16] GPGPU. General-purpose computation using graphics hardware.

[17] Simon Green. The opengl frame buffer object extension. Technical report, NVIDIA Corporation, download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf, 2005.

[18] X. Guo and O. Pont. Surfaces de subdivision et traitement des maillages 3d. 2005.

[19] Mark Harris. General purpose computation on gpus. In Nvidia Developer Technology Group, editor, *GPGPU*, Grenoble France, 2004. Eurographics 2004.

[20] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow : a gpu-based particle engine. In *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM Press.

[21] L. Kobbelt. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. *Eurographics '96*, 15(3) :C409 – C420, 1996.

[22] L. Kobbelt. $\sqrt{3}$-subdivision. *Proceedings of SIGGRAPH 2000*, pages 103–112, 2000.

[23] KOLB, LATTA, and REZK-SALAMA. Hardware based simulation and collision detection for large particle systems. In *HWWS '04 : Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2004.

[24] C.T. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, 1987.

[25] Nvidia. *GPU programming Guide*. developer.nvidia.com/object/gpu_programming_guide.html, v2.4.0 edition, 08 2005.

[26] Williams of NVIDIA (SPECopc chair) and Evan Hart of ATI. Efficient rendering of geometric data using opengl vbos in specviewperf, 2005.

[27] OpenGL.org. *OpenGL Shading Language*. 192.48.159.181/documentation/oglsl.html, v1.10 edition.

[28] Le-Jeng Shiue, Ian Jones, and J. Peters. A realtime gpu subdivision kernel. In Marcus Gross, editor, *Siggraph 2005, Computer Graphics Proceedings*, Annual Conference Series. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2005.

[29] Suresh Venkatasubramanian. The programmable pipeline. Jan 2005.

[30] Joe D. Warren and Scott Schaefer. A factored approach to subdivision surfaces. *IEEE Computer Graphics and Applications*, 24(3) :74–81, 2004.

[31] Chris Wynn. Opengl render-to-texture. 2002. Nvidia Developer Technology Group.