Rapport du Projet : Moteur 3D

TRAORE & YAMEOGO

2 janvier 2024

Professeur Encadrant : M. Vincent TORRI

Table des matières

1 Introduction

Le rendu 3D, pierre angulaire de nombreux domaines tels que les jeux vidéo, l'architecture, la simulation et la conception assistée par ordinateur, consiste à générer des images bidimensionnelles à partir de modèles tridimensionnels. Ce processus implique une transformation des données géométriques, une gestion de la perspective, de la profondeur et de la visibilité, ainsi qu'un rendu visuel des objets qui respecte les contraintes visuelles de notre vision. Parmi les techniques de rendu les plus utilisées, la rasterisation se distingue par sa capacité à offrir une performance élevée, la rendant incontournable pour des applications interactives en temps réel. Ce rapport présente une analyse approfondie du développement d'un moteur 3D basique, réalisé dans le cadre du projet *"Moteur 3D"* pour le cours de Programmation Avancée et Projet. Ce projet a été conçu comme une exploration pratique des mécanismes fondamentaux du rendu graphique 3D, en utilisant la bibliothèque SDL2. Notre objectif a consisté à élaborer une application capable de manipuler et d'afficher des objets 3D, tout en intégrant des mécanismes de contrôle interactif. Bien que modeste dans sa portée, ce moteur 3D nous a permis d'étudier en profondeur les défis associés au rendu par rasterisation, en nous confrontant aux problématiques de la programmation graphique. Ce rapport documente nos choix de conception, les difficultés techniques rencontrées, les solutions méthodologiques mises en œuvre, ainsi que les perspectives d'amélioration que nous envisageons.

2 Choix de Conception

Dans cette section, nous allons détailler de manière approfondie les choix de conception qui ont guidé le développement de notre moteur 3D, en expliquant les raisons qui ont motivé chaque décision.

2.1 Bibliothèque Graphique : SDL2

Nous avons opté pour la bibliothèque SDL2. SDL2 offre un bon compromis entre la gestion de bas niveau des ressources graphiques et les fonctions nécessaires pour la création de fenêtres et le rendu 2D. Son utilisation nous a permis de nous concentrer sur les aspects fondamentaux du rendu 3D sans nous perdre dans les détails spécifiques des API d'affichage du système d'exploitation. De plus, étant donné que ce cours était basé sur l'utilisation de SDL2, c'était un choix pertinent pour notre projet.

2.2 Modélisation des Objets 3D

Pour la représentation des objets 3D, nous avons choisi une approche par facettes, en décomposant les objets en surfaces planes (les polygones) :

- **Point3D** : Représente un point dans l'espace 3D, défini par trois coordonnées flottantes x, y, et z. Le choix du type *float* assure une plus grande précision et évite des erreurs de calcul lors des transformations géométriques. Nous avons encapsulé ces trois coordonnées dans une classe nommée `Point3D` pour plus de clarté.
- **Triangle3D** : Représente un triangle dans l'espace 3D, défini par trois objets *Point3D* qui représentent ses sommets. Cette classe permet la manipulation des triangles et facilite la construction d'objets 3D.
- **Quad3D** : Représente un quadrilatère 3D, défini par quatre objets *Point3D*. Cette classe a été créée dans le but de faciliter la construction d'objets comme le cube, qui est naturellement constitué de quadrilatères. Cette classe est celle que nous allons utiliser par la suite pour tous nos rendus.

— **Scene3D** : Permet de gérer l'ensemble de la scène 3D, en contenant une collection d'objets *Quad3D*. Cette approche permet de structurer le code et d'avoir une vue d'ensemble des objets qui composent la scène.

Ce choix nous a permis de nous familiariser avec la technique du rendu par rasterisation, utilisée dans la plupart des cartes graphiques modernes. De plus, nous avons choisi d'utiliser la technique du "triangle fan" pour le rendu et le remplissage des quadrilatères. Cela nous a permis de simplifier le rendu en décomposant un quadrilatère en deux triangles lors de l'affichage, sans que cela ne nécessite d'ajouter des méthodes supplémentaires.

2.3 Classes de Base pour le Rendu et la Projection

En plus des classes dédiées à la modélisation 3D, notre projet inclut les classes suivantes, essentielles pour le rendu :

— **Point2D** : Représente un point dans l'espace 2D de l'écran. Elle contient les coordonnées entières x et y nécessaires pour les fonctions de dessin de SDL. Les coordonnées de la classe *Point3D* sont converties en coordonnées *Point2D* grâce à la projection, et ces coordonnées sont utilisées lors du rendu des polygones.

— **Sdl** : Cette classe encapsule toutes les fonctionnalités de SDL, permettant une gestion plus structurée de l'initialisation, de la création de la fenêtre et du moteur de rendu, ainsi que leur libération. Cela a amélioré notre organisation du code et a garanti une libération correcte des ressources SDL. Elle fournit des méthodes pour la gestion de la fenêtre, de la surface d'affichage, et pour l'utilisation du moteur de rendu. Cette approche permet de ne pas polluer le main avec la gestion des ressources SDL et permet une approche beaucoup plus propre.

Le diagramme UML qui résume l'architecture de notre projet est présenté *figure 1.* Ce diagramme a été réalisé avec draw.io.

2.4 Fonctions de Transformation et de Rendu

Plusieurs fonctions clés ont été développées pour gérer les transformations, le rendu et la manipulation des objets 3D :

— **rotatePoint** : Cette fonction prend un point 3D en entrée et applique une transformation de rotation autour des axes X et Y. Elle utilise les matrices de rotation pour transformer la position du point de manière correcte, et est la base des rotations appliquées aux objets de la scène. Cette fonction utilise un système de coordonnées cartésiennes.

— **project** : Cette fonction effectue la projection d'un point 3D sur le plan 2D de l'écran, en utilisant un facteur d'échelle basé sur la profondeur du point. Cela permet de donner l'illusion de profondeur lors de la représentation des objets 3D sur un écran 2D.

— **averageDepth** : Cette fonction calcule la profondeur moyenne d'un quadrilatère 3D, en faisant la moyenne des coordonnées z de ses sommets. Elle est utilisée lors du tri des polygones pour la gestion de la profondeur. La profondeur d'un polygone est une indication de sa distance par rapport à la caméra.

— **drawTriangle** : Cette fonction permet de dessiner un triangle à l'écran. Elle n'est pas utilisée dans le rendu final, mais elle est essentielle à des fins de tests et d'expérimentation. Elle est basée sur l'utilisation de tracés de lignes (`SDL_RenderDrawLine`) et est donc la plus simple possible.

— **drawQuad** : Cette fonction, essentielle à notre projet, est responsable du rendu des quadrilatères (faces des objets) sur l'écran. Elle utilise la fonction `SDL_RenderGeometryRaw` pour un remplissage efficace des polygones et ajoute des contours avec `SDL_RenderDrawLine` pour une meilleure visibilité des objets. Elle applique aussi les transformations des objets

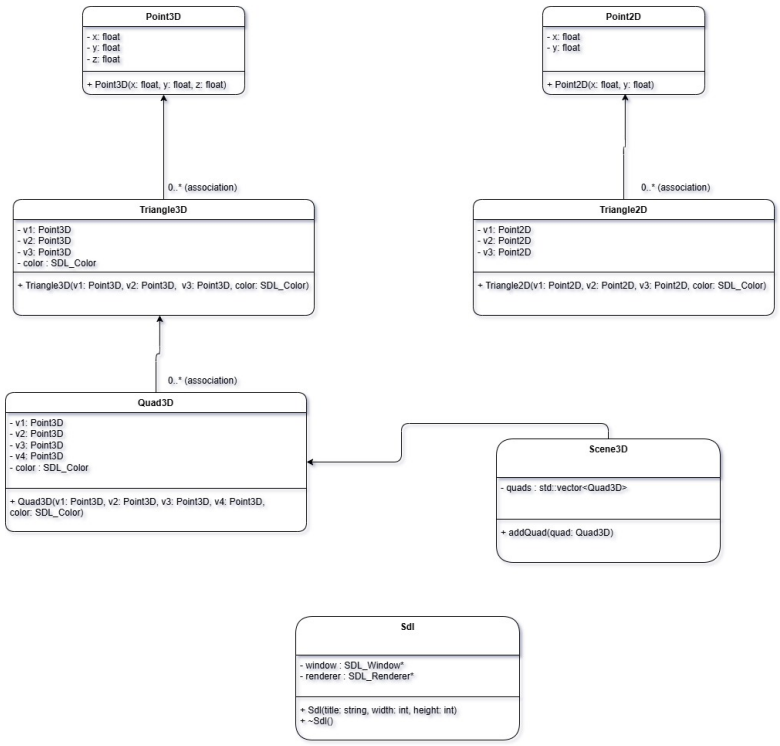


FIGURE 1 – Diagramme UML du projet

de la scène.

2.5 Gestion des Transformations

Pour les transformations des objets 3D, nous avons appliqué les rotations avant les translations :

1. Les points qui définissent les sommets de chaque objet sont d'abord tournés autour des axes X, puis Y, grâce à la fonction `rotatePoint()`. Cette fonction nous permet de faire pivoter les objets de la scène de façon indépendante et correcte.
2. Une fois la rotation effectuée, les points ainsi tournés subissent une translation dans la fonction `drawQuad()`, grâce à des variables globales. Cette translation permet de déplacer l'ensemble de la scène et de simuler des mouvements plus généraux.

Les transformations sont calculées à chaque rendu pour permettre une animation en temps réel.

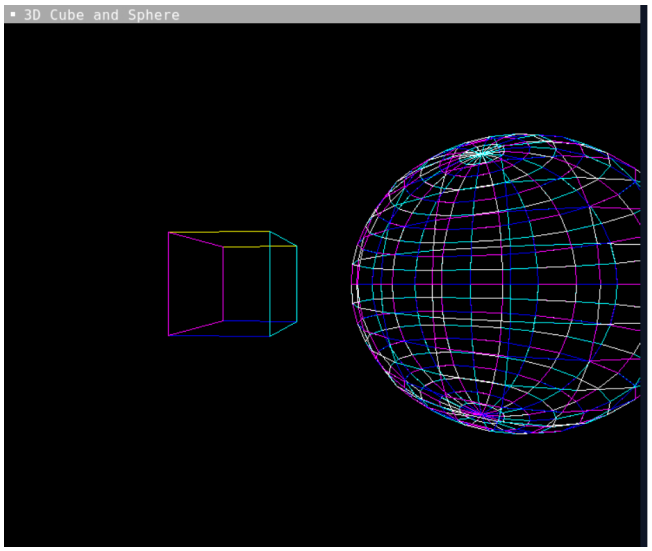
3 Difficultés Rencontrées et Solutions

Ce projet de développement d'un moteur 3D a soulevé de nombreux défis techniques. Dans cette section, nous décrivons les principales difficultés que nous avons rencontrées et les solutions que nous avons mises en œuvre pour y remédier, en justifiant nos choix techniques.

3.1 Défis Initiaux du Rendu : Le Remplissage des Faces

L'Apparition Inattendue de Structures Filaires

Notre premier défi a été lié au rendu des objets 3D. Au début de notre travail, seuls les contours des faces étaient visibles, ce qui donnait l'impression d'une structure filaire et non d'un objet plein. Le remplissage des faces est essentiel pour avoir un rendu réaliste et fidèle aux objets que nous souhaitions modéliser.



Exploration des Solutions et Choix Techniques

Nous nous sommes d'abord tournés vers les fonctions classiques de SDL pour le remplissage de polygones, notamment `SDL_RenderFillPolygon`. Cependant, nous nous sommes rapidement rendus compte que cette fonction n'était pas disponible ou non-fonctionnelle dans l'environnement de compilation que nous utilisions.

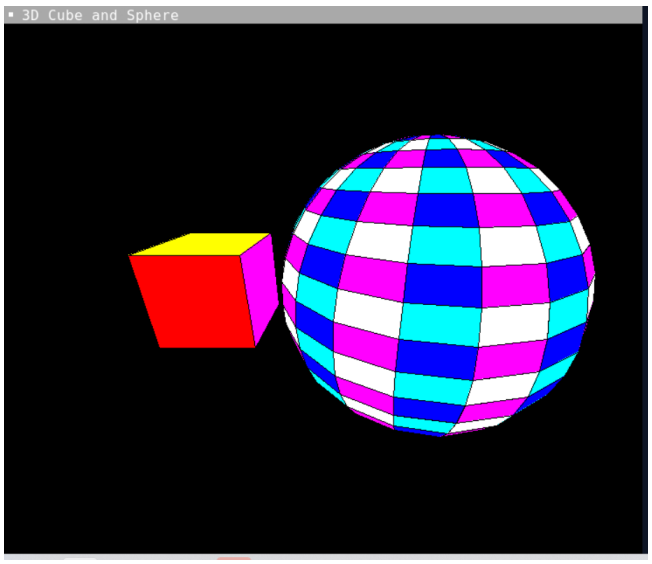
La Solution Retenue : `SDL_RenderGeometryRaw`

Après des recherches approfondies, nous avons découvert que la fonction `SDL_RenderGeometryRaw` offrait une alternative plus appropriée pour le remplissage de polygones. Cette fonction, bien que plus complexe à utiliser, nous a permis de dessiner directement des polygones remplis. Son utilisation a nécessité une compréhension précise de la manière dont elle attend les données (notamment les sommets, les couleurs et les indices). L'utilisation de cette fonction a nécessité un certain temps d'adaptation et une bonne maitrise de ses paramètres.

Implications Techniques

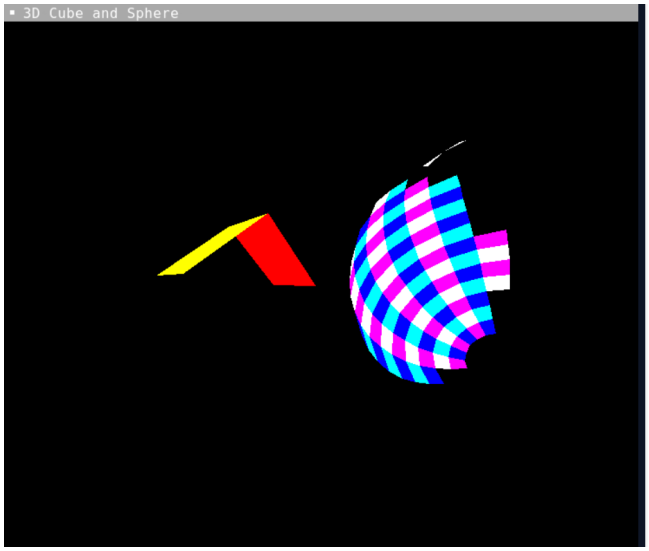
L'utilisation de `SDL_RenderGeometryRaw` a eu plusieurs implications :

- Nous avons dû apprendre à manipuler des tableaux de données séparés pour les coordonnées des sommets et les couleurs, car cette fonction ne reçoit pas directement un tableau de structures `SDL_Vertex`. Cela a nécessité un effort supplémentaire d'abstraction, car les coordonnées et les couleurs ne sont pas directement liées à un point en particulier.
- Nous avons aussi décidé de manipuler des tableaux de données plus élémentaires (des tableaux de `float` pour les coordonnées, des tableaux de `SDL_Color` pour les couleurs, et un tableau d'entiers pour les indices), ce qui s'est avéré plus simple d'utilisation et plus performant.



3.2 Problèmes de Profondeur et Disparition des Faces

L'Apparition de Superpositions et de Disparitions Inattendues
Nous avons rencontré des difficultés liées à la gestion de la profondeur des objets et à la visibilité des faces lors de la rotation. Plus précisément, les faces des objets les plus éloignés étaient parfois affichées par-dessus les faces plus proches, ce qui donnait un rendu incorrect de la profondeur. De plus, nous avons constaté que certaines faces disparaissaient lors des rotations des objets, causant un effet visuel perturbant. Cela suggérait des lacunes au niveau de la gestion de la visibilité des faces.



Nous avons d'abord cherché à comprendre comment gérer la visibilité des faces à l'aide d'un algorithme de "backface culling" que nous avons implémenté, qui consiste à ne pas afficher les faces qui étaient tournées vers l'arrière de la caméra, mais cette approche ne s'est pas avérée assez robuste et était la cause des disparitions. Nous avons aussi testé plusieurs approches du tri des polygones basées sur la profondeur, en tentant des implémentations différentes des algorithmes de tri, mais nous avons fini par utiliser `std::sort` qui était la plus performante.

La Solution Adoptée : Tri par Profondeur et Désactivation du Backface Culling

— **Implémentation du Tri par Profondeur** : Afin de gérer correctement l'ordre d'affichage, nous avons implémenté une méthode de tri par profondeur, dans laquelle les faces les plus éloignées de la caméra sont affichées avant les faces les plus proches. Cela a été

rendu possible en utilisant une fonction `averageDepth()` qui calcule la moyenne de la coordonnée z des sommets d'un quadrilatère, ce qui sert de critère pour trier les faces avec la fonction `std::sort`. L'utilisation de l'algorithme *std : :sort* de la bibliothèque standard a été le plus efficace et nous a permis de nous concentrer sur la logique du rendu plus que sur l'algorithme de tri lui même.

— **Désactivation du Backface Culling** : La méthode de "backface culling" que nous avions tentée d'implémenter (qui consiste à ne pas afficher les faces arrières) n'a pas fonctionné comme attendu et était à l'origine des disparitions de faces. Nous avons donc temporairement désactivé ce backface culling afin de stabiliser l'affichage et garantir que les objets soient visibles sous tous les angles. Bien que cette solution ne soit pas idéale (elle peut avoir un léger impact sur la performance et la visibilité), cela nous a permis de résoudre le problème de la disparition des faces pour l'instant.

Limitations et Pistes d'Amélioration

Bien que ces solutions nous aient permis de corriger les problèmes de base, nous sommes conscients de leurs limites : la performance n'est pas optimale car nous trions toutes les faces à chaque rendu et nous affichons toutes les faces (au lieu de cacher les faces arrières), et la gestion de la profondeur reste une approche simplifiée. Il serait intéressant d'implémenter un vrai algorithme de Z-Buffer dans le futur pour améliorer l'affichage des objets.

3.3 Organisation du Code et Gestion des Ressources SDL

Difficulté Rencontrée : Problèmes d'Organisation et Gestion Inefficace des Ressources

Au début de notre projet, l'initialisation et la gestion des ressources de la bibliothèque SDL (fenêtre, moteur de rendu) étaient dispersées directement dans la fonction `main()`. Cette approche présentait plusieurs inconvénients, tels qu'un code difficile à lire, à modifier et à maintenir, ainsi qu'un risque de fuites de mémoire.

Solution Adoptée : Création de la Classe `Sdl`

Pour remédier à ces problèmes d'organisation, nous avons créé une classe nommée `Sdl` dédiée à la gestion des ressources de SDL. Cette classe encapsule l'initialisation de SDL (avec `SDL_Init()`), la création de la fenêtre (avec `SDL_CreateWindow()`) et du moteur de rendu (avec `SDL_CreateRenderer()`), ainsi que la libération des ressources (avec `SDL_DestroyRenderer()`, `SDL_DestroyWindow()`, et `SDL_Quit()`).

Avantages de l'Encapsulation

— L'utilisation d'un constructeur permet de faire l'initialisation de SDL et la création de la fenêtre, du moteur de rendu en une seule ligne de code. On utilise une approche de *RAII* (Resource Acquisition Is Initialization) dans le constructeur.

— L'utilisation d'un destructeur permet de libérer les ressources SDL de manière automatique lors de la fin de l'exécution du programme. Cela a permis d'éviter les fuites de mémoire et de garantir un bon comportement du programme.

— L'encapsulation a permis d'avoir un code plus propre, plus modulaire, et a facilité sa maintenance et sa lecture.

4 Conclusion

Ce projet de développement d'un moteur 3D basique nous a apporté de précieuses connaissances et expériences dans le domaine de la programmation.

4.1 Synthèse des Réalisations

Nous avons réussi à concevoir une application qui :

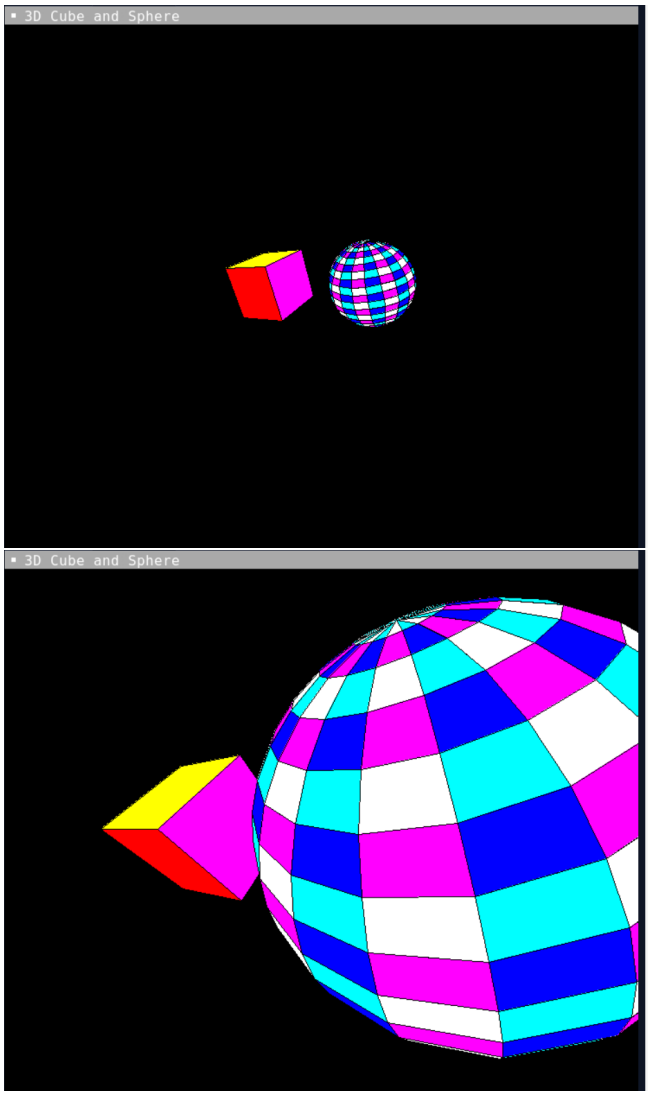
— Affiche des objets 3D (cube et sphère) sur un écran 2D.
— Applique des transformations de rotation et de translation de manière interactive et en temps réel.
— Utilise des couleurs distinctes pour chaque face des objets.
— Gère un tri par profondeur afin d'afficher les faces dans le bon ordre.

4.2 Description de l'Interface et de son Utilisation

L'application que nous avons développée est interactive et permet de manipuler la scène 3D. Voici comment vous pouvez interagir avec notre moteur 3D **après avoir cliqué sur la fenêtre** :

— **Rotations** : Les rotations autour des axes X et Y des objets sont contrôlées à l'aide des touches directionnelles du clavier (flèches).
— **Translations** : La translation (déplacement de toute la scène) est contrôlée à l'aide des touches *W, A, S, D, Q et E* du clavier.

Il faut souligner que ces déplacements permettent d'interagir avec la scène en entier. L'affichage de l'écran se fait dans une fenêtre SDL, et l'application se termine en fermant la fenêtre.



4.3 Difficultés et Enseignements

Ce projet nous a permis de mieux maîtriser le développement d'applications graphiques, et de développer nos compétences en programmation C++. Nous avons compris la complexité

de certaines fonctions comme `SDL_RenderGeometryRaw` qui sont très performantes mais qui nécessitent une bonne compréhension des mécanismes qu'elles utilisent. Les difficultés que nous avons rencontrées nous ont aidé à développer notre capacité de résolution de problèmes et à mieux maitriser les concepts fondamentaux du rendu 3D par rasterisation.

4.4 Perspectives d'Avenir

Pour la suite, nous envisageons plusieurs améliorations afin de perfectionner notre moteur 3D :

- **Implémentation de l'éclairage** : L'intégration de sources de lumière et de calculs d'éclairage pour rendre les objets plus réalistes.
- **Textures** : L'application de textures sur les objets afin d'améliorer leur aspect visuel.
- **Amélioration du Backface Culling** : Mettre en place une méthode robuste de Backface Culling afin d'éviter de dessiner les faces arrières et d'améliorer la performance du rendu.
- **Implémentation d'un Z-Buffer** : Utiliser un Z-Buffer pour une gestion plus fine de la profondeur et améliorer la performance et le rendu des objets qui se chevauchent.
- **Ajout de formes géométriques** : Développer notre bibliothèque d'objets en ajoutant d'autres formes telles que des tores, des cônes, des cylindres, etc.
- **Optimisation de Performance** : Étudier les algorithmes de rendu et les optimiser pour améliorer la vitesse du moteur.