Rapport de Projet C++ : Moteur 3D

Binôme :
MARTIN Aslan
MORALES Tommy

ENSIIE - Année 2025-2026

Table des matières

1 Introduction

L'objectif de ce projet était de concevoir un moteur de rendu 3D "from scratch", en n'utilisant que le langage C++ standard et la librairie SDL2 pour l'affichage brut des pixels. Le défi principal consistait à réimplémenter toute la pipeline graphique (transformations, projections, rasterisation) que gèrent habituellement les cartes graphiques.

Nous avons structuré notre travail en deux phases :

1. La réalisation du moteur CPU (Rasterisation) imposé par le sujet, permettant d'afficher des objets 3D en temps réel.
2. Une extension technique vers le calcul GPU (Ray Tracing avec CUDA) pour obtenir un rendu physiquement réaliste (ombres portées) impossible à atteindre avec notre moteur CPU afin de pouvoir exploiter et dépasser la partie 'Threads' du cours.

2 Partie 1 : Architecture et Moteur CPU

2.1 Choix d'Architecture

Nous avons opté pour une approche modulaire stricte afin de séparer les données mathématiques de la logique d'affichage.

— **Généricité (Templates) :** Toutes nos classes géométriques (`Point3D`, `Shape`) sont templatisées. Cela nous a permis de développer en `float` pour la performance, tout en gardant la possibilité de passer en `double` ou `int` (pour la 2D) sans réécrire le code.

— **Gestion RAII :** La classe `Sdl` encapsule la gestion de la fenêtre. L'initialisation se fait dans le constructeur et le nettoyage (`SDL_DestroyWindow`) dans le destructeur. Cela nous a évité toute fuite de mémoire, même lors des crashs pendant le développement.

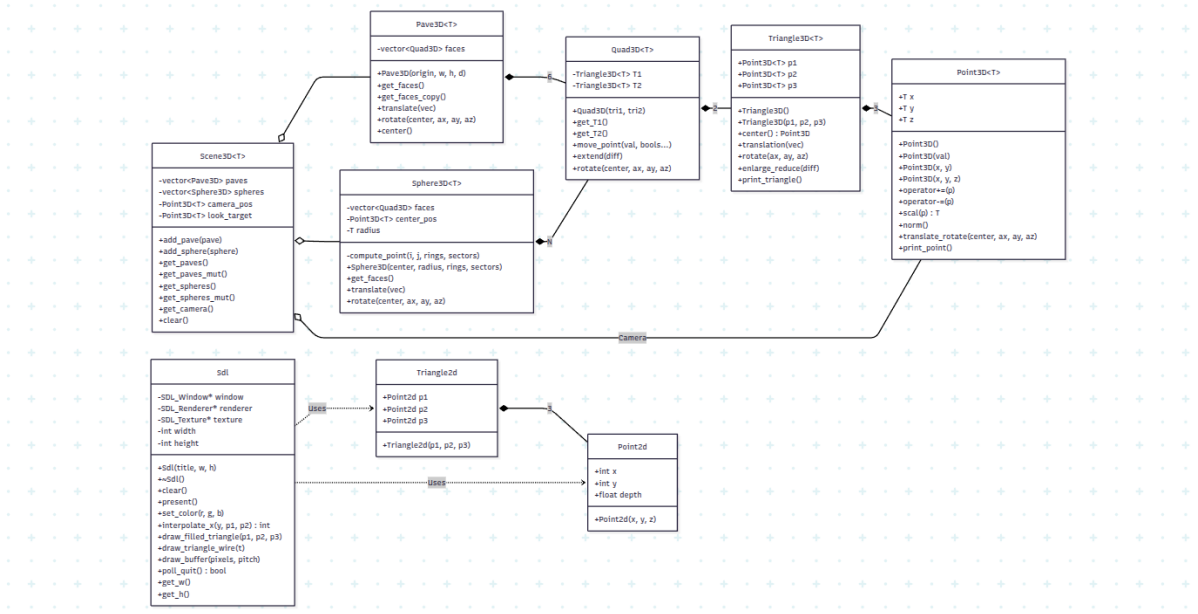


FIGURE 1 – Architecture globale du projet

2.2 Implémentation Mathématique et Difficultés

Le cœur du projet repose sur la manipulation de coordonnées 3D. C'est ici que nous avons rencontré nos premiers obstacles majeurs.

2.2.1 Le Problème des Rotations

Pour animer les objets, nous appliquons des matrices de rotation (Euler). Mathématiquement, la rotation autour de l'axe X est :

$$y' = y \cdot \cos(\theta) - z \cdot \sin(\theta)$$

$$z' = y \cdot \sin(\theta) + z \cdot \cos(\theta)$$

Le problème : Lors de notre première implémentation, le cube se déformait et "fondait" petit à petit. **La cause :** Nous modifions les coordonnées "en place". Nous calculions le nouveau y, puis nous utilisions ce *nouveau* y pour calculer le z, faussant le résultat. **La solution :** Nous avons dû utiliser des variables temporaires pour stocker l'état du point à l'instant t avant d'appliquer la transformation.

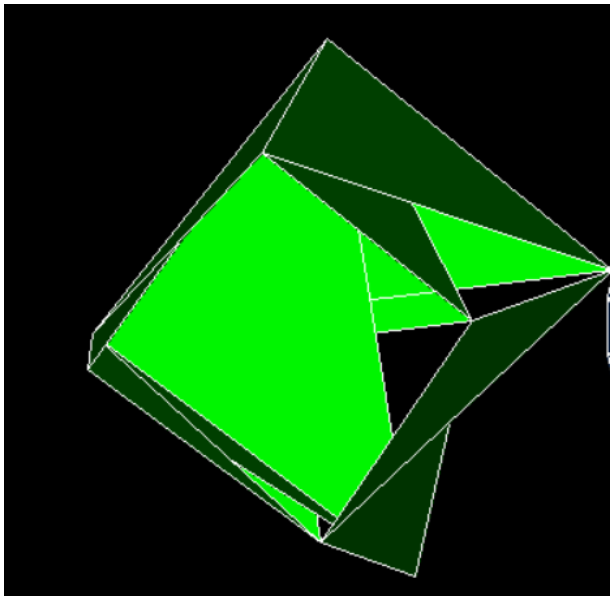


FIGURE 2 – Cube qui se déforme (bug)

2.2.2 Le "Tesseract" (Problème de Culling)

Une fois les rotations corrigées, nous avons affiché le cube plein. Le résultat était visuellement incompréhensible : on voyait l'intérieur du cube, donnant l'impression d'une figure impossible (Tesseract).

Analyse technique : Le "Back-face Culling" détermine si une face est visible via le produit scalaire entre la normale $\vec{N}$ et le vecteur vue $\vec{V}$. Notre condition était $\vec{N} \cdot \vec{V} \geq 0$. Or, selon la convention trigonométrique (sens des points), nos normales pointaient vers l'intérieur. **Correction :** En inversant la condition (≤ 0), le cube est devenu "solide".

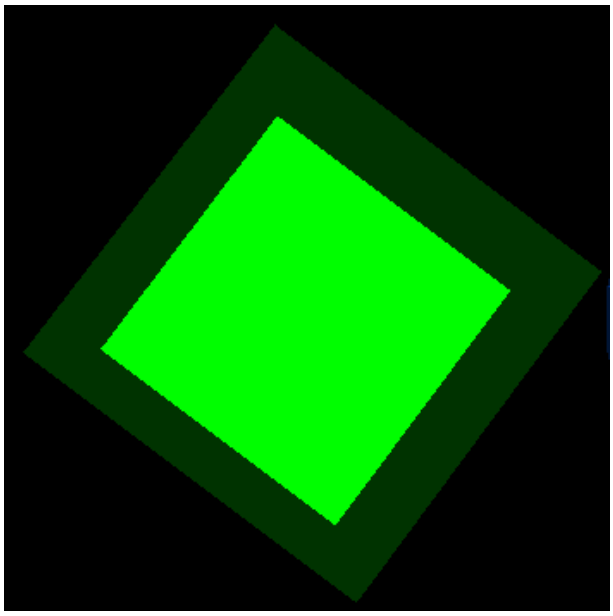


FIGURE 3 – Effet visuel du culling inversé (on voit l'intérieur).

2.3 Rasterisation (Scanline)

Comme la SDL ne trace pas de triangles pleins, nous avons codé l'algorithme du **Scanline**. La logique est la suivante :

1. **Tri :** On ordonne les 3 sommets $P1, P2, P3$ selon Y.
2. **Découpe :** On divise le triangle en deux parties (Haut et Bas) à la hauteur de $P2$.
3. **Balayage :** Pour chaque ligne Y, on calcule par interpolation linéaire (Thalès) les bornes X_{start} et X_{end}.
4. **Dessin :** On trace une ligne horizontale entre ces bornes.

Le Bug Linux ("Ghosting") : Sur la machine de développement (Linux hybride Intel/Nvidia), le buffer ne s'effaçait pas (`SDL_RenderClear` inopérant), créant une trainée fantôme. Nous avons contourné le problème en dessinant un rectangle noir géant à chaque frame.

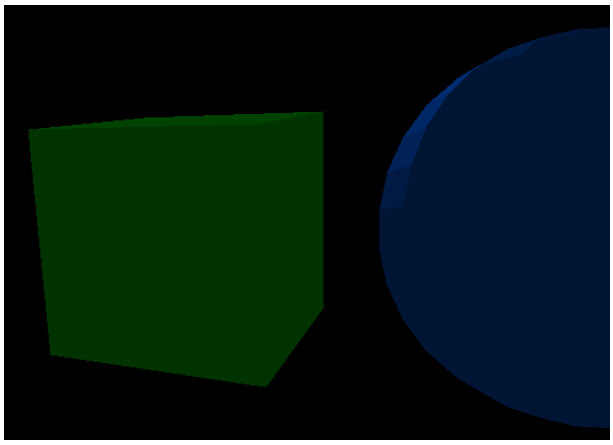
2.4 Bonus : Animation

Conformément à la demande du sujet ("rajouter une animation des objets"), nous n'avons pas réalisé une scène statique. Nous appliquons une rotation incrémentale à chaque frame sur tous les objets de la scène (le cube et la sphère).

Cela nous a permis de :

— Valider la robustesse de nos matrices de rotation.

— Vérifier que notre algorithme de *Back-face Culling* fonctionne bien dynamiquement (les faces apparaissent et disparaissent correctement quand l'objet tourne).



3 Partie 2 : Extension Ray Tracing (CUDA)

Après avoir validé le moteur CPU, nous avons voulu aller plus loin. La rasterisation gère mal les ombres portées. Nous avons donc choisi d'implémenter un Ray Tracer sur GPU via CUDA.

3.1 Changement de Paradigme

Contrairement au CPU où l'on projette les objets (Rasterisation), ici nous lançons un rayon pour chaque pixel (Ray Tracing).

— **CPU :** Boucle sur les objets → projeter → dessiner.

— **GPU :** Boucle sur les pixels (parallèle) → lancer rayon → chercher intersection.

3.2 Implémentation Technique

Nous avons dû gérer la mémoire manuellement (`cudaMalloc`, `cudaMemcpy`) pour envoyer notre scène (Sphères) sur la VRAM de la carte graphique.

3.2.1 Intersection Rayon-Sphère

Pour chaque rayon $R(t) = O+tD$, nous résolvons l'équation quadratique d'intersection avec une sphère :

$$at^2 + bt + c = 0$$

Si le discriminant $\Delta > 0$, le rayon touche la sphère. Nous gardons le plus petit $t > 0$ pour trouver l'objet le plus proche.

3.3 Gestion de la Lumière et des Ombres

C'est l'étape qui a demandé le plus d'itérations.

Étape 1 : Ombres inversées. Au début, nous avions défini une lumière ponctuelle mal positionnée par rapport à la caméra. Résultat : les objets étaient éclairés "de dos", et les ombres semblaient inversées.

Étape 2 : Ombres portées (Shadow Rays). Pour corriger cela, nous avons implémenté un système solaire :

— Une sphère centrale (Soleil) est la source de lumière.

— Pour chaque pixel, une fois qu'on touche un objet, on lance un **second rayon** vers le soleil.
— Si ce second rayon touche un objet en chemin, alors le pixel est dans l'ombre.

```
// Logique CUDA pour les ombres
Ray shadow_ray = { hit_point, direction_vers_soleil };
if (hit_any_object(shadow_ray)) {
    couleur = ambiante; // Ombre
} else {
    couleur = ambiante + diffuse; // Eclaire
}
```

FIGURE 4 – Système solaire final avec ombres portées

4 Conclusion

Ce projet a été l'occasion de confronter la théorie mathématique à la réalité du code. La partie CPU nous a appris la rigueur de la gestion mémoire et des transformations matricielles. L'extension CUDA nous a permis de comprendre comment les moteurs modernes obtiennent des images photoréalistes en exploitant le parallélisme.

Les difficultés rencontrées (bugs d'affichage Linux, problèmes géométriques) ont été frustrantes mais formatrices, nous obligeant à comprendre chaque étape de la pipeline graphique plutôt que de l'utiliser comme une boîte noire.

A Annexes

A.1 Guide de Compilation et d'Exécution

Comme notre projet contient deux moteurs distincts (le CPU obligatoire et le GPU bonus), nous avons séparé les fichiers principaux.

A.1.1 1. Compiler le Moteur CPU (Sujet Obligatoire)

C'est la partie qui répond strictement au cahier des charges (Rasterisation). Elle se compile avec la commande standard demandée :

```
g++ -g -Wall -Wextra -o prog main.cpp $(pkg-config --cflags --libs sdl2)
./prog
```

Commandes : L'animation est automatique. Fermez la fenêtre pour quitter.

A.1.2 2. Compiler le Moteur GPU (Bonus CUDA)

C'est l'extension Ray Tracing. Elle nécessite le compilateur NVIDIA (`nvcc`).

```
nvcc -o bonus main_bonus.cu -lSDL2
./bonus
```

A.2 Organisation des fichiers

Pour respecter la consigne "PAS de sous-répertoires", tous les fichiers sont à la racine :

- **main.cpp** : Point d'entrée du moteur Rasterisation (CPU).
- **main_bonus.cu** : Point d'entrée du moteur Ray Tracing (GPU).
- **shape.h** : Définition des classes mathématiques et géométriques (Point3D, Pave3D, Sphere3D...).
- **sdl_class.h** : Wrapper pour gérer la fenêtre SDL et le dessin (Lignes pour CPU, Texture pour GPU).