Rapport du projet de fin d'études
« Visualisation temps réel de modèles détaillés »

Rhenaud Dubois
Fan Jiang
Nicolas Moignot

Clients : Pierre Bénard et Romain Pacanowski
Encadrant : Pascal Desbarats

28 mars 2022

Table des matières

Introduction **3**

1 Analyse de l'existant **4**
1.1 Techniques existantes 4
1.2 Outils existants 7
1.3 Données 8

2 Structure du projet **10**
Objectifs à l'origine 10
Objectifs additionnels 10
Structure du projet 11
Contraintes 11
2.1 Visionneur 12
2.2 Décimateur 15
2.3 Script de synthèse 16

3 Organisation du projet **17**
3.1 Composition de l'équipe 18
3.2 Environnement de travail et communication 19
3.3 User stories 21
3.4 Tâches 23
3.5 Tests 26
3.5.1 Visionneur 26
3.5.2 Décimateur 26
3.5.3 Script de benchmarking 27
3.6 Gantt 27
3.6.1 Gantt prévisionnel 27
3.6.2 Gantt effectif 29

4 Choix techniques **30**
4.1 Visionneur 30

4.1.1 Architecture de l'application 31
4.1.2 Paramètres en entrée 32
4.1.3 Interface graphique 33
4.1.4 Chargement de fichier PLY 33
4.1.5 Moteur de rendu 35
4.1.6 Gestion des shaders 38
4.2 Décimateur 41
4.2.1 Paramètres en entrée 41
4.2.2 Décimation 41
4.3 Script de synthèse 43
4.3.1 Paramètres en entrée 43
4.3.2 Benchmarking 43

5 Réalisations **45**
5.1 Adaptation par rapport au prévisionnel 45
5.1.1 Problèmes rencontrés 45
5.2 Résultats obtenus 46

Conclusion **48**
Améliorations possibles 49

Table des figures **52**

Bibliographie **53**

Glossaire **56**

Acronymes **58**

Annexes **59**

Introduction

La Coupole est une plateforme permettant de scanner des objets 3D avec une très haute résolution, développée dans le cadre du projet *Material* soutenu par l'*ANR* en collaboration avec les équipes *Maverick* (à Grenoble) et *Manao* (à Bordeaux) de l'*Inria*, le *MEB*, l'*Institut d'Optique* de Bordeaux et le *CNRS*[1][2]. Elle est notamment utilisée pour reconstituer virtuellement des textiles anciens. L'une des principales différences de la plateforme comparée à un scanner traditionnel est qu'elle permet de capturer la réponse optique de l'objet.

Pour cela, l'objet à scanner est placé au centre d'un dôme de 1 080 lampes blanches, activables individuellement, et l'acquisition se fait à l'aide d'une caméra haute résolution montée sur un bras robotisé sur 6 axes. En faisant varier les points de vue et l'éclairage de l'objet, on obtient une très grande quantité de données brutes (de l'ordre de 20 To/m^2), qui seront ensuite compilées pour générer un maillage à faces triangulaires, ayant des matériaux permettant de simuler leur réponse optique.

Le maillage créé est alors bien plus léger (de l'ordre de plusieurs giga-octets), mais reste encore trop lourd pour pouvoir être facilement visualisé et manipulé à l'aide des outils standards.

Le sujet proposé par les clients est de concevoir un logiciel de visualisation permettant d'afficher et de manipuler ces maillages en temps réel, adapté aux données spécifiques acquises par *la Coupole* (comme la gestion des matériaux par face) ayant des meilleures performances que des outils plus génériques. Il est également demandé d'implémenter plusieurs méthodes de rendu et de les comparer en fonction de paramètres tels que le nombre de lampes dans la scène.

Chapitre 1

Analyse de l'existant

1.1 Techniques existantes

Une scène 3D définit différents éléments dans un espace en trois dimensions, dont à minima :

— une *caméra*, qui permet de définir un point de vue ;

— un ou plusieurs *objets* primitifs (comme un plan, un cube ou une sphère, qui peuvent être construits à l'aide de paramètres) ou sous forme de maillage (une liste de sommets et une liste de faces liant ces sommets) ;

— et une ou plusieurs *lampes*, de différents types (comme les lampes directionnelles, ponctuelles et ambiantes), qui permettent d'éclairer les objets de la scène et créer des effets de lumière.

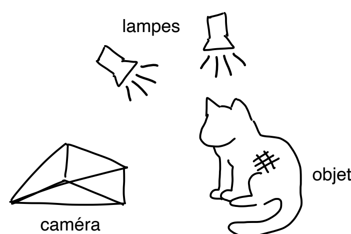


FIGURE 1.1 – Contenu d'une scène 3D simple

Lors d'un *rendu 3D*, l'ordinateur crée une image 2D de la scène 3D depuis le point de vue de la caméra, en faisant interagir les autres éléments entre eux de telle sorte à connaître la valeur de chaque pixel de l'image finale. Pour faire cela, il y a deux grandes techniques :

— le *rendu par tracé de rayon* (ou *ray-tracing*), où des rayons sont envoyés depuis la caméra et vont rebondir dans la scène afin de calculer la valeur de chacun des pixels de l'image, ce qui peut être très long à calculer mais donne d'excellents résultats ;

— et le *rendu par rastérisation*, où les objets (en particulier les faces d'un maillage) de la scène sont aplatis en fonction du point de vue de la caméra pour être appliqués sur l'image, ce qui est bien plus rapide puisqu'elle s'appuie sur des calculs simples qui peuvent être effectués en parallèle (plusieurs faces peuvent être traitées en même temps) sur la carte graphique (qui permet aujourd'hui d'effectuer le même calcul avec différentes données en parallèle), et où le calcul de l'éclairage se fait seulement pour les pixels qui seront affichés sur l'image.

Le *tracé de rayon* est principalement utilisé dans des cas où la qualité du rendu doit être élevée sans contrainte de temps, comme dans les domaines du cinéma. La *rastérisation* quant à elle est utilisée dans des cas où le rendu doit être fait en temps réel, c'est-à-dire que l'image doit pouvoir être mise à jour aussi vite que la scène est modifiée : cette méthode est privilégiée dans les cas où l'utilisateur interagit avec la scène, comme dans la modélisation ou le jeu vidéo. D'autres techniques existent, certaines se basant sur une fusion des deux. Dans le cas d'un logiciel de visualisation standard, on utilise la technique de *rendu par rastérisation*.

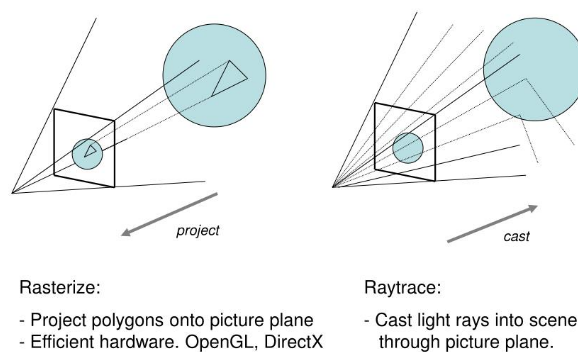
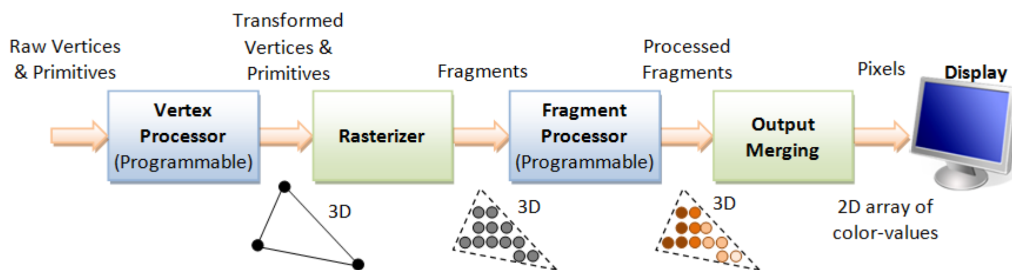


FIGURE 1.2 – Comparaison du rendu par tracé de rayon et par rastérisation[3]

En général, la géométrie de la scène (sommets et faces) et d'autres données (comme les textures) sont envoyées par le processeur à la carte graphique pour être stockées dans sa mémoire cache. Pour faire un rendu, le processeur demande à la carte graphique de traiter une liste de faces, stockées dans cette mémoire cache, ce qui va lancer une *pipeline 3D* sur ces faces : celle-ci va chercher les sommets correspondants et effectuer différents traitements pour pouvoir trouver leurs positions sur l'image, pour ensuite les relier pour reconstituer la face, tout en gardant en mémoire la face la plus proche déjà traitée dans un *depth buffer* pour chaque pixel (on parle alors de *fragment*).



3D Graphics Rendering Pipeline: Output of one stage is fed as input of the next stage. A vertex has attributes such as (x, y, z) position, color (RGB or RGBA), vertex-normal (n_x, n_y, n_z), and texture. A primitive is made up of one or more vertices. The rasterizer raster-scans each primitive to produce a set of grid-aligned fragments, by interpolating the vertices.

FIGURE 1.3 – Exemple de pipeline 3D standard[4]

Une partie de cette *pipeline 3D* est programmable : à l'aide de programmes appelés *shaders*, on peut redéfinir le traitement des sommets (*vertex shader*), permettant entre autre de les déplacer ou de leurs lier de nouvelles données, ou le calcul des pixels (*fragment shader*), permettant de calculer l'éclairage de la face, et donc la valeur du pixel dans l'image finale. Pouvoir programmer ces deux parties de la *pipeline* permet également de mettre en place différentes méthodes de rendu :

— le *forward shading* est une méthode basique où le calcul de l'éclairage est effectué directement pour chaque face : son principal problème est qu'en fonction de l'ordre des faces dans la liste passée à la *pipeline*, l'éclairage de fragments qui ne seront pas utilisés peut être calculé inutilement, représentant un temps de calcul d'autant plus long qu'il y a de lampes dans la scène ;

— le *deferred shading* est une méthode se basant sur une première passe sans calcul de l'éclairage (seulement la rastérisation des faces et la recherche du fragment le plus proche), dont le résultat sera stocké dans

un *g-buffer* (qui contiendra tous les éléments nécessaires pour pouvoir faire le calcul de l'éclairage, à savoir la couleur du fragment, sa position, sa normale et son matériau), pour ensuite effectuer une seconde passe sur ce *g-buffer* (chargé et lu comme une texture appliquée sur un rectangle couvrant le point de vue de la caméra) afin de calculer l'éclairage uniquement sur les fragments qui seront forcément affichés à l'écran, pouvant ainsi réduire le temps de calcul en cas d'un nombre élevé de lampes ;

— le *clustered deferred shading* est une méthode basée sur la précédente, où au lieu de traiter l'image complète (et donc possiblement perdre du temps à calculer l'impact des lampes sur des pixels qui ne seront pas impactés), l'image est divisée en bloc de sorte à ce que chaque bloc aie une liste de lampes qui vont impacter son éclairage.

Encore une fois, d'autres méthodes plus poussées existent afin de réduire la charge de la carte graphique au maximum et donc de réduire le temps de rendu tout en ayant des scènes plus complexes.

1.2 Outils existants

De nombreuses applications permettent couramment d'afficher des modèles 3D : on peut penser à des applications de création de modèles 3D telles que *Blender*, ou à des moteurs de jeux gérant des modèles 3D tels que *Unity* ou *Unreal Engine*.

Cependant, aucune de ces applications n'est idéale pour subvenir aux besoins de ce projet, car le but principal est de pouvoir afficher un modèle 3D très détaillé en temps réel, et de pouvoir possiblement effectuer des opérations de *benchmarking* sur l'affichage de ces modèles, en suivant l'évolution de un ou plusieurs paramètres, tels que le nombre de matériaux utilisés, le nombre de faces du modèle traité, le nombre de lumières à considérer avec l'objet, et bien d'autres cas.

Après recherches, nous n'avons pas trouvé d'outil permettant de lier l'optimisation de l'affichage d'un modèle 3D détaillé et la réalisation de *benchmark* sur un même ou différents maillages. Ce projet est donc une opportunité de réunir ces deux aspects majeurs de l'application.

1.3 Données

Un maillage 3D peut être représenté à l'aide d'un fichier au format *Polygon File Format (PLY)*. Comme pour la plupart des autres formats de fichiers, un fichier PLY est constitué d'une en-tête (ou *header*, définissant le type et la quantité de données, ainsi que la méthode de stockage) puis des données elles-mêmes.

Dans le cas du format PLY, les données correspondent à la liste des données des sommets, suivie de la liste des données des faces (voir Fig. 5.4 en annexe pour un exemple de fichier). Ces données peuvent être enregistrées soit en texte brut (`ascii`), soit en binaire (`binary_little_endian` ou `binary_big_endian`). L'en-tête est quant à elle toujours enregistrée en texte brut.

En général, les sommets sont représentés par leurs positions (valeurs x, y et z en 3D), et les faces sont représentées comme étant une liste d'indices de sommets, les indices correspondant à leur ordre de définition dans le fichier (le premier sommet a comme indice `0`, le deuxième `1` et ainsi de suite). Dans la grande majorité des cas, les faces sont représentées avec trois indices de sommets : les faces sont triangulaires. Le format PLY permet de définir d'autres attributs par sommet et par face.

Dans le cadre du projet, les maillages exportés à partir des acquisitions faites par *la Coupole* contiennent des données stockées en binaire, avec des attributs supplémentaires par sommet et par face : les sommets sont représentés par leur position (valeurs `x`, `y` et `z`) et par une couleur (valeurs `red`, `green` et `blue`), et les faces sont définies par une liste d'indices de sommets (liste `vertex_indices`) et un identifiant de matériau (valeur `id`). Un exemple d'en-tête est donné Fig. 1.4. Les faces sont de forme triangulaire.

```
ply
format binary_little_endian 1.0
comment generated by coloredMeshConverter
element vertex 28090780
property float x
property float y
property float z
property float red
property float green
property float blue
element face 28996928
property list uchar uint vertex_indices
property int id
end_header
```

FIGURE 1.4 – Exemple d'en-tête de l'un des fichiers fournis par les clients : `gilet_union.ply`

Chapitre 2

Structure du projet

Objectifs à l'origine

Le projet consistait à l'origine en la création d'un outil permettant d'afficher et de manipuler en temps réel un maillage acquis par *la Coupole*, ainsi que de comparer différentes méthodes de rendus. Les objectifs étaient alors :

- la création de l'outil de visualisation ;
- l'implémentation des méthodes de rendu *« forward shading »* et *« deferred shading »* ;
- l'implémentation des approches *« shader unique »* et *« shader-par-matériau »* pour chaque méthode de rendu ;
- l'ajout de lampes dans la scène ;
- et la comparaison des différentes méthodes de rendus et des différentes approches en fonction du nombre de lampes dans la scène.

Objectifs additionnels

Ainsi, il a été décidé d'ajouter plusieurs objectifs au projet, dans l'ordre du plus important au moins important :

- l'utilisation de fichiers externes aux shaders pour définir les fonctions matériau, afin de pouvoir utiliser les mêmes matériaux dans différents shaders sans duplication de code ;
- l'ajout d'un mode complet d'évaluation des performances (mode *benchmark*) à l'outil, pour connaître entre autre la charge processeur, le coût mémoire et le temps de rendu de l'affichage du maillage sélectionné ;
- et l'ajout d'un outil externe pour décimer un maillage en conservant ses propriétés, permettant d'afficher le maillage avec une charge graphique

moins lourde, mais également de lancer un *benchmark* sur un même maillage avec différentes résolutions.

Il a également été défini que, si le temps le permettait, d'autres fonctionnalités pouvaient être ajoutées, tels que :

— l'implémentation de la méthode de rendu *« clustered shading »* ;

— la réorganisation des faces du maillage pour optimiser l'accès à la mémoire cache de la carte graphique ;

— et l'implémentation d'un système de *Level of Detail (LoD)*, qui permet d'utiliser un maillage avec une résolution plus basse suivant la distance entre la caméra et le modèle.

Structure du projet

Avec l'accord des clients, le projet a été divisé en trois outils indépendants, pouvant être implémentés en parallèle :

— un *visionneur*, qui permet d'afficher un fichier PLY dans une scène 3D simple à l'aide de plusieurs méthodes de rendu, mais aussi d'évaluer leurs performances en fonction de la scène ;

— un *décimateur*, qui permet de réduire le nombre de faces d'un fichier PLY passé en entrée, suivant différentes méthodes de décimation ;

— et un *script de synthèse* qui permet de lancer les deux programmes et de compiler les données acquises lors de l'évaluation des méthodes de rendu en fonction d'une scène.

Contraintes

Plusieurs contraintes ont été posées par les clients dès le début du projet, ce qui nous a permis de choisir les technologies à utiliser au plus tôt. Ces contraintes sont :

— le développement en *C++* avec l'API *OpenGL* ;

— l'utilisation multi-plateforme : le projet doit pouvoir être lancé sur n'importe quelle machine récente, que ce soit sous *Windows*, *macOS* ou une distribution *Linux* ;

— et l'utilisation en temps réel : le *visionneur* doit permettre d'afficher et de manipuler le maillage avec comme objectif une moyenne de 30 images par seconde sur les ordinateurs du *CREMI* de l'*Université de Bordeaux*.

2.1 Visionneur

Le *visionneur* prend la forme d'une application graphique permettant d'afficher et de manipuler le maillage, à l'aide de différentes méthodes de rendus. Il peut être lancé en mode *benchmark* pour pouvoir estimer l'efficacité d'une méthode de rendu avec différents paramètres par rapport à d'autres pour une scène 3D donnée.

Parmi les paramètres que l'on peut spécifier dans l'application, il doit y avoir à minima :

— le chemin vers le maillage à afficher ;

— la méthode de rendu : *forward shading* ou *deferred shading* ;

— l'approche du rendu : *shader unique* ou *shader-par-matériau* ;

— et le nombre de lampes dans la scène, placées uniformément sur un hémisphère englobant le maillage pour simuler une configuration de *la Coupole*.

Les différents paramètres de l'application peuvent être spécifiés de plusieurs façons : en arguments du programme, via un fichier de configuration ou directement depuis l'interface de l'application (via des contrôles clavier ou à l'aide d'options dans les menus en mode *visionneur*). En mode *benchmark*, les paramètres ne peuvent pas être modifiés après le lancement de l'application, permettant de lancer plusieurs *benchmarks* avec des paramètres variés de façon automatisée et reproductible, tout en évitant de corrompre les résultats par inadvertance.

Voici la liste complète des fonctionnalités à implémenter dans le *visionneur* :

— l'ouverture de la fenêtre avec un contexte *OpenGL* et la gestion des entrées clavier et souris ;

— la gestion des paramètres passés en argument au lancement ou stockés dans un fichier de configuration ;

— l'ajout d'une interface permettant de modifier ces paramètres et d'afficher des informations ;

— l'ajout de raccourcis clavier pour modifier ces paramètres ;

— la lecture d'un fichier PLY comprenant un maillage à faces triangulaires ;

— le traitement du maillage lu pour être affiché : le calcul des normales par sommet, un stockage des données adapté pour le transfert vers la carte graphique ;
— la gestion et la compilation automatique des shaders ;
— l'ajout de fichiers et de code généré automatiquement dans les shaders ;
— la gestion de la scène, avec des lampes et une caméra, ainsi que leurs matrices de transformation quand c'est nécessaire ;
— les mouvements de la caméra à l'aide des entrées clavier et souris ;
— l'exportation de la scène et des mouvements de caméra afin de pouvoir les reproduire en mode *benchmark* ;
— le rendu du maillage et son affichage en temps réel dans la fenêtre ;
— le changement de méthode et d'approche de rendu ;
— le lancement du programme en mode *benchmark* pour n'effectuer que les rendus et quitter le programme ;
— et l'exportation de fichiers de données brutes en mode *benchmark*, afin de les compiler et exporter des résultats utilisables dans le *script de synthèse*.

Pour respecter les contraintes dictées par les clients :
— l'interface graphique doit être gérée à l'aide d'une bibliothèque optimisée pour le rendu 3D ;
— la lecture du fichier PLY en entrée doit être plus rapide qu'elle ne l'est sur d'autres logiciels comme *MeshLab*[5] ;
— et les résultats du mode *benchmark* doivent être reproductibles.

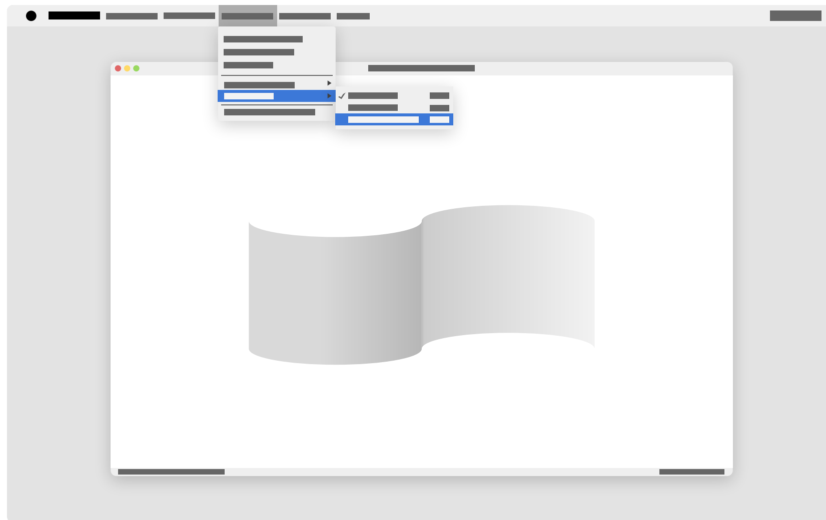


FIGURE 2.1 – Première maquette de l'interface du *visionneur*, présentée aux clients le mercredi 3 février 2022

Depuis la première maquette présentée aux clients (Fig. 2.1), il a été question que la zone principale de la fenêtre n'affiche que le maillage dans la scène. Des paramètres sont accessibles dans la barre d'outils. La barre d'état affiche des données concernant le maillage, comme le nombre de sommets et de faces, ainsi que le temps pris pour faire le dernier rendu. Il a été convenu de ne pas perdre de temps sur une interface graphique poussée, et de se concentrer exclusivement sur les fonctionnalités du programme.

2.2 Décimateur

Le *décimateur* est une application séparée du *visionneur* présenté plus tôt. Son but est de prendre un fichier PLY en entrée, de réduire le nombre de faces totales de ce maillage, avant d'écrire ce nouveau maillage dans une fichier PLY en sortie. Cette application tierce est utilisée dans le but d'automatiser le processus de *benchmarking* dans le cas où l'on veut tester la vitesse de rendu d'un modèle sur différentes résolutions.

L'application est capable de générer un fichier PLY de taille réduite en utilisant un *coefficient de décimation*, qui indiquera le facteur par lequel le nombre de faces courantes doit être divisé pour savoir combien de faces le maillage décimé comprendra.

Ainsi, si un maillage de 1 000 faces est donné en entrée avec un coefficient de décimation égal à 4, alors le nombre de faces du maillage décimé créé sera d'au maximum 250 faces.

Au vue des différents maillages que l'application se doit de gérer, le *décimateur* se doit aussi de gérer les couleurs des sommets, ainsi que les identifiants de matériau des faces du maillage traité.

Suivant les fonctionnalités décrites ci-dessus, nous avons donc besoin de plusieurs briques pour que l'application produise le résultat attendu :

- un lecteur de fichiers PLY : les fichiers PLY d'entrée doivent être chargés correctement afin d'être ensuite décimés ;
- un manipulateur de maillage : même s'il est possible de modifier les données directement, une bibliothèque capable de gérer une structure de données permettant de représenter et manipuler un maillage triangulaire avec des attributs par face est la bienvenue, afin de savoir exactement ce que l'on est en train de modifier ;
- et un exporteur de fichiers PLY : une fois nos manipulations effectuées, les données doivent être sauvegardées dans un nouveau fichier PLY pour qu'elles puissent être utilisées par d'autres applications, telles que le *visionneur*.

Additionnellement, un outil de gestion d'arguments de ligne de commande doit être utilisé afin de gérer les différentes entrées du programme.

2.3 Script de synthèse

Le *script de synthèse* est un script écrit en *Python* qui permet de lancer les deux autres outils afin de lancer automatiquement des *benchmarks* et d'exporter des résultats utilisables dans un fichier *CSV*.

Peu de fonctionnalités ont été évoquées avec les clients concernant cette partie du projet. En voici une liste grossière :

- le lancement automatique du *décimateur* sur un fichier PLY pour pouvoir générer plusieurs maillages de différentes résolutions ;
- le lancement automatique du *visionneur* en mode *benchmark* sur ces fichiers PLY et avec différents paramètres (tels que le nombre de lampes) ;
- la lecture des fichiers de données brutes exportés par le *visionneur* ;
- la compilation de ces données en résultats utilisables ;
- l'exportation de ces résultats sous forme de fichier CSV ;
- et l'exportation de ces résultats avec du texte et des images, sous forme d'un rapport au format PDF ou HTML.

Il n'y a pas de contraintes qui affectent réellement cette partie du projet, un script *Python* étant de base lançable sous *Windows* (après installation), *macOS* et *Linux*. Seul le choix des fonctions et bibliothèques utilisées peut possiblement impacter le projet.

Chapitre 3

Organisation du projet

Le projet se tient du jeudi 20 janvier (date du premier rendez-vous avec les clients) au lundi 28 mars 2022 (date de remise des livrables), en tant que projet de fin d'études de *Master Informatique*. Un cours de gestion de projet, *Méthodes et Outils de Conduite de Projets Informatiques (MOCPI)*, est suivi en même temps que le projet.

Le cadre de développement suivi, imposé dans le cadre de MOCPI, est une sorte de *Scrum* allégé :

— la conception du projet se base autout d'*user stories* et des tâches qui en découlent ;

— le suivi se base sur un processus itératif d'une durée d'une semaine, changeant le mardi de chaque semaine (pour coincider avec les TD de MOCPI) ;

— il y a une négociation constante avec les clients sur les choix techniques et le déroulement de l'implémentation ;

— et le code source est régulièrement mis à jour de telle sorte à pouvoir donner lieu à une démonstration à n'importe quel moment.

Il était également demandé d'effectuer un *Test Driven Development*, mais cela s'appliquait peu à la majorité du projet (voir section 3.5).

3.1 Composition de l'équipe

Notre équipe est composée de *Rhenaud Dubois*, *Fan Jiang* et *Nicolas Moignot*. Chacun d'entre nous a eu un rôle bien défini dans le projet :

— *Rhenaud* a apporté des tests pour le *visionneur* avant de se concentrer sur le *décimateur*, tout en assumant le rôle de Scrum master. Il s'est occupé du développement sous une distribution *Linux*.

— *Fan* a travaillé sur le *visionneur* pour une grande partie du projet avant de se concentrer sur la création du *script de benchmarking* en Python qui sert de cohésion entre les différentes applications de ce projet. Il s'est occupé du développement sous *Windows*.

— *Nicolas* a passé le plus de temps sur le *visionneur* et a été celui qui a mis en place une grande partie du projet. Il s'est occupé du développement sous *macOS*.

3.2 Environnement de travail et communication

Le développement sous *Windows* a été effectué sous *Windows 10 64-bit*, sur un ordinateur doté d'un processeur *Intel Core i3 540* de deux cœurs cadencés à $3,07$ GHz, muni d'une carte graphique dédié *NVIDIA GeForce GTX 950* avec 2047 Mo de mémoire et d'une mémoire *LPDDR3* de 8 Go cadencée à 2133 MHz.

Le développement sous *Linux* a été fait sur une machine virtuelle *Ubuntu 21.10 'Impish Indri'*, sur un ordinateur doté d'un processeur *AMD Ryzen 9 5900HX* de 16 cœurs, dont 8 cœurs ont été attribué à la machine virtuelle, cadencés à 3.3 GHz et muni d'une carte graphique intégrée *AMD Radeon Graphics* avec 495 Mo de VRAM, de la moitié d'une mémoire vive de 16 Go et d'une carte graphique dédiée *NVidia GeForce RTX 3060* avec 6 Go de VRAM.

Le développement sous *macOS* a été effectué en grande partie sous *macOS 12.2* puis *macOS 12.2.1*, sur un ordinateur doté d'un processeur *Intel Core i7* (génération *Skylake*) de quatre cœurs cadencés à $2,6$ GHz et muni d'une carte graphique intégrée *Intel HD Graphics 530* avec 1536 Mo de mémoire, d'une mémoire *LPDDR3* de 16 Go cadencée à 2133 MHz et une carte graphique dédiée *AMD Radeon Pro 450* avec 2 Go de mémoire.

Pour partager le code avec le reste de l'équipe, nous avons mis en place deux dépôts *Git* sur *GitHub* : un pour le *visionneur* et le *script de synthèse*[6], et un autre spécifiquement pour le *décimateur*[7].

Le but principal de cette manoeuvre est d'éviter des conflits entre les bibliothèques externes partagées par certains modules ; par exemple, la bibliothèque *PMP* requiert l'utilisation d'une version spécifique de *GLFW*, qui permet de gérer la fenêtre de l'application, tandis que notre projet utilise une version différente de cette même bibliothèque. Le chargement de deux versions d'une même bibliothèque dans un même projet apporte des problèmes de compilation que cette séparation nous a permis de régler.

L'organisation de la charge de travail a été faite à l'aide de *Trello* pour mettre en place le scrumboard et de *Agantty* pour le diagramme de Gantt.

La rédaction des documents a été faite à l'aide d'outils en ligne : *Overleaf*

pour la rédaction de ce rapport au format LaTeX et *Google Slides* pour la création des différentes diapositives.

Pour communiquer avec les autres membres de l'équipe, nous avons rapidement mis en place un serveur *Discord*, nous permettant de nous informer sur les tâches effectuées par les autres, de se partager des informations et des liens, ainsi que de s'entre-aider en cas de problème.

Nous avons également veillé à garder une communication constante avec les clients à l'aide de rendez-vous hebdomadaires. Nous échangions également des mails avec eux lors de gros blocages.

3.3 User stories

Notre projet a plusieurs cas d'utilisation divers, que nous allons présenter ici, ainsi que leurs implications :

[A00] En tant qu'utilisateur voulant faire un rendu, j'ouvre une fenêtre et charge un fichier.

— Création d'une application pour le rendu *OpenGL* via une interface graphique ou en ligne de commande, multiplateforme (*Windows, macOS, Linux*).

— Chargement du maillage (et possiblement de la scène) à l'ouverture de l'application.

— Ouverture d'une fenêtre de sélection de fichier si aucun fichier n'est spécifié lors du lancement de l'application.

— Choix de la méthode de rendu et des matériaux (via l'utilisation de shaders externes).

— Ajout d'une barre d'outils et de raccourcis claviers pour pouvoir ouvrir un autre fichier si besoin.

[B00] En tant qu'utilisateur voulant faire un rendu, j'affiche un modèle 3D avec une méthode de rendu spécifique.

— Implémentation simple des différentes méthodes de rendus.

— Choix de la méthode de rendu dans la ligne de commande de lancer de l'application.

— Ajout d'un menu déroulant dans l'interface pour sélectionner la méthode de rendu.

[C00] En tant qu'utilisateur voulant faire un rendu, je manipule la scène 3D.

— Ajout de déplacement et de rotation de la caméra, à travers la souris et/ou le clavier.

— Ajout de sources lumineuses dans la scène à l'entrée de l'application ou directement dans l'application, avec une intensité commune à chaque source de lumière.

[D00] En tant qu'utilisateur, je modifie les shaders du rendu et les matériaux utilisés.

— Création d'un outil permettant de générer des shaders à partir de fusion de plusieurs fichiers (un *ubershader* comprenant tous les matériaux ou plusieurs shaders, un par matériau).

— Ajout de commandes et d'arguments de la ligne de commande pour importer ces shaders dans le programme.

[E00] En tant qu'utilisateur, je simplifie le modèle 3D de façon automatisée.

— Création d'un outil multiplateforme externe utilisable via la ligne de commande exclusivement.

— Ajout de méthodes de simplification du maillage automatisées (suppression aléatoire de points, de faces, etc.).

— Importation/exportation des fichiers PLY avec la même entête.

[F00] En tant qu'utilisateur voulant faire un *benchmark*, je lance un test sur une scène personnalisée.

— Ajout d'options à l'application de rendu pour le *benchmark*.

— Exportation d'un fichier de données brutes en CSV contenant les différentes métriques à chaque trame (temps de rendu, etc.).

[G00] En tant qu'utilisateur voulant faire un *benchmark*, je lance plusieurs tests et obtiens un rapport à partir d'un script Python

— Création d'un outil de génération de rapports.

— Importation des fichiers de données brutes exportés lors du benchmark.

— Exportation d'un fichier contenant les résultats.

3.4 Tâches

La plupart des tâches ont été décidées au début du projet, d'autres ont été ajoutées en fonction des demandes des clients. Certaines sont marquées comme optionnelles, d'autres ont été supprimées car n'étaient plus adaptées aux demandes des clients. Voici la liste des tâches à la fin du projet, triée par *user story* :

[A00] En tant qu'utilisateur voulant faire un rendu, j'ouvre une fenêtre et charge un fichier.

- **[A01]** Rechercher une bibliothèque d'interface graphique.
- **[A02]** Implémenter la structure de l'application basée sur cette bibliothèque *(Dear ImGui)*.
- **[A03]** Définir et traiter les arguments de l'application en ligne de commandes.
- **[A04]** Créer une fenêtre de sélection de fichiers.
- **[A05]** Ajouter des commandes de gestion des fichiers.
- **[A06]** Rechercher et intégrer à l'application une bibliothèque de gestion de fichiers PLY.
- **[A07]** Lire un fichier PLY et vérifier sa structure.
- **[A08]** Rechercher et intégrer à l'application une bibliothèque de gestion de fichiers TOML.
- **[A09]** Interpréter un fichier de configuration TOML.

[B00] En tant qu'utilisateur voulant faire un rendu, j'affiche un modèle 3D avec une méthode de rendu spécifique.

- **[B01]** Créer des shaders simples.
- **[B02]** Rechercher et intégrer à l'application une API OpenGL.
- **[B03]** Créer une session OpenGL et compiler les shaders.
- **[B04]** Implémenter le rendu direct.
- **[B05]** Ajouter des commandes de sélection de la méthode de rendu à utiliser.
- **[B06]** Implémenter le rendu deferred.
- **[B07]** Implémenter le rendu clustered *(optionnel)*.
- **[B08]** Ajouter une fenêtre pop-up pour définir la taille des tuiles du rendu clustered *(optionnel)*.
- **[B09]** Ajouter des lumières (directionnelle, ponctuelle, lampe d'aire...).

[C00] En tant qu'utilisateur voulant faire un rendu, je manipule la scène 3D.

- **[C01]** Définir des raccourcis clavier pour la gestion des sources lumineuses.
- **[C02]** Implémenter l'ajout/suppression de sources lumineuses aléatoirement réparties sur l'hémisphère englobant.
- **[C03]** Implémenter l'ajout/suppression de sources lumineuses régulièrement réparties sur l'hémisphère englobant.
- **[C04]** Implémenter l'exportation de fichiers comportant la liste des sources lumineuses et leurs attributs.
- **[C05]** Implémenter l'importation de fichiers comportant des sources lumineuses.
- **[C06]** Ajouter les mouvements de la caméra dans la scène en utilisant la souris et le clavier

[D00] En tant qu'utilisateur, je modifie les shaders du rendu et les matériaux utilisés.

- **[D01]** Création du shader statique d'affichage simple.
- **[D02]** Création des shaders statiques d'affichage différé.
- **[D03]** Création de points d'entrée d'ajout de « scripts matériaux » dans les shaders.
- **[D04]** Création de « scripts matériaux » d'exemple (diffus, GGX...)

[E00] En tant qu'utilisateur, je simplifie le modèle 3D de façon automatisée.

- **[E01]** Recherche d'un outil adapté pour simplifier un maillage de grande taille.
- **[E02]** Création de la base de l'application subsidiaire pour simplifier le modèle 3D.
- **[E03]** Ajout de la gestion de plusieurs matériaux sur un groupe de faces à fusionner.
- **[E04]** Gérer les différents arguments d'entrée de l'application.
- **[E05]** Gérer les couleurs et les matériaux.
- **[E06]** Charger le fichier PLY correctement.
- **[E07]** Déplacer le vertex collapsé entre les deux vertices du halfedge.
- **[E08]** Sauvegarder le fichier PLY correctement.

[F00] En tant qu'utilisateur voulant faire un *benchmark*, je lance un test sur une scène personnalisée.

- **[F02]** Mise en place d'une architecture de script de benchmarking.
- **[F03]** Gestion des modèles 3D d'entrée.

[G00] En tant qu'utilisateur voulant faire un *benchmark*, je lance plusieurs tests et obtiens un rapport à partir d'un script Python

- **[G01]** Rechercher un moyen de générer un rapport.
- **[G02]** Rechercher une bibliothèque de plotting.
- **[G03]** Créer les fichiers de modèles décimés.
- **[G04]** Mettre les données dans le CSV.

3.5 Tests

3.5.1 Visionneur

Étant donné que l'application est une application graphique, la majorité des tests que nous pouvons effectuer sur celle-ci est en rapport avec des éléments ou des actions graphiques, ce qui veut dire que créer des tests unitaires ou de validation est compliqué, voire impossible. De par cette raison, toute intéraction graphique est constituée de *« monkey testing »*, qui consiste à tester les différentes intéractions à la main, avec un utilisateur humain.

De par ce fait, les tests sur le chargement de fichier *PLY* à l'entrée de l'application, les tests de manipulation de la scène soit par la souris, soit par le clavier, et le reste des intéractions via la souris et/ou le clavier sont testées manuellement.

Cependant, certains modules sont testables, tel que les différents paramètres d'entrée de l'application, que ce soit via le fichier de configuration en *TOML* ou les arguments de la ligne de commande utilisée pour appeler l'application.

Pour ces derniers, chaque argument de la ligne de commande et chaque option est testée séparément, avec un cas fonctionnel, et des cas possibles non fonctionnels (tel que le manque de la commande si elle est requise, ou l'utilisation de deux options ou arguments exclusifs l'un de l'autre), et enfin un cas général non fonctionnel (tel qu'un mauvais type de donnée passé en entrée).

Des tests supplémentaires ont aussi été effectués sur le bon chargement de fichiers *PLY* dans l'application, en comparant par exemple le chargement d'un fichier spécifique avec une structure qu'il est censé charger.

3.5.2 Décimateur

Malgré le retard pris lors de la création du logiciel, le décimateur n'a aucune routine de test au jour du rendu. Cependant, des tests simples tels que le bon chargement d'un fichier *PLY*, la bonne sauvegarde d'un fichier traité et la correspondance entre un fichier sorti par l'application et un autre fichier décimé valide avec une graine de randomisation donnée est tout à fait possible.

3.5.3 Script de benchmarking

Le script *Python* a été ajouté à la fin de projet et n'a pas été systématiquement testé par manque de temps. Si le temps le permet, nous pouvons faire un test de lecture d'un fichier *CSV*, un test du format du contenu d'un fichier *CSV* et un test d'écriture d'un fichier *CSV*.

3.6 Gantt

Nous gérons l'avancement du projet grâce à des *diagrammes de Gantt.* Dans 3.4, nous avons clarifié chaque tâche qui doit être accomplie dans l'ensemble du projet et défini le temps dont elles ont besoin. Sur cette base, dans le diagramme de Gantt, nous devons principalement effectuer deux types de travail. La première consiste à déterminer les dépendances entre les tâches, et d'autre part, en fonction de ces dépendances et du temps requis pour chaque tâche, attribuer le temps spécifique pour chaque tâche.

Étant donné que le diagramme de Gantt d'origine n'est qu'un plan, dans le travail réel, certaines tâches ne peuvent pas être complétées conformément au plan, et dans le processus du projet, nous constatons parfois que certaines tâches ne sont pas nécessaires, et parfois nous devons ajouter des tâches supplémentaires, de sorte que nous devons constamment mettre à jour le diagramme de Gantt dans le processus de travail, jusqu'à la fin du projet pour produire la version finale. Compte tenu de cette situation, nous gardons deux version des diagrammes de Gantt, l'une est le Gantt prévisionnel et l'autre est le Gantt effectif.

3.6.1 Gantt prévisionnel

Dans le Gantt prévisionnel, nous organisons les tâches selon certains principes. Tout d'abord, nous trions les dépendances entre les tâches. Par exemple, la tâche *A02* doit attendre la fin de la tâche *A01* avant d'être commencée, tandis que la tâche *A04* doit dépendre de la tâche *A02*

Deuxièmement, nous organisons le nombre de tâches simultanées en fonction du nombre de membres de l'équipe et réservons un certain temps pour résoudre les problèmes qui surviennent au cours de la tâche. Puisque nous avons un total de trois membres qui écrivent le code, nous devons essayer de ne pas considérer plus que trois tâches en même temps.

Enfin, nous prévoyons suffisamment de temps pour rédiger le rapport, et en même temps traiter les éventuels problèmes laissés dans le code.

En suivant ces principes, nous organisons chacunes de nos tâches ainsi :

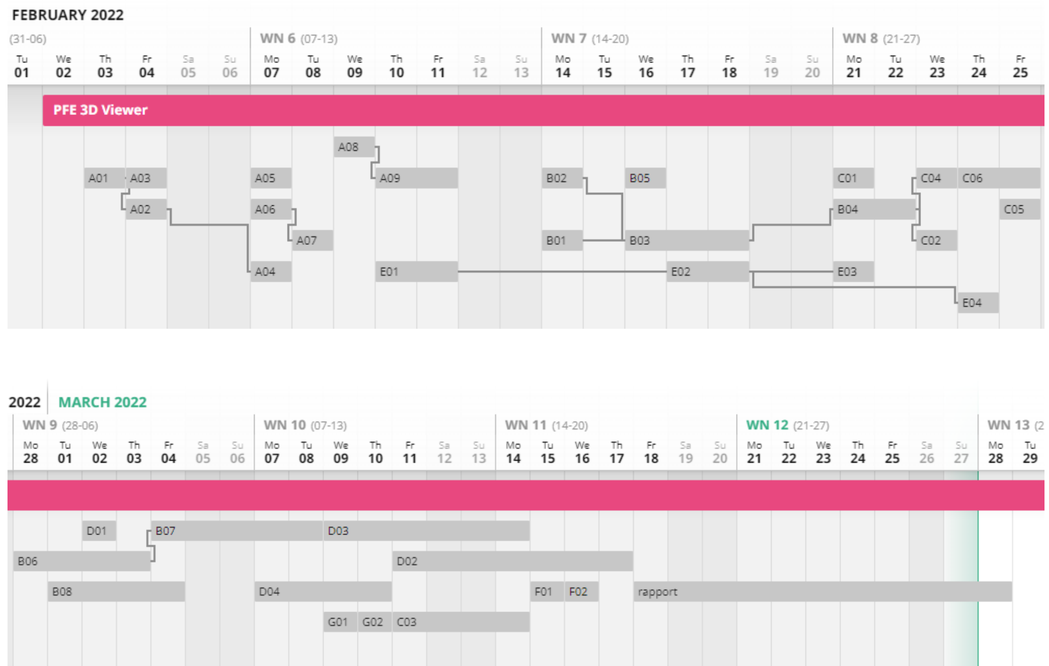


Figure 3.1 – Gantt prévisionnel

3.6.2 Gantt effectif

Cependant, au fur et à mesure de l'avancement de l'application, nous avons recontré quelques problèmes qui seront vite apparents dans la figure ci-dessous :

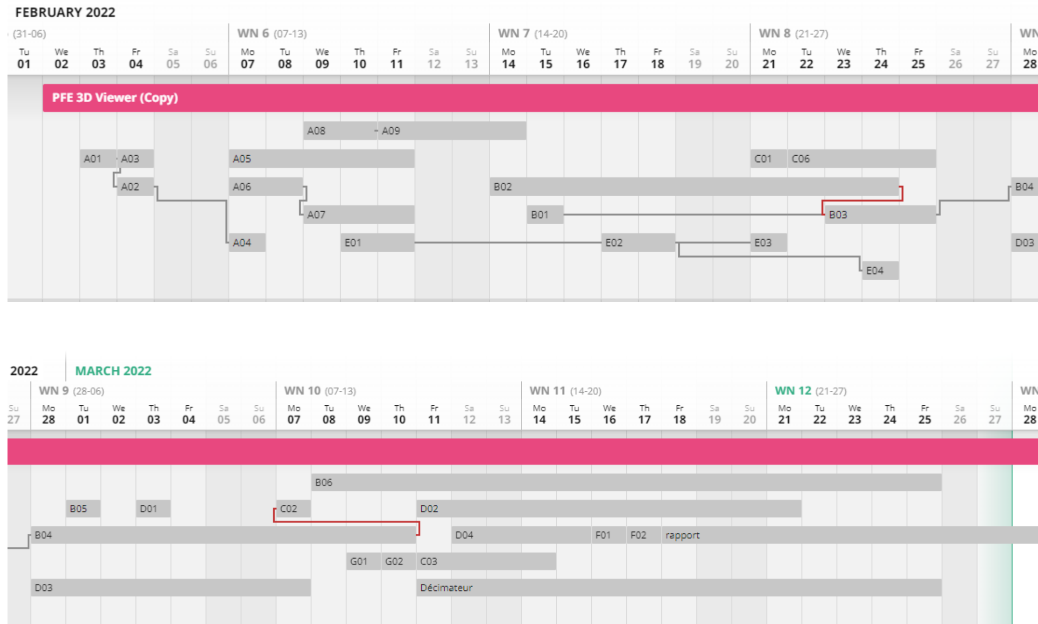


FIGURE 3.2 – Gantt effectif

Comme il est possible de le voir, de nombreues tâches telles que la tâche *B02*, qui auraient dû nous prendre une journée, a fini par prendre deux semaines, et c'est loin d'être le seul cas dans ce projet.

Ce retard a eu un effet boule de neige qui a retardé chacun de nos modules, et ce plusieurs fois sur des tâches que nous pensions atteignable en un jour au maximum, mais qui se sont révélées être bien plus difficiles à réaliser suite à des problèmes tel qu'un manque de connaissances sur le système utilisé, ou une impossibilité d'effectuer la tâche de la manière dont nous l'avons envisionné.

Nous tenions à avoir un temps de préparation pour la rédaction du rapport et la création de la présentation suivante, mais ce temps nous a été coupé court par le retard accumulé au travers des différentes tâches plus longues que ce que nous avions prévu.

Chapitre 4

Choix techniques

4.1 Visionneur

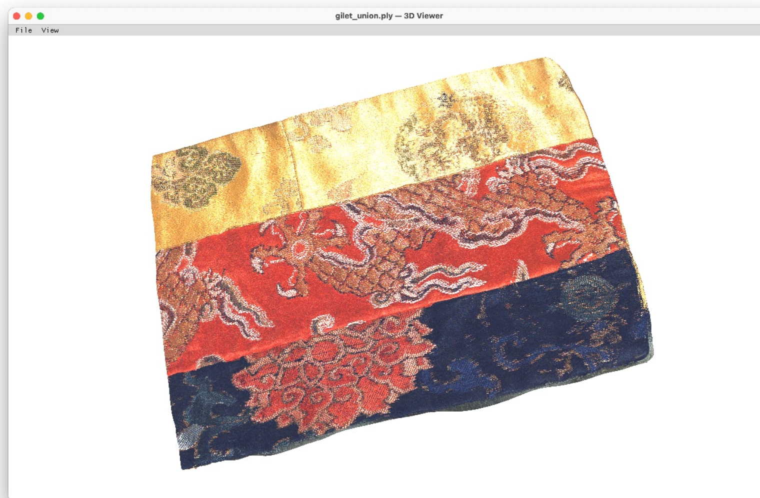


FIGURE 4.1 – Capture d'écran du visionneur

Pour rappel, le *visionneur* est une application graphique permettant d'afficher et de manipuler le maillage en faisant varier la méthode de rendu. Il peut être lancé en mode *benchmark* pour calculer le temps de rendu d'un maillage en fonction de paramètres.

Les principales contraintes sont le développement multi-plateforme en *C++* et se basant sur *OpenGL*, ainsi que la rapidité de l'application, notamment sur le chargement du fichier et sur le temps de rendu.

4.1.1 Architecture de l'application

Notre application est basée sur une classe `Context`, créée et lancée par le point d'entrée de l'application, la fonction `main()`. Elle permet de lier tous les objets et de leur permettre de communiquer entre eux aisément. Un exemple d'organisation des classes avec un tel architecture est donné Fig. 4.2.

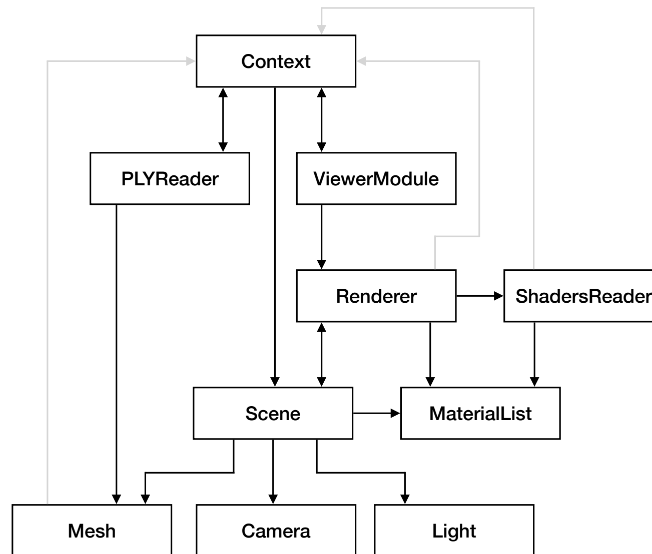


FIGURE 4.2 – Diagramme d'architecture montrant l'organisation des différentes classes liées au rendu 3D

4.1.2 Paramètres en entrée

Pour pouvoir être lancé de façon automatique avec différents paramètres pour le mode *benchmark*, nous avons dû mettre en place plusieurs moyens pour spécifier ces paramètres, avec un système de priorité. De plus faible au plus fort :

— des valeurs par défaut, définies directement dans le code ;

— des valeurs stockées dans un fichier de configuration *TOML*, passé en paramètre ;

— des valeurs passées en paramètre au lancement de l'application ;

— et des valeurs modifiées pendant l'exécution, via des menus de l'interface et des raccourcis clavier.

Le choix d'utiliser un système de fichier de configuration permettait de pouvoir exporter les paramètres de l'application en mode *benchmark*, afin de rendre reproductibles facilement les résultats obtenus, en ayant à passer seulement un argument au lancement.

Nous avons choisi pour cela de passer par des fichiers au format *Tom's Obvious Minimal Language (TOML)* : même si il est relativement récent par rapport à d'autres formats (sa première version stable ayant été publiée début 2021), elle a l'avantage de se baser sur les fichiers *INI*, introduits par *Microsoft* en 1985 et largement adoptés depuis, mais dont le format n'a jamais été formalisé[8]. Il a également l'avantage de bénéficier d'une grande quantité de bibliothèques de lecture de fichier, disponibles dans plusieurs langages dont le *C++*.[9]

Pour lire nos fichiers de configuration au format TOML, nous sommes partis sur l'utilisation de la bibliothèque *TOML11*[10] : cette bibliothèque est l'une des trois bibliothèques C++ recommandées par le toml lui-même, elle est compatible avec la dernière version de TOML v1.0.0 et C++11, et elle affiche des messages d'erreur très informatifs, donc plus facile à déboguer.

Pour gérer les arguments passés au logiciel à son lancement, nous avons utilisé *CLI11*[11] : le but recherché avec cette bibliothèque est de pouvoir analyser les arguments d'une ligne de commande et d'ajouter facilement des conditions sur ces derniers, en utilisant une bibliothèque sans dépendance. De plus, elle est compatible avec *Windows*, *macOS* et *Linux*, ce qui est un des besoins de l'application. Finalement, elle permet de savoir assez facilement quelle erreur a été produite si un mauvais argument est passé.

4.1.3 Interface graphique

Pour l'implémentation de l'interface graphique, nous avons choisi d'utiliser la bibliothèque *Dear ImGui*[12], et plus particulièrement son implémentation se basant sur *GLFW*[13] et *OpenGL* : cela permet de créer la fenêtre et son interface de façon uniforme sur chacune des plateformes demandées par les clients, à savoir *Windows*, *macOS* et *Linux*.

Cette bibliothèque présente plusieurs avantages : elle est optimisée pour faire du rendu 3D et elle donne accès à plusieurs *widgets* qui nous seront utiles (comme la barre d'outils). De plus, il existe des extensions permettant de rajouter d'autres *widgets* dont nous aurons besoin (comme la fenêtre de sélection de fichier).

Pour la gestion de l'interface graphique à l'intérieur de notre application, nous avons choisi de nous baser sur des classes dérivées que nous appelons des *modules* dans le code : chaque module est initialisé avec des données et peut être appelé à n'importe quel moment pour le rendu de la fenêtre, à l'aide de la méthode `Render()` que chaque classe implémente. La plupart des modules s'affichent dans la fenêtre sous forme de sous-fenêtres, tandis que le module `ViewerModule`, qui lance le rendu 3D de la scène et l'affiche, s'étend automatiquement sur la zone de la fenêtre disponible.

Pour faire le rendu de la fenêtre, les fonctions `Render()` de certains modules (comme `ViewerModule`) sont appelées directement, puis celles de la liste de tous les autres modules actifs. Cela permet d'afficher autant de modules que l'on veut en même temps, et cela nous a également permis d'intégrer facilement un mode *Scène uniquement*, activable avec la touche `Tab`, qui n'affiche que le module `ViewerModule`.

Pour que ces modules puissent communiquer avec le reste de l'application, chacun d'entre eux contient un pointeur vers l'objet *contexte de l'application*. Les modules marqués comme devenus inutiles sont supprimés de la liste des modules et sont détruits.

4.1.4 Chargement de fichier PLY

Pour lire les fichiers PLY dans l'application, nous sommes partis sur l'utilisation de la bibliothèque *miniPLY*[14] : elle est vantée comme étant plus rapide en lecture que les autres[15], ce qui s'est avéré vrai d'après nos com-

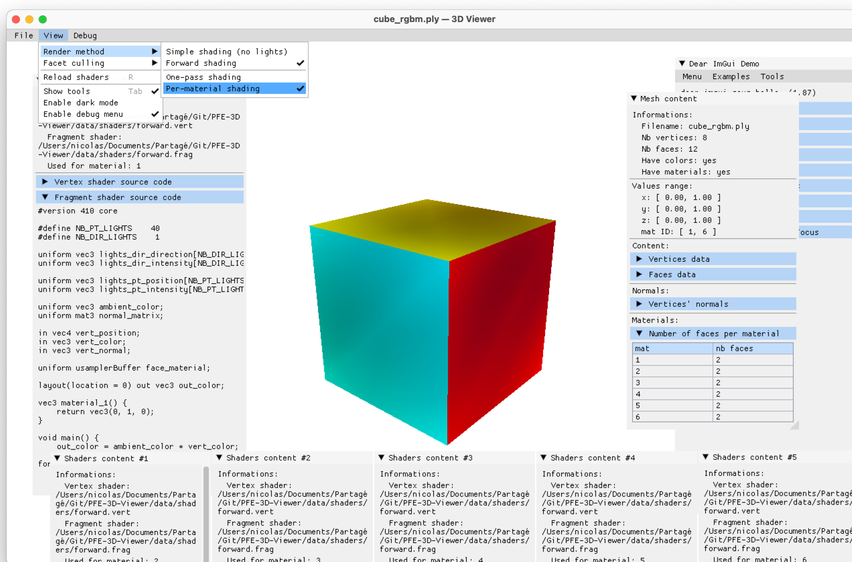


FIGURE 4.3 – Utilisation des fonctionnalités *Dear ImGui* dans le visionneur, avec une barre d'outils et différentes sous-fenêtres créées à partir des modules

paraisons [1] avec d'autres bibliothèques.

D'autres avantages qu'a *miniPLY* sont le fait qu'elle gère les fichiers dont les données sont stockées en binaire comme en texte brut, et qu'elle permet de lire des attributs de sommets et de faces non-standards, ce qui est nécessaire pour lire les identifiants de matériau des faces. Un inconvénient de cette bibliothèque est le fait qu'elle ne permet pas d'écrire un fichier PLY, mais cela n'est pas une fonctionnalité qui devait être intégrée à l'application, donc cela n'a pas été préjudiciable.

Dans notre implémentation, la lecture du fichier se fait à l'aide d'un objet `PLYReader`. Dans un premier temps, les données sont écrites dans un objet `MeshData`. Après le chargement complet du fichier, elles sont copiées et traitées simultanément dans un objet `Mesh`. Cela permet plusieurs choses : de ré-arranger les données pour qu'elles soient plus simples à envoyer à la carte graphique, de supprimer les sommets qui sont inutilisés, et de ré-ordonner les faces en fonction de l'identifiant du matériau (utile pour pouvoir implémenter facilement l'approche de *rendu par matériau*). Les normales aux sommets

1. Nous avons testé *miniPLY* (version du 12 janvier 2021)[14] et *tinyPLY* (versions `2.1`[16] et `2.3`[17]).

sont également calculées à partir de celles des faces (également calculées) à ce moment-là.

4.1.5 Moteur de rendu

Pour communiquer avec la carte graphique facilement en *OpenGL*, nous avons choisi d'utiliser la bibliothèque *glbinding*[18][19] que nous avions déjà utilisée à plusieurs reprises en travaux dirigés (notamment pour l'implémentation du *deferred shading*), qui a une bonne documentation et qui est multi-plateforme.

Nous avions également envisagé d'utiliser *globjects*[20][21], une bibliothèque qui agit comme une extension de *glbinding*, ajoutant des objets plus haut niveau, facilitant ainsi le développement *OpenGL*. Malheureusement, nous n'avons ni réussi à proprement compiler *globjects* et *glbinding* (nécessaire pour le premier) en même temps, ni à trouver une documentation réellement à jour. Pour éviter de perdre plus de temps sur ce point, nous avons choisi de rester sur *glbinding* que nous connaissions déjà et que l'on arrivait à utiliser, quitte à compliquer le développement. Nous avions alors décidé que si le temps le permettait, nous essayerions de nouveau d'utiliser *globjects* pour faciliter la maintenance du logiciel.

À la demande des clients, nous devions implémenter à minima le *forward shading* et le *deferred shading*, et si le temps le permettait, également le *clustered deferred shading*. Chacune des méthodes devait pouvoir être lancée avec les deux approches pour la gestion des matériaux : via un *shader unique* (ou *uber shader*) ou via plusieurs shaders (un shader par matériau).

Pour pouvoir tester prématurement le bon fonctionnement du `ViewerModule` chargé de lancer le rendu et de l'afficher dans la fenêtre, nous avons choisi d'ajouter une méthode de rendu : appelée *simple shading* dans le programme, cette méthode ne calcule pas l'impact de l'éclairage sur le maillage et affiche la couleur du fragment tel qu'il a été interpolé à partir des différentes couleurs spécifiées aux sommets de la face. Par extension, les matériaux ne sont pas traités, puisqu'ils dépendent de l'éclairage. Nous l'avons laissé dans le code car nous pensons qu'il présente un intérêt dans le cadre de l'affichage du maillage : il permet de faire un rendu rapide du maillage sans se préoccuper de l'éclairage et de son impact.

Le *forward shading* a également été implémenté avec succès. Les deux méthodes de rendus permettent de lancer les deux approches de gestion des

matériaux (même si peu utile dans le cas du *simple shading* avec les shaders actuels). Les différentes options peuvent être sélectionnées à partir de la barre d'outils. La méthode de rendu peut également être spécifiée au lancement du programme via les arguments `-simple` (pour le *simple shading*) et `-forward` (pour le *forward shading*).

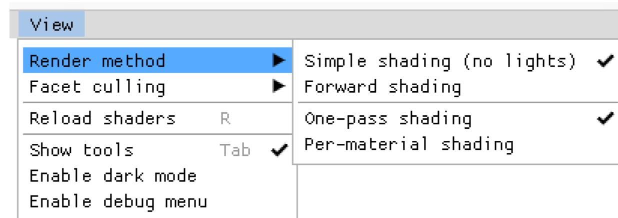


FIGURE 4.4 – Menu de la barre d'outils permettant de modifier la méthode de rendu

Par manque de temps, nous n'avons pas pu implémenter la méthode du *deferred shading*, voulant rendre le *forward shading* complètement fonctionnel avant puisque les deux partagent en grande partie le même code.

Nous avons également ajouté la possibilité de gérer le *facet culling* : cette méthode permet d'afficher les faces en fonction de la face qu'elles exposent à la caméra (définie par rapport à l'ordre des indices de ses sommets), et ainsi de réduire la charge de la carte graphique. Ainsi, les clients peuvent définir si ils veulent utiliser cette méthode, ainsi que les paramètres (seulement la face avant ou seulement la face arrière).

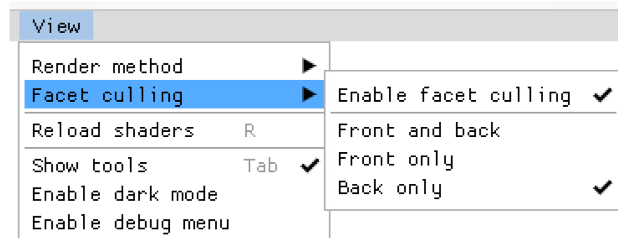


FIGURE 4.5 – Menu de la barre d'outils permettant de modifier le *facet culling*

Deux types de lampes ont été implémentés : les lampes directionnelles (ayant une intensité et une direction, appliquée à toute la scène) et les lampes

ponctuelles (ayant une intensité et une position, appliquée à une zone restreinte de la scène).

Les lampes sont stockées sous forme d'objets à part entière dans l'objet `Scene` et leurs données préparées pour être envoyées à la carte graphique sont stockées dans les objets `Renderer` : ils seront intégrés aux shaders à l'aide de plusieurs tableaux uniformes dont la taille est passée à la compilation du shader (voir section 4.1.6).

À la demande des clients, nous devions ajouter des raccourcis clavier permettant de rajouter ou de supprimer une lampe ponctuelle à la scène. Les lampes ponctuelles devaient être uniformément réparties sur la sphère englobante de la scène. Pour tester les lampes et leur gestion dans les shaders du *forward shading*, nous avons implémenté deux raccourcis clavier permettant d'ajouter une lampe directionnelle (touche `L`, la direction est fixe) ou une lampe ponctuelle (touche `U`, position aléatoire sur la sphère englobante de la scène).

Là encore, par manque de temps, nous n'avons pas pu implémenter la répartition uniforme des lampes ponctuelles sur la sphère englobante de la scène.

Pour la caméra, nous sommes repartis des fichiers `camera.cpp` et `trackball.cpp` que nous avions utilisé pendant nos TD. Nous avons fusionné les deux fichiers en un seul (`camera.cpp`) et implémenté différentes fonctions utiles à l'application (comme la gestion de coordonnées polaires sur la sphère englobant la scène).

Nous avions décidé d'implémenter différents modes de déplacement pour la caméra : le mode *autour de l'objet*, où la caméra se déplace sur la sphère englobant de l'objet et pointe en continu vers le centre de l'objet, et le mode *caméra libre*, où la caméra peut être déplacée librement dans l'espace.

La caméra en mode *autour de l'objet* peut être contrôlée avec les touches directionnelles pour les déplacements et avec les touches `O` et `P` pour le zoom. En mode *caméra libre*, les déplacements peuvent être effectués à l'aide des touches `Z`, `Q`, `S` et `D` sur les axes X et Z du repère de la caméra, et l'orientation de la caméra peut être modifiée avec le clic gauche et le déplacement de la souris. Le passage d'un mode à l'autre se fait automatiquement.

Concernant le passage des matériaux à la carte graphique, les shaders ne permettent pas d'accéder à des attributs de faces sur lesquelles ils sont

appelés, contrairement à ceux des sommets : il a donc fallu trouver un autre moyen pour y accéder. Comme pour les lampes, on aurait pu passer par un tableau 1D passé en tant que paramètre uniforme, mais cela est limité par l'espace mémoire alloué à ce genre de paramètres et qui s'avère trop limité pour passer un tableau de plusieurs millions de valeurs (même réduits à un octet par valeur). Avec l'aide des clients, nous avons décidé de passer par l'utilisation d'un *Texture Buffer Object*, qui permet en plus de garder ces données dans la mémoire de la carte graphique et ne pas à avoir à les repasser à la carte à chaque appel de shader en ayant besoin. Ce genre d'objet est également limité par l'espace alloué par la carte graphique dans sa mémoire, mais pas autant que pour les paramètres uniformes. Au final, on peut accéder aux données dans la texture comme s'il s'agissait d'un tableau 1D.

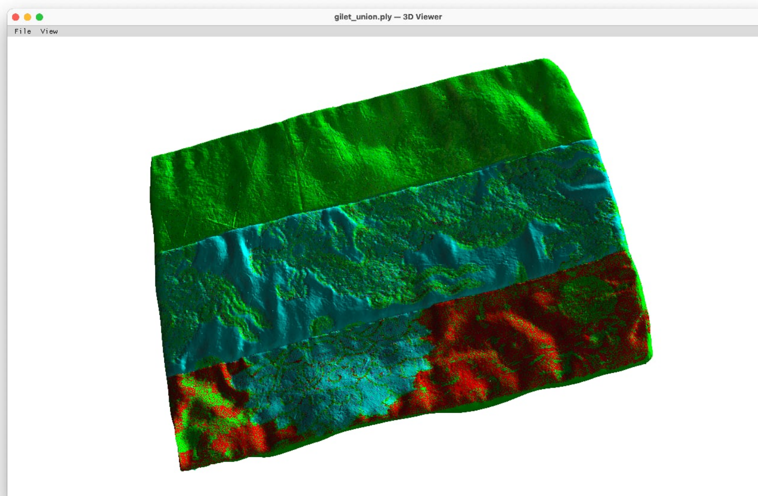


FIGURE 4.6 – Capture d'écran des fonctions matériaux de tests (qui retournent des couleurs fixes) pour vérifier le bon passage des identifiants des matériaux à la carte graphique et leur bon accès

4.1.6 Gestion des shaders

Pour la gestion des shaders, nous sommes repartis du fichier `shader.cpp` que nous avions utilisé pendant nos TD, et l'avons adapté à nos besoins. Les shaders sont gérés par la classe `ShadersReader` qui permet de les interpréter, de les appeler dans la carte graphique (s'ils ont été compilés), d'accéder à leurs codes source et de les recharger facilement. Un objet `ShadersReader` gère un programme complet, soit deux shaders (*vertex shader* et *fragment*

shader) à la fois.

Nous avons ajouté un système de tags dans les shaders qui sont interprétés et remplacés au chargement du fichier avant sa compilation, afin de permettre à la fois d'ajouter directement dans le code le nombre de lampes (et donc la taille des différents tableaux passés en paramètres uniformes comprenant leurs données) via le tag `@define_macros` et d'ajouter et appeler du code externe.

À la demande des clients, les fonctions utilisées pour définir les matériaux (leur réponse optique en fonction de l'éclairage) sont spécifiées dans des fichiers séparés, afin de pouvoir être modifiées facilement et utilisées dans différents shaders sans duplication du code. Pour cela, nous avons mis en place des *« fichiers matériau »* (avec comme extension `.mat`) qui ne sont que des fichiers écrits en GLSL (comme les shaders) et ne comportant qu'une fonction pouvant prendre n'importe quel paramètre et qui retourne une intensité lumineuse (sous la forme d'un `vec3`).

```
1 vec3 myMaterial(vec3 intensity, vec3 direction, vec3 normal) {
2     return intensity * max(dot(normal, direction), 0.);
3 }
```

FIGURE 4.7 – Exemple de contenu d'un « fichier matériau » simple, calculant l'impact d'une lampe directionnelle d'intensité `intensity` et de direction `direction` sur une face de normale `normale`

L'ajout de ces fichiers dans le code source du shader se fait à l'aide du tag `@define_materials` : la liste des fichiers fournis comme étant les matériaux à utiliser (sous la forme d'un objet `MaterialList` dans le code pour faciliter son partage) permet de savoir quels fichiers sont appelés afin de les ajouter au code. À ce moment là, le programme va préparer leurs appels dans la suite du shader en listant les noms des fonctions et des paramètres : la ligne `vec3 myMaterial(vec3 intensity, vec3 direction, vec3 normal)` sera mémorisée en tant que `myMaterial(intensity, direction, normal)`.

Pour appeler ces fonctions ajoutées, le shader utilise le tag `@call_materials` : s'il n'y a qu'un seul appel préparé, il sera directement ajouté au code, sinon, une condition `if`/`elif`/`else` est ajoutée pour appeler la fonction correspondante en fonction de l'identifiant du matériau de la face. Pour intégrer l'appel à la fonction dans du code (pour pouvoir gérer toutes les lampes dans

la condition plutôt que de devoir re-tester la condition pour chaque lampe par exemple), le programme recherche le tag `@mat` dans le reste de la ligne. L'identifiant du matériau est automatiquement cherché dès le premier appel à `@call_materials`.

```
7  @define_materials
8
9  void main() {
10 float a = .2;
11 @call_materials out_color = @mat;
12 }
```

(a) Code source d'origine du shader

```
7  vec3 mat1() { return vec3(1.); }
8  vec3 mat2(float a) { return vec3(a); }
9  vec3 default() { return vec(1., 0., 0.); }
10
11 void main() {
12 float a = .2;
13 uint material = uint(texelFetch(face_material, gl_PrimitiveID).r);
14 if (material == 1) { out_color = mat1(); }
15 else if (material == 2) { out_color = mat2(a); }
16 else { out_color = default(); }
17 }
```

(b) Code source après interprétation des tags

FIGURE 4.8 – Exemples de tags interprétés par `ShadersReader` dans un shader

4.2 Décimateur

Le but du décimateur étant de décimer un maillage (de réduire son nombre de faces), alors le but premier est de choisir une ou des bibliothèques capables de gérer des maillages conséquents (de l'ordre de 25M faces) et de les traiter avec un temps acceptable.

Cet outil est créé en *C++* comme le visionneur, et n'est utilisable qu'en ligne de commande, car certaines données doivent nécessairement être envoyées à cette application.

4.2.1 Paramètres en entrée

Pour ce qui est de son utilisation, il suffit d'envoyer un fichier PLY d'entrée existant, un coefficient de décimation supérieur à 1 et un fichier PLY de sortie pour que l'application fonctionne. Pour assurer la validité des données, la bibliothèque *CLI11* a été utilisée, ce qui assure l'existence du fichier PLY d'entrée, les extensions des fichiers ainsi que la validité du facteur de décimation.

En plus de ces données, il est possible de passer une *graine de randomisation* en paramètre, ce qui permet d'avoir un résultat de décimation *déterministe*, comparé à l'utilisation d'une graine de randomisation aléatoire.

4.2.2 Décimation

Quant à la décimation elle-même, le chargement du fichier PLY, sa manipulation et sa sauvegarde sont gérés par la bibliothèque *PMP*, qui permet de charger un grand nombre de types de maillages dans l'application. L'application calcule le nombre de faces du modèle final après décimation, puis la fonction *collapse* de *PMP* est utilisée, ce qui permet de supprimer un *sommet* en contractant une *demi-arête*, ce qui fusionne le sommet d'origine de la demi-arête avec son sommet de destination.

Cette opération permet de supprimer un sommet à tous les coups, mais supprime deux arêtes et une face si le sommet fait partie du bord du modèle, ou bien trois arêtes et deux faces le cas échéant.

Il est possible que l'opération *collapse* soit illégale si le résultat forme un maillage non-variété. Dans ce cas-ci, on essaye de voir si la demi-arête inverse peut être réduite à la place. Si l'opération ne marche toujours pas, alors seule l'arête traitée est marquée, et le processus de recherche d'arêtes pouvant être réduites est repris.

Une fois que le nombre d'arêtes maximum du nouveau maillage est atteint, le fichier PLY est sauvegardé, et est maintenant utilisable par une autre application.

Une optimisation envisagée fut, dans le cas où l'opération est valide, de marquer toutes les arêtes connectées au sommet résultant pour éviter un traitement de ces arêtes, afin d'éviter une concentration de suppression de faces dans une zone donnée du maillage, cependant cette optimisation n'a pas pû être mise en place par manque de temps.

4.3 Script de synthèse

La fonction du script de synthèse est d'exécuter le visionneur et le décimateur pour obtenir divers changements de données pendant l'exécution du programme. Un objectif important de ce programme est de permettre aux utilisateurs de comprendre intuitivement quelle méthode de rendu est la plus appropriée pour telle ou telle forme de données, afin que la meilleure méthode de rendu puisse être choisie en fonction de son efficacité. Pour atteindre cet objectif, nous utilisons le script pour réaliser un *benchmarking* (ou test de performance) qui compare les performances obtenues par différentes méthodes de rendu sur différents objets cibles dans différentes conditions d'éclairage.

Ce script est écrit en *Python*, qui accepte différents paramètres d'entrée, permettant aux utilisateurs de personnaliser les conditions à comparer.

Il est aussi potentiellement modifiable par quiconque veut adapter le processus de test, car le code *Python* est assez facile d'utilisation.

4.3.1 Paramètres en entrée

Ce script a des conditions d'exécution intégrées. Il sélectionne un dossier spécifique, en extrait le chemin d'accès au fichier et forme une ligne de commande avec des paramètres définis par l'utilisateur. En exécutant ces lignes de commande, nous pouvons obtenir le nombre d'images par seconde (ips) sous différentes conditions d'entrée.

Le script peut être lancé sans paramètre d'entrée. Dan ce cas-ci, le script obtient le nombre d'ips de chaque fichier exécuté en mode simple. Les utilisateurs peuvent également utiliser `--fs (forwardShading)` et `--ds (deferredShading)` pour ajuster le mode de rendu du visionneur, et utiliser `--pl (pointLight)` pour définir le nombre de lumières ponctuelles dans la scène principale du visionneur.

4.3.2 Benchmarking

Après avoir obtenu le nombre d'ips, nous pouvons utiliser Python pour générer différents types de graphiques pour le *benchmarking*. En supposant que nous avons utilisé le *décimateur* pour générer plusieurs versions du même fichier original de différentes tailles, et que le script a été exécuté avec ces fichiers, nous pouvons utiliser le script pour générer un histogramme afin

d'observer visuellement la vitesse à laquelle chaque fichier s'exécute, afin de comprendre dans quelle mesure le fichier d'origine doit être réduit pour être rendu en temps réel.

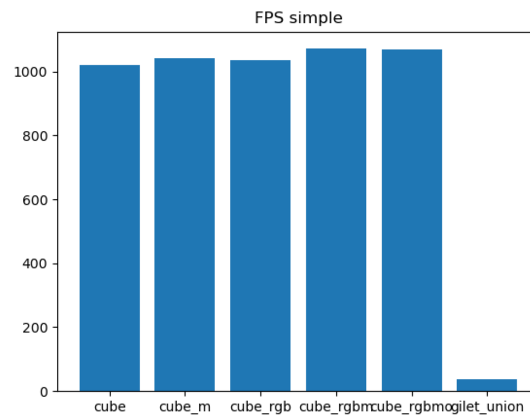


FIGURE 4.9 – Histogramme sur les vitesses de rendu avec différents objets, rendu avec *Windows*

De plus, en ajoutant des paramètres d'éclairage et des méthodes de rendu, la courbe générée peut nous permettre de visualiser l'atténuation de la vitesse suivant le nombre de lumières ponctuelles dans la scène.

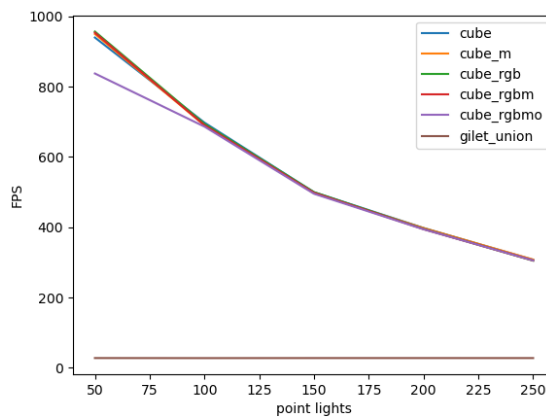


FIGURE 4.10 – Courbes décrivant le nombre d'images par seconde suivant le nombre de lumières ponctuelles dans le scène, rendu avec *Windows*

Chapitre 5

Réalisations

5.1 Adaptation par rapport au prévisionnel

5.1.1 Problèmes rencontrés

L'un des gros problèmes vient de la conception du projet et de l'organisation des tâches : en effet, il était originellement prévu d'implémenter les rendus via *forward shading* et via *deferred shading* l'un après l'autre. Hors, les méthodes de rendus sont en constante évolution pour pouvoir gérer les autres fonctionnalités à ajouter par la suite (comme la gestion des matériaux et la création des shaders par matériau) : il était préférable alors de ne s'attarder que sur l'un avant d'attaquer l'autre, afin à la fois de pouvoir tester les nouvelles fonctionnalités, mais également pour éviter de devoir modifier de façon similaire plusieurs classes, ce qui aurait ammener à des lenteurs lors du développement.

Au final, alors qu'elle était prévue pour être terminée en deux jours, la tâche **[B04]** correspondant à l'implémentation du *forward shading* nous a pris un peu moins d'une semaine, mais a évolué en même temps que les autres tâches étaient effectuées. Couplé aux problèmes rencontrés lors de la tentative d'utilisation de *globjects* (voir section 4.1.5) qui nous a pris une semaine avant que l'on se mette à utiliser *glbinding*, mais également que d'autres tâches avaient été oubliées (comme la création des classes `Light` et `Scene`), cela a complétement décalé notre planning.

On peut aussi dire que chacun des membres de l'équipe a rencontré un problème soit matériel, soit logiciel au cours du projet, ce qui nous a mis en retard de plusieurs jours vers la fin du projet : *Nicolas* a eu un problème d'écran à changer sur son ordinateur portable, ce qui a pris une bonne partie

d'une semaine, *Fan* a eu une mise à jour défectueuse sur sa version de *Windows*, et *Rhenaud* a eu un problème de carte graphique empêchant le bon fonctionnement du logiciel sur sa machine.

5.2 Résultats obtenus

Pour tester les résultats de vitesse de rendu avec des fichiers de différentes tailles, nous utilisons notre fichier python de benchmarking pour exécuter le programme et passer 5 maillages différents au programme. Ces maillages sont fournis par le client, le plus petit fichier gilet_union.ply a 28 millions de sommets et 29 millions de faces, tandis que le plus grand est chemise_union.ply, composé de 40 millions de sommets et 80 millions de faces. Sous MacOS, nous obtenons les résultats suivants :

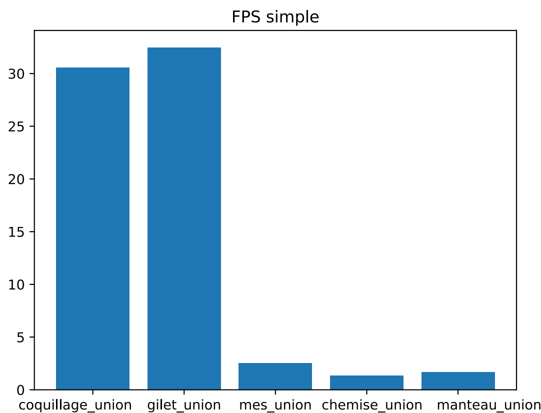


FIGURE 5.1 – Histogramme du nombre d'images par seconde obtenu avec chaque maillage dans le scène, rendu avec la machine sous *macOS* décrite section 3.2

Nous observons qu'à l'exception de coquillage_union.ply et gilet_union.ply qui peuvent être rendus en temps réel sur cette machine, aucun des trois autres fichiers ne le peut. De plus, nous devons tenir compte du fait qu'il n'y a pas de source de lumière ajoutée à la scène lors de ce test, donc si des sources de lumière sont ajoutées à celle-ci, le nombre d'ips reçu sera encore plus petit.

Cependant, la machine utilisée a des performances bien moindres comparées aux machines du CREMI, qui nous avaient été assignées pour tester la

rapidité de notre programme, donc une étude plus poussée pourrait donner des résultats bien plus encourageants.

Essayons donc un test de performance de ces fichiers en forward shading sous différentes conditions d'éclairage :

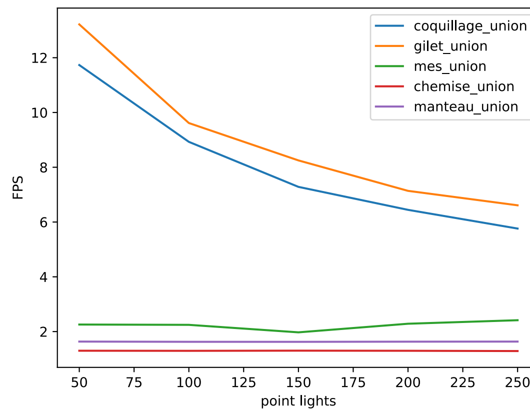


FIGURE 5.2 – Courbes décrivant le nombre d'images par seconde suivant le nombre de lumières ponctuelles dans la scène, rendu avec la machine sous *macOS* décrite section 3.2

Les résultats expérimentaux montrent que le taux de rafraichissement va généralement diminuer avec l'augmentation du nombre de lumières ponctuelles. Seule la courbe de mes_union ne semble pas conforme à cette conclusion, ce qui peut être causé par un possible ralentissement du programme dans un cas qui nous est inconnu à cette date.

En même temps on voit aussi que lorsque le nombre de lumières ponctuelles dans la scène est à 50, aucun maillage ne peut être rendu en temps réel sur cette machine. Afin d'atteindre l'objectif de rendu en temps réel, nous devons donc possiblement soit décimer le maillage d'entrée, soit utiliser le deferred shading, qui a tendance à être plus rapide que le simple shading.

Conclusion

Les deux mois passés sur la réalisation du projet ont été assez chaotiques, car nous avons sous-estimé la quantité de travail à effectuer sur la totalité de l'application. Comme nous avons pu le présenter, même si le but principal de l'application était d'afficher des modèles 3D détaillés en temps réel, le réel but de l'application est de pouvoir effectuer des tests de performance, ce qui a rajouté de nombreuses problématiques supplémentaires à notre projet de fin d'études.

Dû à ces changements et à un manque de cohésion de l'équipe en général, l'application est à un stage de *« proof of concept »* pour le moment, ce qui veut dire que même si les bases de l'application sont là et que la majorité des cas d'utilisations sont couverts, il faudrait du travail supplémentaire afin de parfaire le projet et de couvrir les cas d'utilisations que notre ensemble d'applications ne nous permet pas de couvrir à la date du rendu final.

En effet, même si le *visionneur* est dans l'ensemble assez complet, il reste des fontionnalités qui ont été discutées mais que nous n'avons pas pu implémenter.

Quant au décimateur et au script de benchmarking, ces derniers sont à un stade embryonnaire car leur développement n'a commencé que tard dans l'évolution du projet. Il en va sans dire que l'aboutissement de ces parties du projet aurait été bien meilleur si les modules du projet avaient été séparés plus tôt. Nous avons perdu beaucoup de temps sur le visionneur, ce qui a repoussé le début de la création du décimateur et du script de benchmarking.

Un autre axe de retard du projet est que la date de rendu s'est comporté comme un trou noir : plus on se rapprochait d'elle, plus le temps était dilaté, et donc moins les tâches étaient faites rapidement. De par ce fait, le stress de la rédaction du rapport et de la préparation de la présentation s'est immiscé dans notre efficacité, ce qui a forcément réduit la vitesse de développement

vers la fin du temps imparti.

Améliorations possibles

Visionneur

Comme dit dans la section 4.1, les retards pris pendant le développement nous ont obligés à faire l'impasse sur différentes fonctionnalités, principalement :

- l'implémentation de la méthode de rendu du *deferred shading* ;
- l'utilisation poussée des fichiers de configuration pour pouvoir re-lancer un *benchmark* ;
- et l'implémentation de l'ajout de lampes ponctuelles réparties uniformément sur la sphère englobante de la scène.

Nous avions également discuté avec les clients d'autres fonctionnalités que nous aurions aimé avoir le temps d'implémenter, comme la gestion des animations de caméra et les changements automatisés du nombre de lampes ponctuelles réparties uniformément dans la scène.

Différents aspects de l'application pourraient être améliorés, notamment :

- la gestion par l'utilisateur du *tone-mapping* : pour l'instant, les couleurs sont normalisées par rapport à la valeur maximum ou par 2^{18} (donnant de bons résultats sur la plupart des maillages de tests données par les clients pour tester l'application) ;
- la gestion de la scène et du maillage : au lieu d'adapter la scène au changement de maillage, utiliser une matrice de transformation sur le maillage dans les shaders afin de modifier sa position et sa taille permettrait d'utiliser les mêmes lampes et la même caméra ;
- la gestion des entrées clavier et souris : pour l'instant, toutes les entrées correspondant aux déplacements de la caméra l'affecte, même si le `ViewerModule` n'est pas l'élément actif ;
- la gestion de la caméra et de ses différents modes : le changement de mode se fait automatiquement lors de l'utilisation des contrôles de l'un et de l'autre ;
- l'ajout d'arguments, de paramètres dans les fichiers de configuration et de raccourcis clavier.

Quelques bugs ont également été rencontrés par l'équipe, dont certains n'ont pas encore pu être corrigés :

— Après un changement de maillage, les matériaux peuvent ne pas être les bons ;
— Sous certaines conditions encore indéterminées, le programme peut fermer inopinément à son lancement à l'ouverture d'un fichier généré par le décimateur.

Décimateur

Comme énoncé plus haut, le retard de cette application a fait que certains points n'ont pas pû être explorés avec le temps qu'il nous restait. Malgré ceci, il est possible de dégager certains points d'amélioration avec le décimateur.

Tout d'abord, l'itération courante de l'application utilise les fonctions de PMP afin de charger les données du fichier *PLY* d'entrée et de sauvegarder les données du fichier *PLY* de sortie, qui sont basés sur la bibliothèque *rply*. Cependant, cette dernière ne gère pas les données autres que le nombre, l'emplacement et le lien entre les faces, cotés et sommets du maillage, ce qui veut dire que les données supplémentaires telles que la couleur ou l'ID de matériau de chaque sommet ou face n'est pas traité.

Pour pallier à ce problème, l'inclusion d'une autre bibliothèque de chargement et de sauvegarde de fichier *PLY* serait bienvenue, suivi d'un lien manuel entre les données de cette bibliothèque et du système de gestion de maillages de PMP, dans le sens de la lecture comme de l'écriture. Cela permettrait d'assigner les données de couleur et de matériau d'ID à chaque élément, et de pouvoir les manipuler et les garder après la sauvegarde du fichier décimé.

De plus, l'algorithme de décimation utilisé dans l'application n'est pas idéal : l'utilisation directe de la fonction *collapse* de la bibliothèque PMP peut donner des résultats peu satisfaisants, mais après un retour avec les clients, la méthode utilisée par l'algorithme *SurfaceSimplification* de la classe *pmp*, qui réduit le nombre de faces d'un maillage à un nombre donné en paramètre. Cette méthode correspond exactement à ce que le décimateur est censé faire, et essaye aussi de réarranger les sommets du maillage pour conserver sa forme originale du mieux que possible, et pourrait donc remplacer *collapse* pour possiblement de meilleures performances et un meilleur maillage en sortie.

Enfin, nous avons vu un ralentissement général de la décimation sur un gros volume de données (plus exactement sur une décimation de facteur 10

avec un maillage d'entrée ayant un nombre de faces avoisinant 30 millions), malgré un appel récurrent à une fonction de *garbage collection*, qui gère la suppression permanente de données (sommets, côtés et faces) non utilisées par l'application, ce qui devrait normalement éviter ce type de ralentissement. Il est possible que la résolution du problème précédent induise la résolution de ce problème, sinon, il serait possible de regarder plus en détail ce qui ralentit autant l'application avec un nombre de données accru.

Script de benchmarking

Le script *Python* n'est pas parfait. Les données auxquelles il se réfère actuellement ne sont que des images par seconde, il y a donc encore des domaines qui peuvent être améliorés comme suit :

— l'introduction de l'analyse comparative de l'utilisation du processeur

— l'introduction de l'analyse comparative de l'utilisation de la mémoire

— l'ajout d'une analyse comparative pour le deferred shading

— l'ajout d'une analyse comparative pour le forward shading et le deferred shading sous différents nombres de lumières ponctuelles.

Table des figures

1.1 Contenu d'une scène 3D simple 4
1.2 Comparaison du rendu par tracé de rayon et par rastérisation[3] 5
1.3 Exemple de pipeline 3D standard[4] 6
1.4 Exemple d'en-tête de l'un des fichiers fournis par les clients : `gilet_union.ply` 9

2.1 Première maquette de l'interface du *visionneur*, présentée aux clients le mercredi 3 février 2022 14

3.1 Gantt prévisionnel 28
3.2 Gantt effectif 29

4.1 Capture d'écran du visionneur 30
4.2 Diagramme d'architecture montrant l'organisation des différentes classes liées au rendu 3D 31
4.3 Utilisation des fonctionnalités *Dear ImGui* dans le visionneur, avec une barre d'outils et différentes sous-fenêtres créées à partir des modules 34
4.4 Menu de la barre d'outils permettant de modifier la méthode de rendu 36
4.5 Menu de la barre d'outils permettant de modifier le *facet culling* 36
4.6 Capture d'écran des fonctions matériaux de tests (qui retournent des couleurs fixes) pour vérifier le bon passage des identifiants des matériaux à la carte graphique et leur bon accès 38
4.7 Exemple de contenu d'un « fichier matériau » simple, calculant l'impact d'une lampe directionnelle d'intensité `intensity` et de direction `direction` sur une face de normale `normale` . . . 39
4.8 Exemples de tags interprétés par `ShadersReader` dans un shader 40
4.9 Histogramme sur les vitesses de rendu avec différents objets, rendu avec *Windows* 44

4.10 Courbes décrivant le nombre d'images par seconde suivant le nombre de lumières ponctuelles dans le scène, rendu avec *Windows* 44

5.1 Histogramme du nombre d'images par seconde obtenu avec chaque maillage dans le scène, rendu avec la machine sous *macOS* décrite section 3.2 46
5.2 Courbes décrivant le nombre d'images par seconde suivant le nombre de lumières ponctuelles dans la scène, rendu avec la machine sous *macOS* décrite section 3.2 47
5.3 Sujet du projet 60
5.4 Exemple de fichier PLY représentant un cube à l'aide de faces triangulaires, avec des couleurs par sommet et des matériaux par face (un matériau par face carrée du cube) 61

Bibliographie

[1] Histoire et archéologie : vérifier les données et visualiser le passé. https://www.inria.fr/fr/reproduire-lapparence-de-materiaux-textiles-un-defi-technologique, February 2021.

[2] Reproduire l'apparence de matériaux textiles, un défi technologique. https://www.inria.fr/fr/reproduire-lapparence-de-materiaux-textiles-un-defi-technologique, March 2020.

[3] What's the difference between ray casting, ray tracing, path tracing and rasterization ? Physical light tracing... https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html, October 2019.

[4] Chua Hock Chuan. 3D graphics with OpenGL : Basic theory. https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html, July 2012.

[5] MeshLab. https://www.meshlab.net.

[6] PFE 3D Viewer. https://github.com/Nimoka/PFE-3D-Viewer.

[7] PFE 3D Decimator. https://github.com/Nimoka/PFE-3D-Decimator.

[8] TOML. https://fr.wikipedia.org/wiki/TOML, January 2021.

[9] TOML : Tom's Obvious Minimal Language. https ://toml.io/en/.

[10] Toru Niina. toml11 (v3.7.0). https://github.com/ToruNiina/toml11/tree/647381020ef04b5d41d540ec489eba45e82d90a7, May 2021.

[11] CLIUtils. CLI11 : Command line parser for C++11 (v2.1.2). https://github.com/CLIUtils/CLI11/tree/70f8072f9dd2292fd0b9f9e5f58e279f60483ed3, October 2021.

[12] Omar Cornut. Dear ImGui (v1.87). https://github.com/ocornut/imgui/tree/c71a50deb5ddf1ea386b91e60fa2e4a26d080074, February 2022.

[13] GLFW. GLFW (v3.3.6). https://github.com/glfw/glfw/tree/7d5a16ce714f0b5f4efa3262de22e4d948851525, December 2021.

[14] Vilya Harvey. miniply (version du 12 juin 2021). https://github.com/vilya/miniply/tree/d7005b901e64c66bfb39e88c4882d006dcf70628, June 2021.

[15] Maciej Halber. Ply file I/O benchmark. https://github.com/mhalber/ply_io_benchmark, November 2020.

[16] Dimitri Diakopoulos. tinyply (v2.1). https://github.com/ddiakopoulos/tinyply/tree/bed8cec40d4796fee45a31eb534e5ebc9d9f9fd3, June 2018.

[17] Dimitri Diakopoulos. tinyply (v2.3). https://github.com/ddiakopoulos/tinyply, February 2022.

[18] CG Internals. glbinding. https://glbinding.org, 2018.

[19] CG Internals. glbinding (v3.1.0). https://github.com/cginternals/glbinding/tree/28d32d9bbc72aedf815f18113b0bd3aa7b354108, April 2019.

[20] CG Internals. globjects. https://globjects.org, 2018.

[21] CG Internals. globjects (v1.1.0). https://github.com/cginternals/globjects/tree/2f0c753c8647a6db9c201747ebc43ff9cc0ada76, February 2017.

Glossaire

benchmark	Test dont l'objectif est de déterminer la performance d'un système informatique, plus précisément ici la fréquence de rafraichissement de l'application dans des cas donnés
depth buffer	Masque stockant la distance entre la caméra et le fragment le plus proche pour chaque zone de l'écran
décimation	Suppression d'un certain nombre de faces d'un maillage
décimer	Supprimer un certain nombre de faces d'un maillage
FPS	« Frames Per Second », ou nombre de rafraichissement de l'écran par seconde
fragment	Élément indivisible destiné, à l'issue de son traitement dans le pipeline graphique, à déterminer la couleur d'un pixel
maillage	Structure de données géométrique permettant de représenter des subdivisions de surface à l'aide d'un ensemble de polygones
scène	Environnement dans lequel différents éléments 3D peuvent être agencés
shader	Programme informatique utilisé en image de synthèse pour paramétrer une partie du processus de rendu réalisé par une carte graphique ou un moteur de rendu logiciel
temps réel	Faire en sorte que le programme tourne de manière assez fluide, avec au moins 30 FPS
texture	Image en deux dimensions que l'on va appliquer sur une surface ou un volume en trois dimensions de manière à habiller cette surface ou ce volume

Acronymes

ANR	Agence Nationale de la Recherche
API	Application Programming Interface
CNRS	Centre National de la Recherche Scientifique
CREMI	Centre de Ressources pour l'Enseignement des Mathématiques et de l'Informatique
CSV	Comma-Separated Values
GLSL	OpenGL Shading Language
Inria	Institut National de Recherche en Informatique et en Automatique
ips	images par seconde
LoD	Level of Detail
MEB	Musée d'Ethnographie de l'université de Bordeaux
MOCPI	Méthodes et Outils de Conduite de Projets Informatiques
PLY	Polygon File Format
PMP	Polygon Mesh Processing
TD	travaux dirigés
TOML	Tom's Obvious Minimal Language

Annexes

Projets de fin d'Études 2021-2022 – Master 2 IIS

Visualisation temps-réel de modèles détaillés

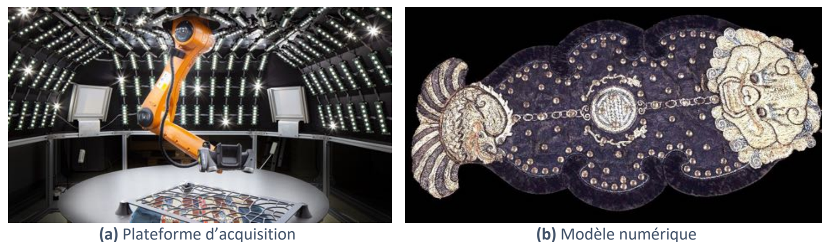
Encadrants

Romain Pacanowski – romain.pacanowski@inria.fr
Pierre Bénard – pierre.benard@u-bordeaux.fr

Sujet

La Coupole[1] (Figure 1.a) est une plateforme d'acquisition de la forme (géométrie) et de l'apparence (matériaux) développée par l'équipe Manao. Elle est équipée d'un scanner 3D et d'un appareil photographique montés sur un bras robotisé à 6 axes, permettant de les positionner dans l'enceinte d'un dôme de 2,4m de diamètre avec une très grande précision. Les 1 080 LED blanches réparties uniformément sur le dôme sont individuellement contrôlables, ce qui permet le choix de la direction d'éclairage. L'ensemble permet de capturer l'apparence d'objet chaque 20e de millimètre, tout en étant capable de distinguer des détails de l'ordre de 100 µm, afin d'approcher l'acuité du regard humain. La numérisation génère un grand volume de données (de l'ordre de 20 To pour $1m^2$) qui doit être analysé pour reconstruire un modèle numérique exploitable.

Ce modèle (Figure 1.b) est composé d'un maillage triangulaire et d'un ensemble de matériaux (BRDF), chaque face du maillage possédant l'index de l'un de ces matériaux. Même si cette représentation est beaucoup plus compacte que les données brutes, elle reste très volumineuse ce qui rend sa visualisation en temps réel difficile.



(a) Plateforme d'acquisition **(b)** Modèle numérique

Figure 1: système d'acquisition de *La Coupole* et exemple de donné à visualiser

Description du travail

L'objectif de ce projet est d'implémenter un outil de visualisation temps-réel des modèles acquis à l'aide de *la Coupole*. Il a également pour objectifs d'évaluer, en termes de performances et d'impact mémoire, plusieurs algorithmes de rendu afin de choisir l'approche la plus efficace.

Nous souhaitons notamment comparer les techniques de rendu suivantes :

- *Forward rendering*
- *Deffered shading*
- *Clustered shading* (si le temps le permet)

en fonction du nombre de sources lumineuses (uniformément réparties sur un hémisphère).

Pour chaque technique, nous voulons en outre comparer une approche avec un « *uber shader* » pour tous les matériaux et une approche effectuant un rendu par matériau.

Le développement sera réalisé en C++ avec l'API OpenGL. D'autres bibliothèques pourront être utilisées pour faciliter l'implémentation (*globjects*[2] par exemple).

[1] https://www.inria.fr/fr/reproduire-lapparence-de-materiaux-textiles-un-defi-technologique
[2] https://github.com/cginternals/globjects

Figure 5.3 – Sujet du projet

```
ply
format ascii 1.0
element vertex 8
property float x
property float y
property float z
property float red
property float green
property float blue
element face 12
property list uchar uint vertex_index
property int id
end_header
0 0 0 1 1 1
0 0 1 1 1 .2
0 1 1 1 .2 .2
0 1 0 1 .2 1
1 0 0 .2 1 1
1 0 1 .2 1 .2
1 1 1 .2 .2 .2
1 1 0 .2 .2 1
3 0 1 2 4
3 0 2 3 4
3 1 6 2 1
3 1 5 6 1
3 5 1 0 6
3 6 5 7 2
3 0 4 5 6
3 7 5 4 2
3 3 7 4 3
3 3 4 0 3
3 7 3 2 5
3 7 2 6 5
```

FIGURE 5.4 – Exemple de fichier PLY représentant un cube à l'aide de faces triangulaires, avec des couleurs par sommet et des matériaux par face (un matériau par face carrée du cube)