Projet Pipeline

Faveraux Adrien

Master 1 Informatique

Sommaire

I) Présentation du projet

A) Présentation générale

Le projet consiste à la mise en place d'un pipeline graphique. Un pipeline graphique est l'un des constituants d'une carte graphique, elle a pour but de transformer une liste de sommet d'un fichier 3D en un fichier d'image pouvant être aisément affiché à l'écran.

Le but de ce projet est donc de recrée virtuellement ce principe avec des méthodes plus ou moins compliquer et perfectionner afin de nous faire comprendre la mécanique complexe qui anime les cartes graphiques que nous utilisons tous les jours.

Le projet doit être fait en C++ qui est un langage orienter objet utile afin d'organiser clairement les différentes étapes.

Une documentation technique qui constitue aussi notre cour a été fournie pour expliquer chacune des méthodes.

B) Oganisation d'un pipeline

Le pipeline est constitué d'un ensemble d'étapes :

- Une opération de Transformation, appeler communément vertex shader elle consiste a transformer les points qui sont dans un repère local dans le repère de l'observateur et y ajouter un effet de perspective.
- Une opération de rejet trivial qui se découpe en deux méthodes :
    1. Le culling consiste à regarder si la face que l'on veut calculer se trouve dans le champ de la caméra et pas cacher par une autre facette. Si la facette est cachée, elle est abandonnée, car elle est non visible pour accélérer les calculs.
    2. Le cliping est une opération qui a pour but de découper ou rejeter les facettes qui n'apparaissent pas dans l'écran ou n'apparait pas totalement à l'écran. Si les sommets sont hors du plan de l'écran, on les rejette, sinon on découpe la facette pour ne pas calculer ce qui sort de l'écran.
- Une opération de discrétisation qui consiste à tracer la facette a l'écran, il y a deux types de discrétisation :
    1. Le maillage, il faut tracer les contours de la facette
    2. Le remplissage, il faut remplir toute la facette.
- Une opération de pixel qui consiste en trois choses :
    1. La création d'un Zbuffer qui correspond a affiché seulement les pixels le plus près de l'observateur, cela permet d'afficher plusieurs objets et de ne pas réécrire par dessus un autre objet.
    2. Un calcul de l'ombrage au pixel grave au pixel Shader
    3. Écriture de la couleur du pixel à l'écran.

C) Methode mise en Place dans le cadre du cours

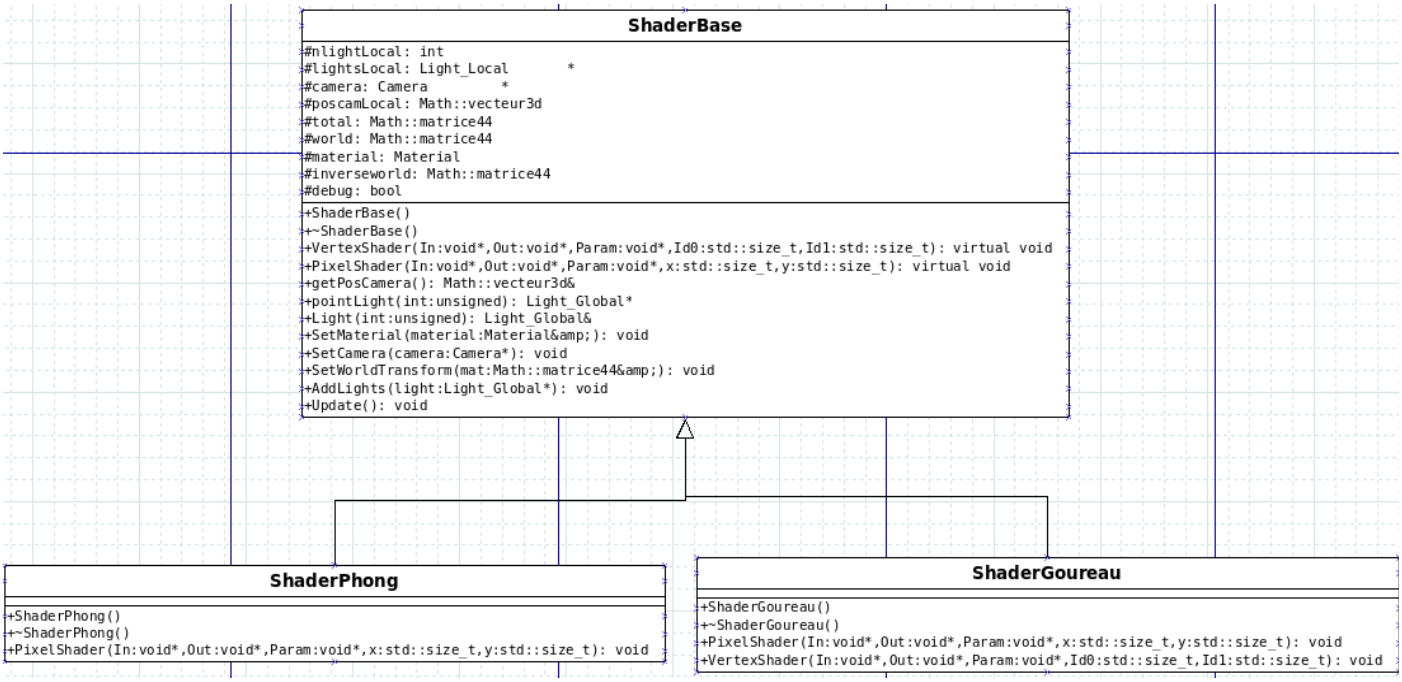
Nous avions le choix de l'implémentation pour effectuer les diverses tâches que le pipeline requière :

- Pour le Tracage du maillage, j'ai implémenté la méthode de Bresenham qui est une méthode optimisée pour faire cette tâche, car cette méthode utilise des coefficients entiers ce qui pour la machine est beaucoup plus facile à faire.
- Pour le remplissage, j'ai utilisé le remplissage naïf qui n'est pas une méthode optimisée, mais une méthode qui marche quand même sauf qu'elle n'utilise pas que des entiers pour fonctionner.
- Pour le culling il n'y avait qu'une seule implémentation possible et elle a été réalisée.
- Pour le cliping, seul un rejet au cas où les 3 points sont hors de l'écran a été réalisé. Aucun découpage de facette n'a été fait.
- Un modèle d'illumination élémentaire a été implémenté avec de la réflexion diffuse et de la réflexion spéculaire.
- Les deux modèles de shader du cours ont été implémenter, l'ombrage de goureau et l'ombrage de phong.
- Une gestion des couleurs a aussi été implémentée au niveau des objets, mais aucune implémentation de texture n'a été faite.

II) Preparation et réalisation du projet

A) Diagramme de classe

Shader :

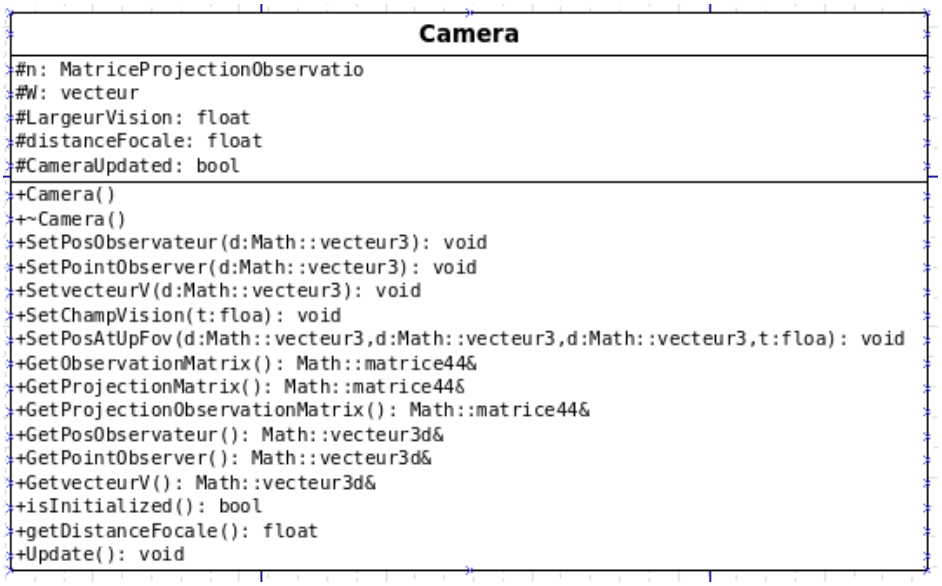


La classe Shader, peuvent être la plus importante des classes deux fonctions principales :

- VertexShader : Cette fonction permet de faire les transformations sur les points en lui appliquant la matrice Full qui contient toutes les transformations nécessaires à l'objet. Puis, la classe calculera ce qui est spécifique à un modèle de shader particulier ex : illumination des 3 points dans Gourreau.
- PixelShader : Cette fonction a pour but de renvoyer la couleur d'un pixel donner affin qu'il puisse être écrite à l'écran. Dans le modèle de base, aucun calcul n'est réalisé. Mais dans Phong et Gourreau, c'est ici qu'on va calculer l'illumination (au pixel pour phong et interpoler pour goureau)

Cette classe contient tous les objets nécessaires pour ses calculs : la camera, la matrice world, le material de l'objet, les lampes de la scène.

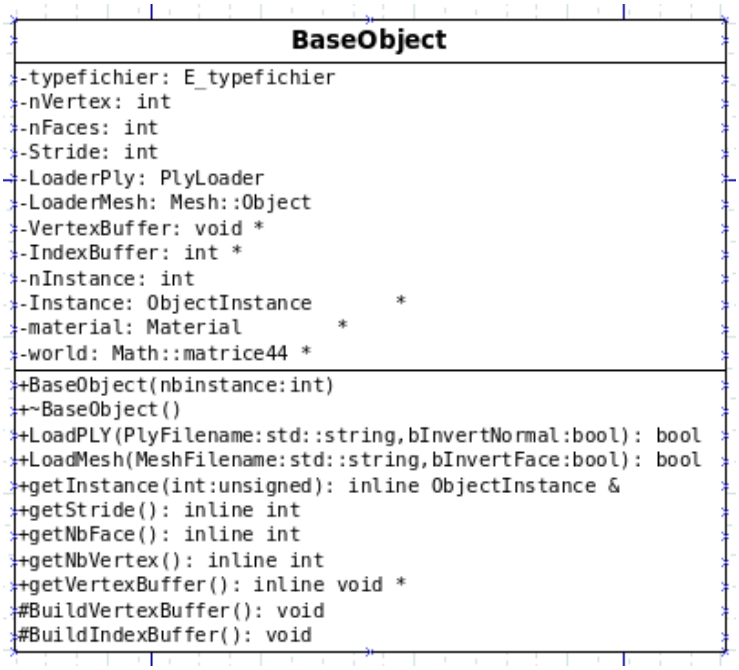
Classe Camera :



La classe Camera a pour fonction de calculer tout ce qui a un rapport avec la camera.

Elle aura pour but, à partie de la position de l'observateur, du point observer, de la direction haut et de la largeur de vision de calculer la matrice d'observation pour passer l'objet dans le repère de l'observateur et la matrice de projection afin de données de la perspective a la scène.

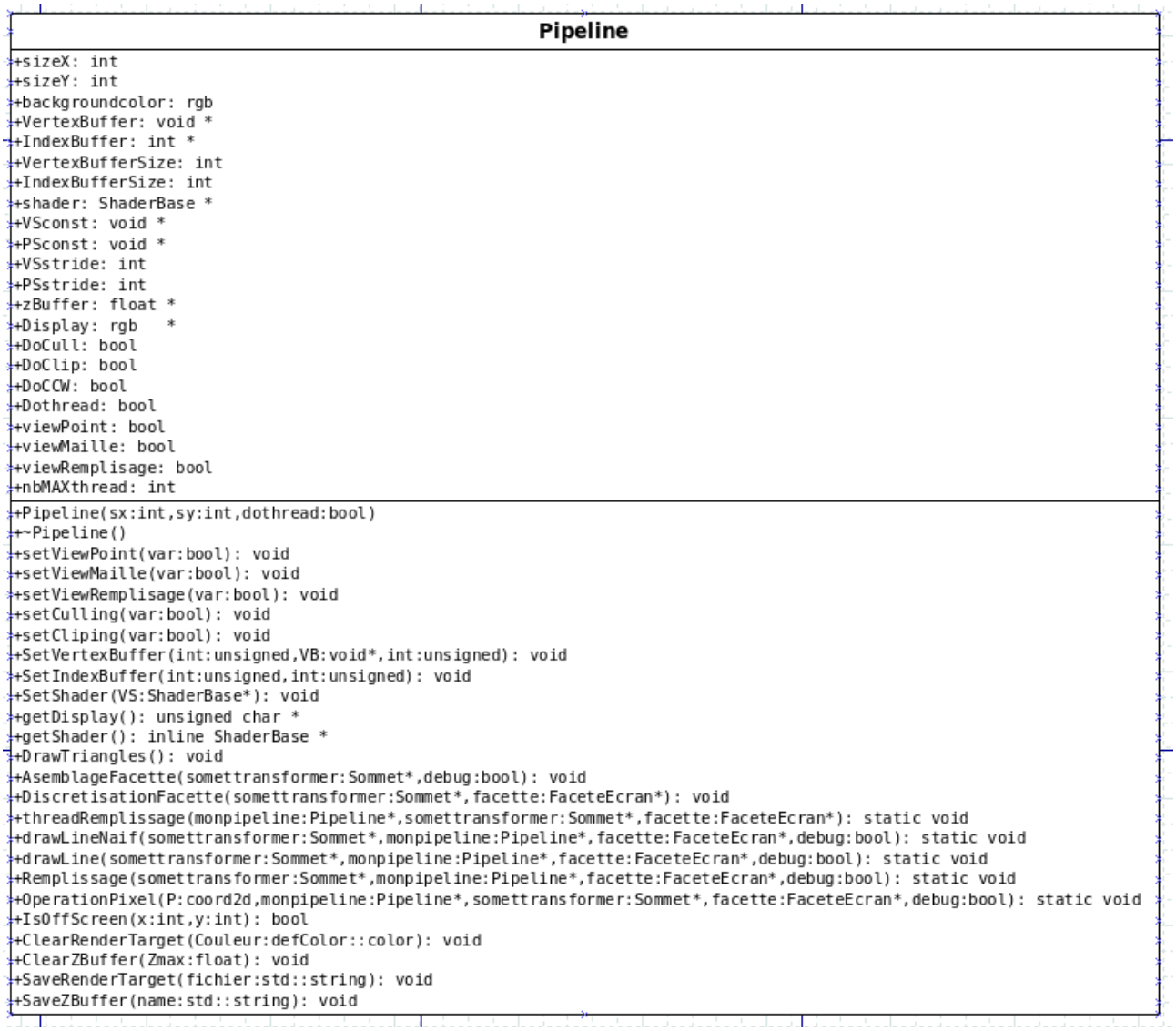
Classe BaseObject :



La classe BaseObject permet de représenter un objet 3D. Il y a donc un vertex et et pixel shader, une matrice de transformation et une structure material pour chaque instance de l'objet.

Ma classe Object se sert des deux loaders Mesh et PLY fournie dans ce cours. Pour pouvoir utiliser ces deux loaders, il a fallu que j'utilise une structure hybride pour standardiser les données 3D envoyées au pipeline.

Class Pipeline :



La classe pipeline a pour but d'organiser tous les calculs de sorte que les données de sortie d'une fonction soit les données d'entrer pour une autre. Comme vu sur le schéma précédent, la classe dispose de plusieurs fonctions principales : Draw Triangle-> AsemblageFacette -> Discretisation Facette -> Remplissage -> OperationPixel qui sont toutes les étapes du pipeline.

Elle dispose aussi de plusieurs fonctions utilitaires afin de pourvoir récupérer toutes les informations calculées de notre pipeline : SaveRenderTarget, SaveZBuffer

B) Organisation des structures

Pour la communication du pipeline, j'ai dû utiliser un ensemble de structure afin de rendre uniforme le tout et ainsi pouvoir faire un code propre. Ces structures sont présentées ici afin de comprendre au mieux l'implémentation et le pourquoi de chaque structure.

Structure de Sommet :

```
struct Sommet {
    /// Coordonnées du sommet
    Math ::vecteur4d pos;
    Math::vecteur3d pos_originel;
    /// Normale au sommet
    Math::vecteur3d norm;
    /// Couleur au sommet
    Math ::vecteur3d col;
    /// Coordonnées de texture (pas de modèle disponible)
    float       u, v;
    /// Couleur au sommet
    float intensité;
}
```

Cette structure est utilisée pour représenter un sommet dans le pipeline, le VertexBuffer contient un ensemble de ces sommets.

Structure FacetteEcran :

```
struct FaceteEcran
{
    point som1; /// Point x et y dans l'écran
    point som2;
    point som3;
};
```

À partir de l'assemblage des facettes, le pipeline passe les points dans le plan de l'écran. Une nouvelle structure est donc utilisée afin de pouvoir mieux visualiser ce fait. De plus les points en coordonnée 3D sont des double, ici, un som contient 2 entiers.

Structure PixelIN :

```
struct pixelIN
{
    Math ::coord2d P; /// Coordonner du pixel calculer
    FaceteEcran *facette; /// Les sommets dans le Plan 2D de l'écran
    Sommet *sommet; /// les sommets transformés et originaux en 3D
};
```

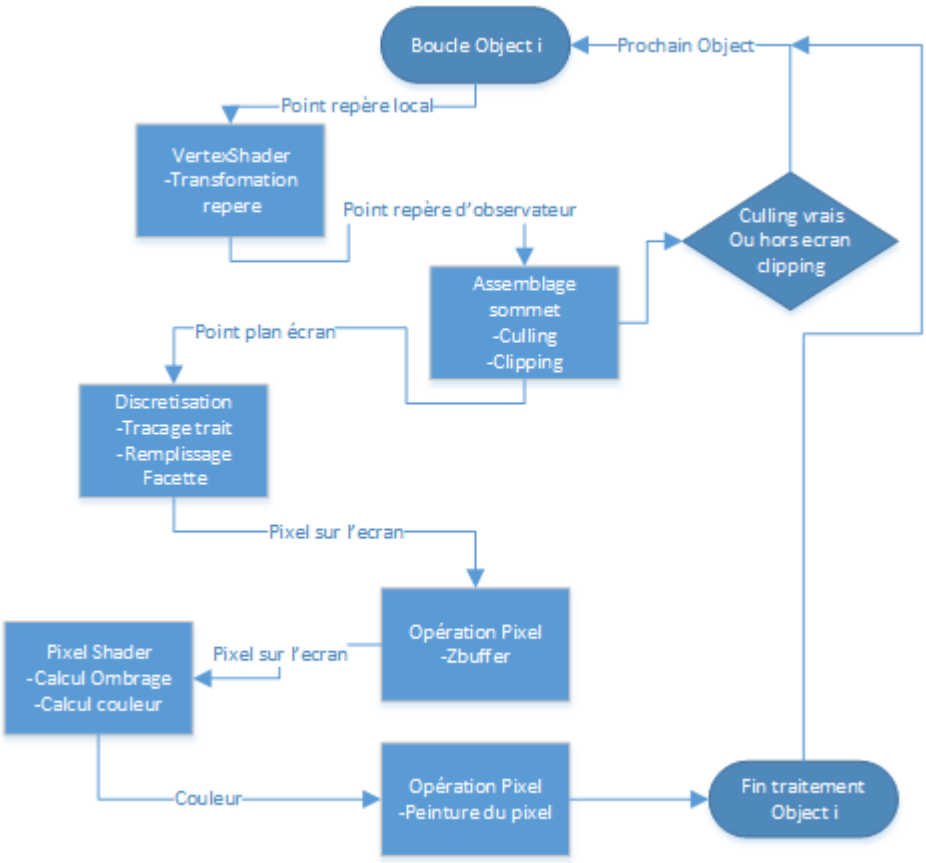
Cette structure est utilisée à l'entrer du pixel shader, le pixel shader peut avoir besoin d'un certain nombre d'informations, cette structure représente tous les besoins possibles pour les deux modèles de shader étudier Gourreau et Phong

Structure PixelOUT :

```
struct pixelOUT
{
    rgb col; /// Couleur du pixel
};
```

La sortie du pixel shader est toujours la couleur au pixel donc une couleur RGB

C) Flux d'exécution



D) Problematique entrer-sortie

Pour le pipeline, les structures servant à stocker les sommets en entrée ne me convenaient pas. Car lorsque j'ai voulu rendre compatible le pipeline avec les fichiers PLY et Mesh, ils utilisaient des structures différentes. J'ai donc créé une structure contenant tous les champs possibles pour stocker les sommets du pipeline. (voir classe Object et les structures) On a donc au chargement d'un objet le choix entre deux types de fichiers, PLY ou Mesh, ils sont copiés directement dans la structure en fonction de leur provenance, cela a simplifié grandement la conception du pipeline, car à la place d'attendre une structure de taille indéfinie, j'ai utilisé une structure obligatoire de représentation de sommet. Le pipeline ne va donc plus se demander quels champs auront quelle valeur.

Cela simplifie toutes les étapes du pipeline, car entre les étapes il se passe le pointeur vers les sommets.

E) Bonus : Multi-Threading

Suite à l'affichage des images en temps réel de mon pipeline, je me suis confronté a des soucies de performance. Pour résoudre ce problème j'ai dû implémenter du multithreading.

En effet j'ai vu avec valgrin que le pipeline faisait appel au pire des cas SIZE X de l'écran * SIZE Y de l'écran fois a Operation de Pixel qui est une résultante du remplissage. On peut donc en conclure que l'opération de remplissage est donc un bon départ pour une augmentation significative des performances.

Réflexion par rapport à l'année dernière : Trop de thread ne me fera pas gagner en performance, pas assez, et je n'obtiendrai pas le maximum de performance possible.

Je ne peux donc pas séparer 1 Facette pour 1 thread. Car on se retrouverait très rapidement avec 10000 thread. J'ai utilisé un système de N thread (configurable) qui font une rotation, cela veut dire qu'il y aura au plus N thread en même temps et que pour chaque facette 1 thread sera crée quand tous les thread sont en travail la prochaine facette attendra la fin d'un des thread.

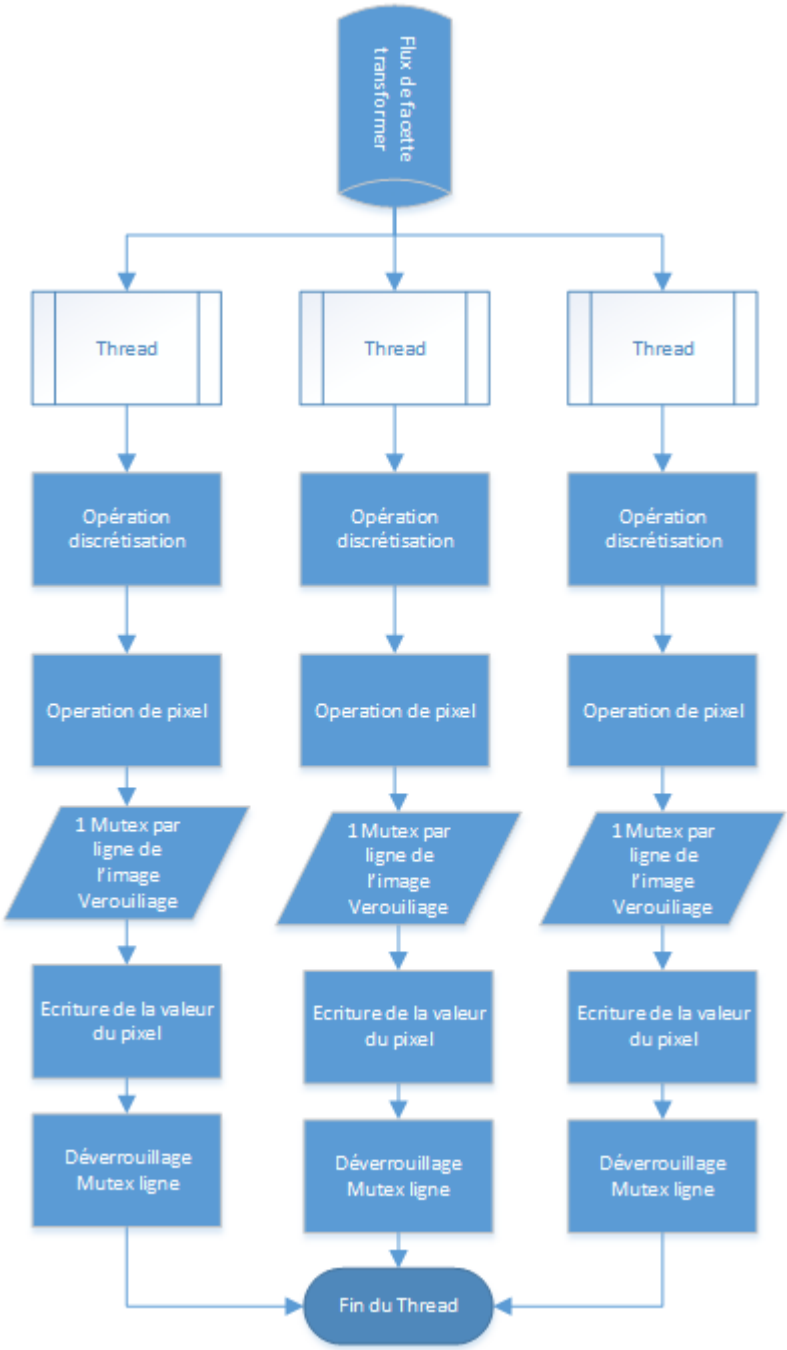
On peut aussi se poser la question des données en commun entre tous les thread qui seront modifiés.

La seule donner modifiée par les thread est l'image représentant l'écran. Pour pouvoir modifier les valeurs de cette image on doit se poser la même réflexion que pour les thread.

Combien de mutex doit-on créer ? 1 seul mutex ne serait sans doute pas réellement efficace, 1 mutex par pixel non plus, car couteux en mémoire.

C'est pour cela que j'ai décider de crée autant de mutex qui y a de ligne dans l'image ainsi on a un nombre raisonnable de mutex pour permettre au thread de peindre sur l'image.

Shema de l'implémentation du multithreading:

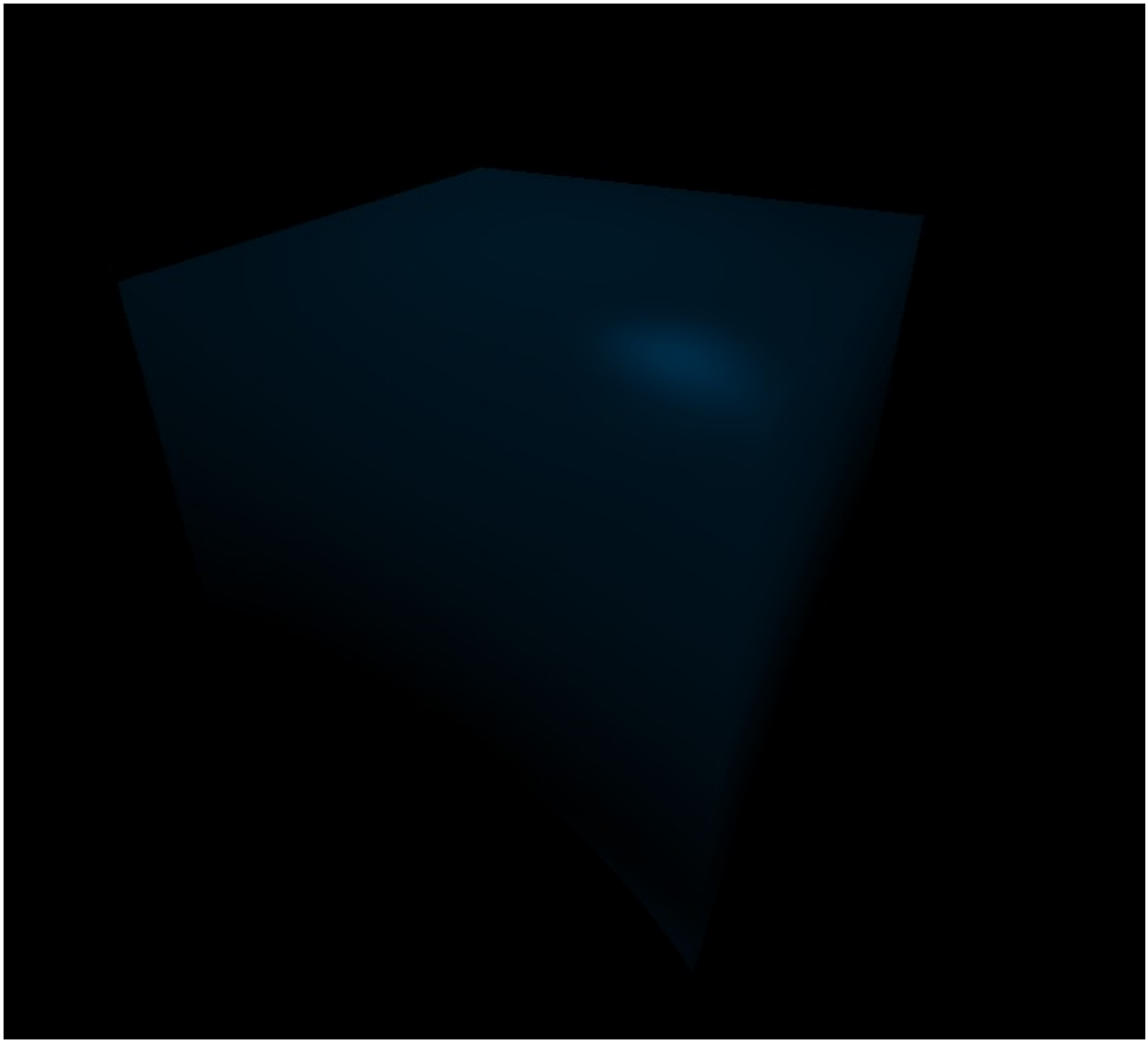


IV) Test

A) Cube

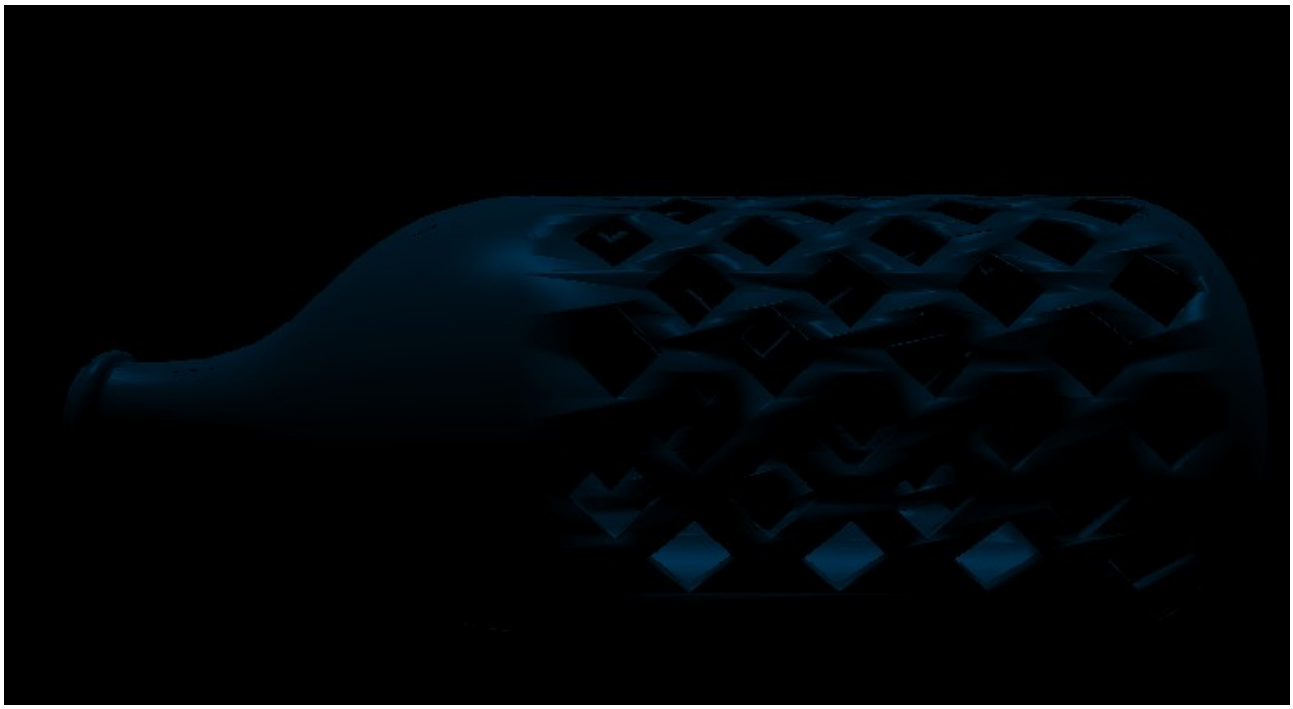
J'ai chargé un cube pour les tests :

Réglage du pipeline : Culling, clipping activer, shader phong, lampe (0,0, 2) p= 200.



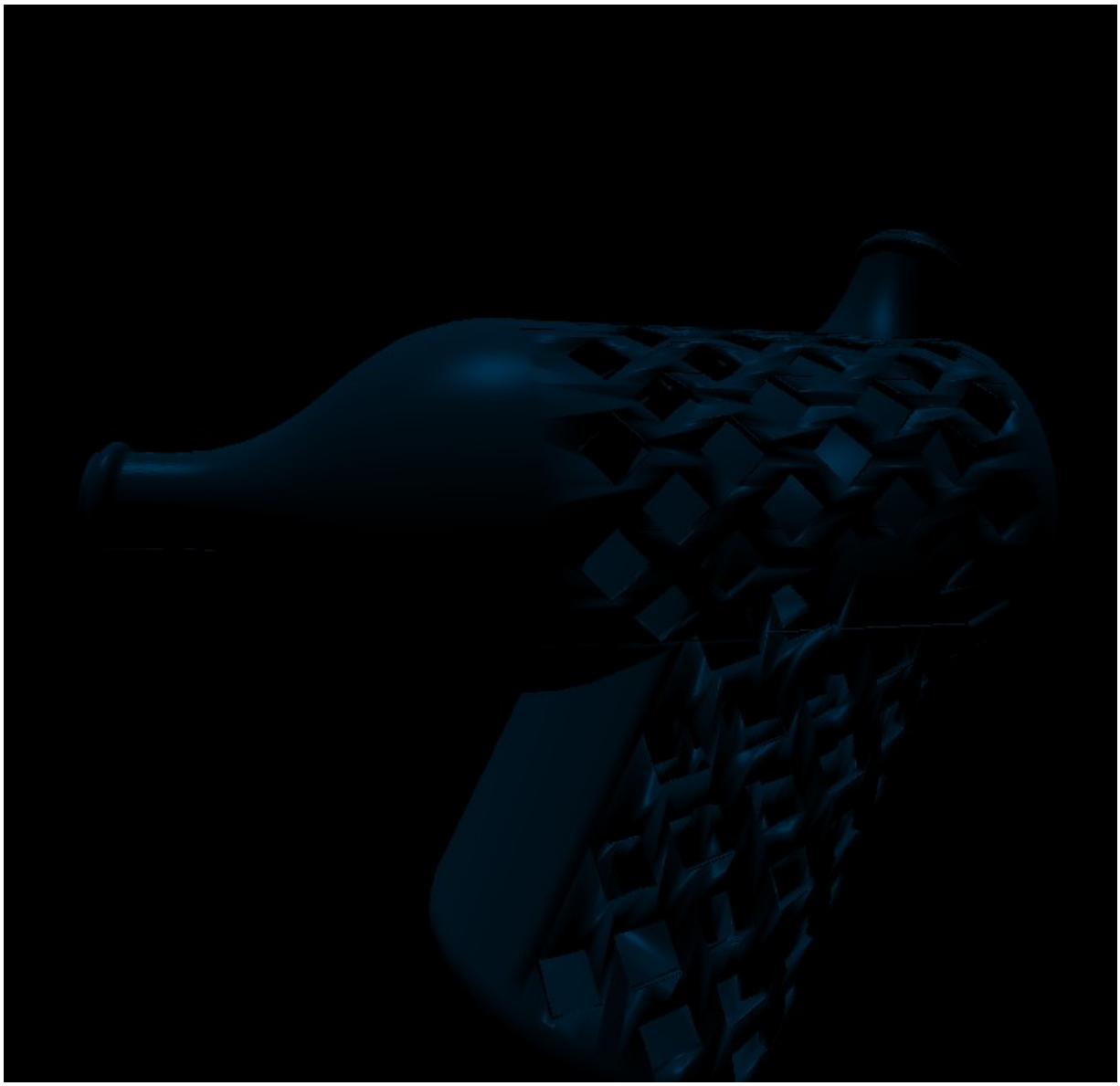
B) Object plus complexe

On va afficher le fichier bottle.ply qui est une bouteille avec la même configuration que le cube



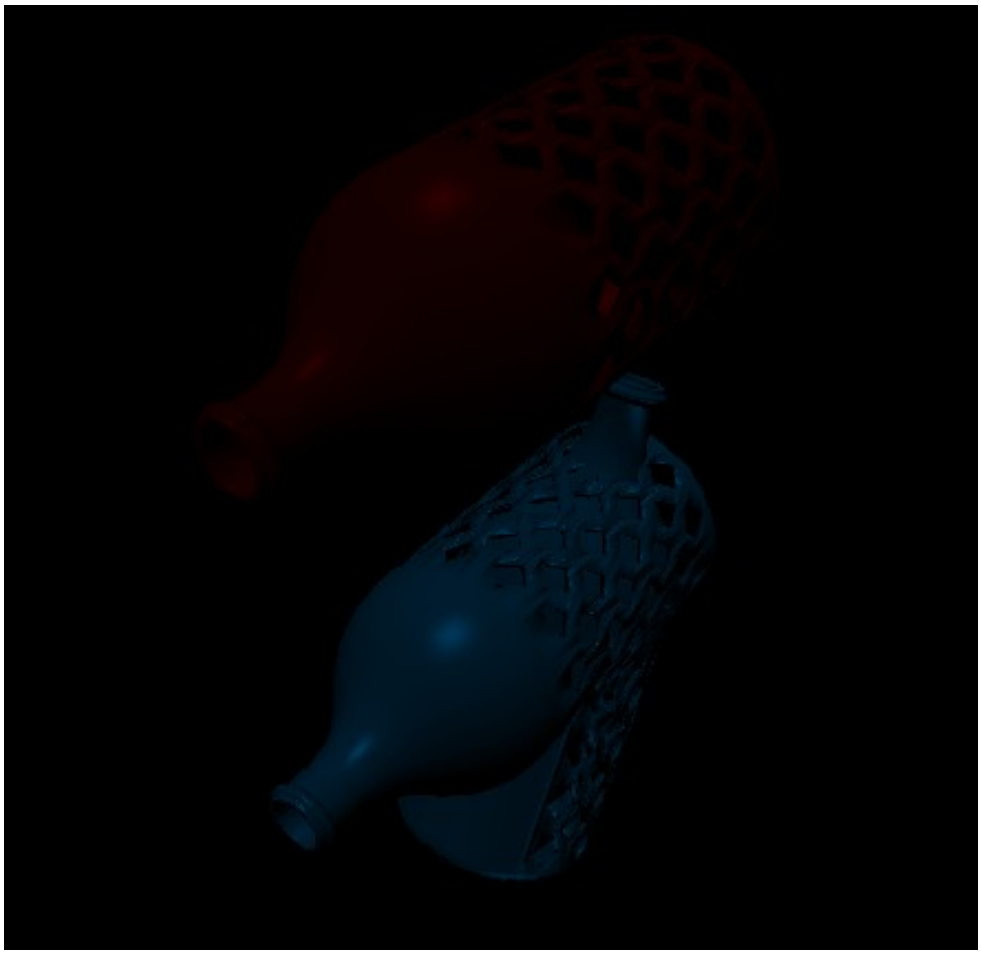
C) Multiple Object

On va reprendre la scène du dessus et ajouter une seconde bouteille en faisant une rotation et une translation en z de 0.2 et une rotation en y de 90 degrés :



Le rendu des deux objets se passe parfaitement et comme on peut le voir sur l'écran le zBuffer fonctionne bien.

Pour le plaisir de la couleur rouge, un troisième objet cette fois-ci en changeant la couleur :



V) Amelioration possible et Conclusion

Dans la liste des améliorations possibles, il y a l'ombrage à implémenter; corriger les problèmes de zflicking par moment, corriger les éventuels bugs pouvant survenir, implémenter la couleur de la lumière.

Pour conclusion on peut dire que c'était un projet passionnant qui requière beaucoup de rigueur et de travail personnel, mais que quand la majorité des fonctions sont implémentée correctement et que tout fonctionne, on est content d'avoir aussi bien travailler et d'avoir appris les bases même d'une carte graphique.