




# CHPS0802 -PROGRAMMATION GPU

*Compte rendu*

*Projet Ray Tracing C++ / CUDA*

Wejdane BOUCHHIOUA

M1 CHPS

2024 -2025

# Introduction :

Ce projet a été réalisé dans le cadre d'un module de **programmation GPU avec CUDA** et porte sur le développement d'un moteur de rendu 3D par **Ray Tracing**. L'objectif principal est de construire un pipeline de rendu fonctionnel en **C++ séquentiel**, puis d'en tirer parti en le migrant vers une **implémentation GPU hautement parallèle**.

La finalité est de comprendre les **bases du Ray Tracing**, puis apprendre à les adapter à un environnement massivement parallèle, en surmontant les limitations techniques imposées par CUDA.

# Le ray Tracing :

Le **Ray Tracing** (ou lancer de rayons) est une méthode de rendu d'images tridimensionnelles qui simule le trajet de la lumière de manière réaliste. Pour chaque **pixel** d'une image, un **rayon est émis** depuis une caméra virtuelle dans la scène, à la recherche d'intersections avec les objets 3D (sphères, plans, triangles...).

En cas d'intersection, on calcule la couleur finale du pixel en prenant en compte :

- la **position du point d'impact**,
- la **normale à la surface**,
- les **propriétés du matériau** (diffusion, brillance, réflexion),
- et l'**influence des sources de lumière**.

La fonction de rendu `ray_color()` peut effectuer plusieurs rebonds (réflexions/réfraction) jusqu'à une profondeur maximale, ce qui rend l'algorithme récursif et computationnellement intensif.

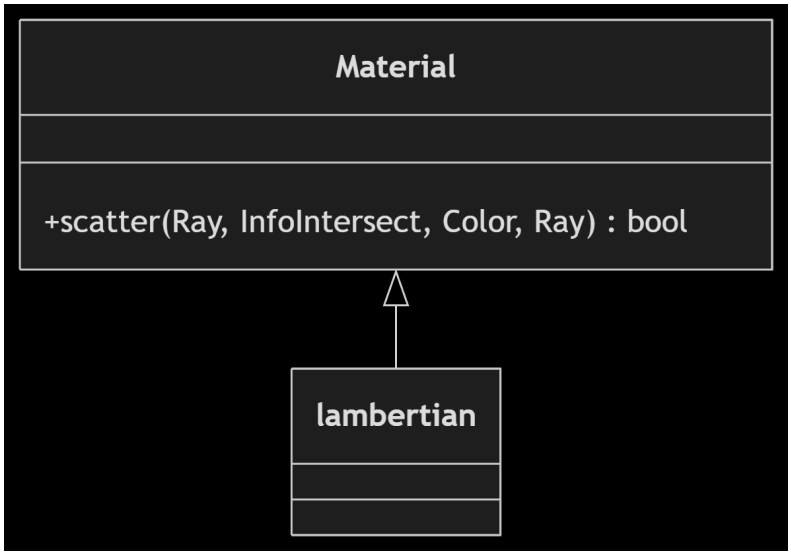
# Partie Séquentielle :

L'implémentation séquentielle repose sur un **design orienté objet**, avec des classes bien distinctes pour la les objets, la caméra, les matériaux, la lumière, la scène, et le moteur de rendu. L'objectif était de garder une architecture claire, maintenable, et facilement testable.

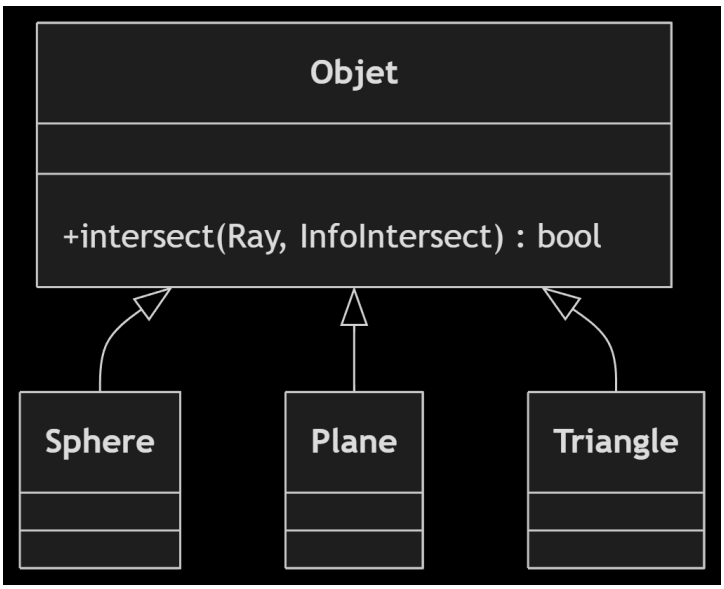
## Les différentes classes et leur rôle

- `Ray` : représente un rayon (origine + direction).

- **`Point3D`** / **`Vecteur3D`** : encapsulent les calculs géométriques.
- **`Color`** : hérite de `Vecteur3D` et représente une couleur RGB.
- **`Material`** : contient les propriétés optiques




- **`Objet`** *(classe abstraite)* : interface commune avec une méthode virtuelle `intersect()`.
    - **Sphères, Triangles, Plans** héritent de `Objet` et implémentent leur logique d'intersection.



- **`Scene`** : contient une liste d'objets (`Objet*`) et de lumières. Gère les tests d'intersections et l'illumination.

- **`Camera`** : génère des rayons à partir des coordonnées d'écran.

## Implémentation C++ et ses spécificités:

L'implémentation du projet de Ray Tracing exploite plusieurs fonctionnalités avancées du langage C++, ce qui permet de concevoir une architecture flexible, modulaire et performante. Parmi ces fonctionnalités, on retrouve le polymorphisme, qui joue un rôle central dans la conception orientée objet du système. Ce mécanisme permet notamment l'utilisation de méthodes virtuelles, comme dans le cas de la méthode `intersect`, qui est déclarée comme virtuelle pure dans la classe abstraite `Objet`. Cela permet à chaque type d'objet dérivé (comme une sphère ou un triangle) de définir sa propre logique d'intersection avec un rayon. Par exemple :

```
class Objet {
public:
    virtual bool intersect(const Ray& r, InfoIntersect& rec) const = 0;
};
```

La gestion de la mémoire est également pensée de manière efficace grâce à l'utilisation de pointeurs intelligents, notamment `std::shared_ptr`, qui facilitent le partage de matériaux entre plusieurs objets sans risques de fuites de mémoire. Par ailleurs, les objets de la scène sont alloués dynamiquement, ce qui permet une plus grande flexibilité dans la construction et la manipulation de la scène 3D à tracer.

J'ai également implémenté la récursivité, notamment dans la fonction `ray_color`, qui se rappelle elle-même à chaque rebond du rayon sur un objet. Cette récursivité permet de simuler les réflexions et les diffusions successives de la lumière dans la scène. Pour éviter des appels infinis et contrôler la performance, une profondeur maximale est fixée à 50 rebonds.

Enfin, pour améliorer la qualité de l'image générée et réduire les effets d'aliasing, une technique d'échantillonnage stochastique est utilisée. Cela signifie que pour chaque pixel, 100 rayons sont lancés avec de légères variations aléatoires dans leur direction. Les couleurs résultantes sont ensuite produites en moyennes, ce qui permet de produire une image plus lisse, sans artéfacts visibles, en simulant un flou réaliste au niveau des bords.

# Partie Parallèle avec CUDA (GPU) :

Le Ray Tracing est naturellement parallélisable, car chaque pixel peut être calculé indépendamment. Cela en fait un candidat idéal pour une implémentation GPU puisque des milliers de threads peuvent être lancés simultanément, chacun traitant un rayon. En addition, le temps de rendu peut chuter drastiquement si le calcul est bien distribué. En plus, CUDA permet un contrôle précis sur la gestion mémoire et la structure des données.

## Les défis rencontrés lors de la migration de la version séquentiel vers la version parallèle CUDA et solutions:

### a) Polymorphisme et Fonctions Virtuelles:

En effet, les GPU NVIDIA ne prennent pas en charge les fonctionnalités avancées de la programmation orientée objet (OOP), comme les méthodes virtuelles et l'héritage. Pour contourner cette limitation, une solution consiste à remplacer ces mécanismes par une structure plus simple basée sur des **énumérations (`enum`)** et des **unions (`union`)**. Chaque objet de la scène est identifié par un type (par exemple `SPHERE` ou `TRIANGLE`), et son comportement est géré manuellement via une structure `Object_GPU` qui contient une union des différents types d'objets et un identifiant de matériau.

```
typedef enum { SPHERE, TRIANGLE } ObjectType;

typedef struct {
    ObjectType type;
    union {
        Sphere_gpu sphere;
        Triangle_gpu triangle;
    };
    int material_id;
} Object_GPU;
```

L'intersection entre un rayon et un objet est ensuite traitée via une **fonction de dispatch manuelle**, qui effectue un `switch` sur le type de l'objet pour appeler la fonction d'intersection appropriée.

```
__device__ bool intersect(const Object_GPU* obj, const Ray& r,
InfoIntersect* rec) {
        switch(obj->type) {
               case SPHERE: return sphere_intersect(&obj->sphere, r,
rec);
```

```
                case TRIANGLE: return triangle_intersect(&obj->triangle,
r, rec);
            }
        }
```

b) Récursivité :

Les appels récursifs sont inefficaces, voire inapplicables sur GPU en raison des contraintes de mémoire et de performance. Pour résoudre ce problème, la récursivité est remplacée par une **version itérative** de la fonction `ray_color`, qui effectue une boucle jusqu'à une profondeur maximale prédéfinie (`MAX_DEPTH`). À chaque itération, les calculs d'intersection et les interactions avec les matériaux sont effectués, et un nouveau rayon est généré. Cette approche permet de simuler les réflexions successives tout en restant compatible avec l'exécution massive et parallèle des threads GPU.

```
        __device__ Color ray_color(Ray r, SceneData scene, curandState*
rs) {
            Color attenuation(1.0f);
            for (int depth = 0; depth < MAX_DEPTH; depth++) {
                // Calcul des intersections
                // Application des matériaux
                r = scattered; // Nouveau rayon
                attenuation *= mat_attenuation;
            }
            return attenuation * background;
        }
```

c) Gestion de mémoire:

Pour la gestion de la mémoire, le transfert des données de la scène depuis le CPU vers le GPU est une étape cruciale. Pour cela, une allocation mémoire sur le device est effectuée à l'aide de `cudaMalloc`, suivie d'une copie des données avec `cudaMemcpy`. Les objets de la scène sont regroupés dans une structure légère `SceneData`, qui contient des pointeurs vers les tableaux d'objets et de matériaux, ainsi que leur nombre total. Cette organisation permet de manipuler facilement l'ensemble de la scène au sein des kernels CUDA.

d) Les nombres aléatoires :

La fonction `rand()` utilisée en CPU n'est pas thread-safe et ne peut pas être utilisée efficacement sur GPU. La solution consiste à utiliser la **bibliothèque CURAND**, qui

permet d'initialiser un état aléatoire par thread grâce à la fonction `curand_init`. Une fois l'état initialisé, chaque thread peut générer des nombres aléatoires de manière indépendante et sécurisée via `curand_uniform`. Cette capacité est essentielle pour des opérations comme l'anti-aliasing ou la génération de rayons diffus.

### Stratégies d'implémentations partie CUDA:

La stratégie d'implémentation du Ray Tracing sur GPU repose sur une organisation efficace des threads, une gestion optimisée de la mémoire et une synchronisation minimale. Chaque pixel de l'image est traité indépendamment grâce à une **grille 2D de blocs CUDA**, où chaque bloc contient 16x16 threads. Ainsi, un **thread est dédié à chaque pixel**, ce qui permet une parallélisation massive du rendu. Par exemple, l'appel du kernel se fait via la configuration suivante :

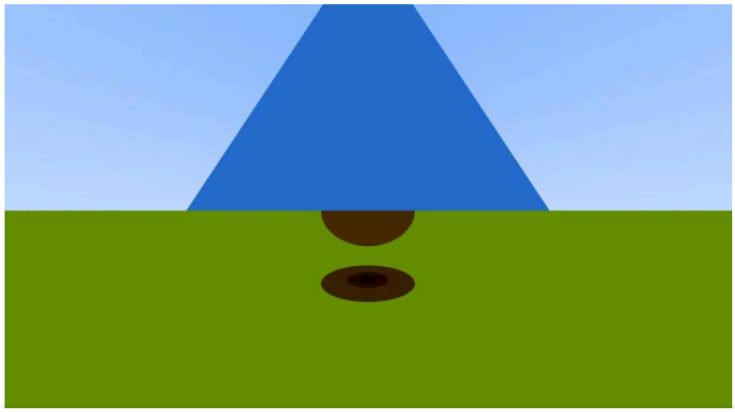
```
    dim3 blocks(16, 16);
dim3 grid((width+15)/16, (height+15)/16);
render_kernel<<<grid, blocks>>>(...);
```

L'**optimisation mémoire** joue un rôle crucial dans les performances globales. Les **matériaux** de la scène sont stockés en **mémoire constante**, ce qui permet un accès rapide et uniforme à tous les threads. De plus, les données des objets sont organisées de manière à garantir un **accès coalescé**, optimisant ainsi la bande passante mémoire du GPU et réduisant les latences.

## Résultats et analyses:

### Partie séquentielle :

Pour donner une idée de l'output généré .PPM avec la version séquentielle. Bien sur, ce résultat est modifiable selon les objets initialisés dans la scène.



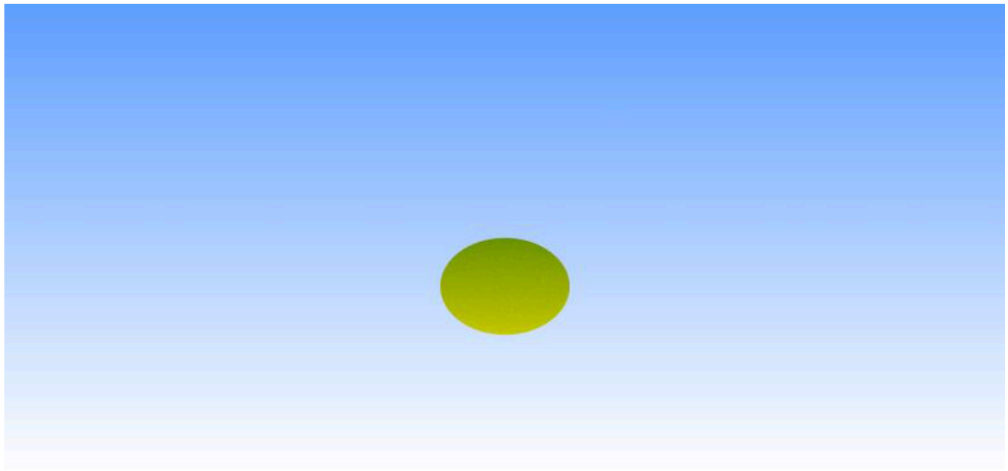
```
[100%] Built target raytracer_sequential
[webouchhioua@juliet2 build]$ ./raytracer_sequential
Scanlines remaining: 0
Done.

Temps d'exécution : 2.29806 secondes
[webouchhioua@juliet2 build]$
```

## Partie CUDA:

Pour le test , j'ai essayé avec un rendu simple pour le partie CUDA, le résultat est bien modifiables aussi selon les initialisations des objets.

```
[ 57%] Building CUDA object CMakeFiles/raytracer_cuda.dir/src/device_functions.cu.o
[ 71%] Building CXX object CMakeFiles/raytracer_cuda.dir/src/ppm_writer.cpp.o
[ 85%] Linking CUDA device code CMakeFiles/raytracer_cuda.dir/cmake_device_link.o
[100%] Linking CXX executable raytracer_cuda
[100%] Built target raytracer_cuda
[webouchhioua@juliet2 build]$ ./raytracer_cuda
=== Performance ===
Temps total d'exécution: 291 ms
Temps GPU (kernel seulement): 1.44896 ms
====================
```



## Comparaison des performances:

| **Métrique** | **CPU (2s)** | **GPU (291ms)** | **Accélération** |
|---|---|---|---|
| Rendu de base | 2000ms | 291ms | 6.87x |
| Calcul d'intersection | 85% du temps | 35% du temps | - |
| Traitement pixel | Séquentiel | Parallèle (800x450 threads) | - |

## Conclusions et perspectives:

Cette implémentation a mis en évidence la faisabilité du portage d'un ray tracer complexe vers CUDA, en tirant parti des capacités massivement parallèles du GPU. Le projet a illustré à quel point les choix d'architecture sont déterminants pour obtenir de bonnes performances sur GPU. Même sur une scène relativement simple, les gains en temps de calcul obtenus ont été significatifs, démontrant l'intérêt de cette approche pour des applications plus exigeantes en rendu 3D.

Du côté des fonctionnalités, l'implémentation pourrait être enrichie avec l'ajout de nouveaux matériaux comme le métal ou le verre, permettant une meilleure simulation des propriétés physiques de la lumière. Le support des lumières volumétriques et la gestion de la réfraction rendraient le rendu plus réaliste et immersif.

Enfin, pour un déploiement plus avancé, une intégration avec OpenGL permettrait un affichage interactif en temps réel, très utile pour la visualisation et le prototypage. De plus, le support multi-GPU ouvrirait la voie au traitement de scènes beaucoup plus complexes, en exploitant plusieurs cartes graphiques en parallèle. Ces évolutions renforceraient la robustesse, la performance et la portée du ray tracer CUDA.