Raytracing

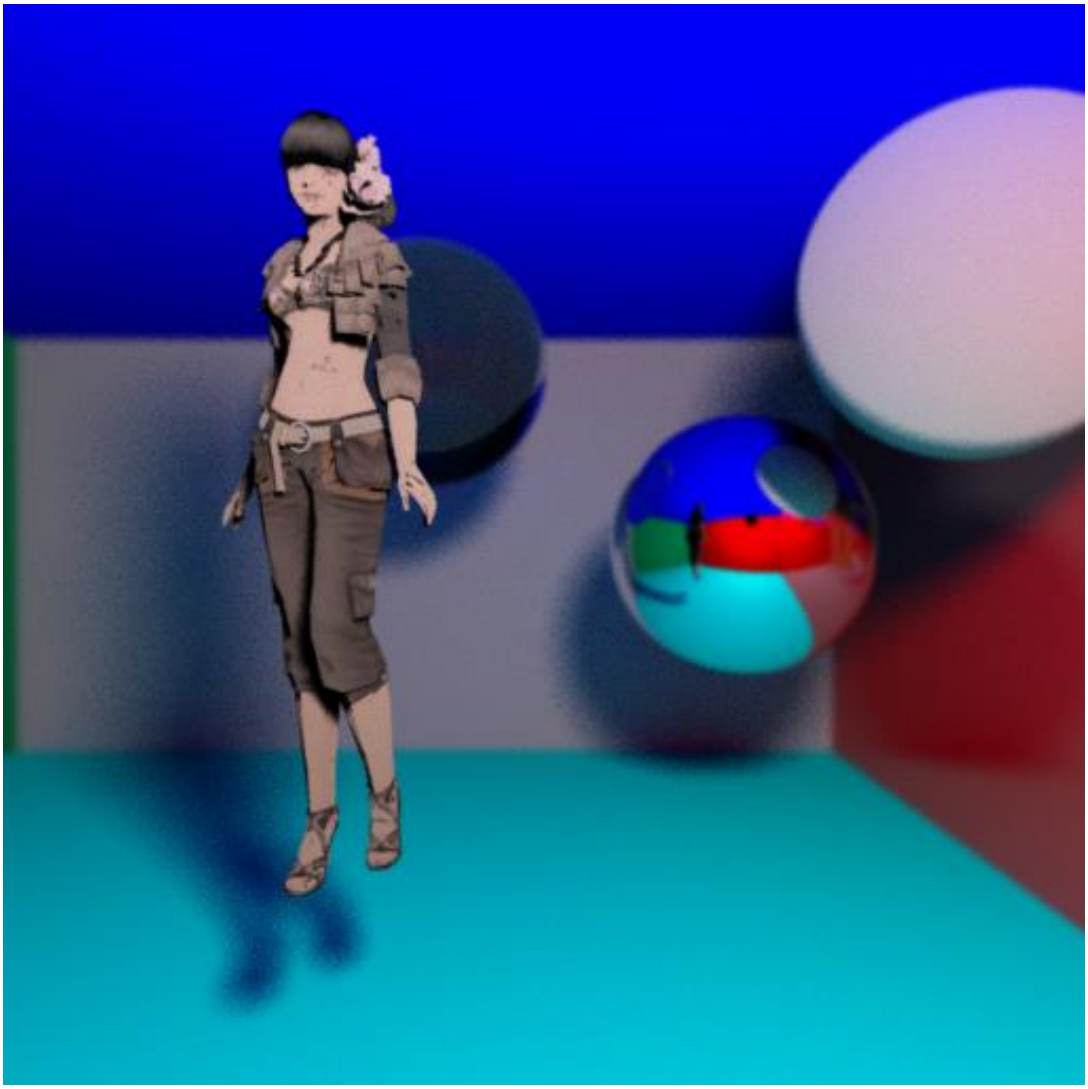
Cours 3ème année, Nicolas BONNEEL

Ecole Centrale Lyon, 2020

Rapport de projet

Noé PETERLONGO

Dépôt GitHub : https://github.com/NoePeterlongo/RayTracing.git



Source étendue, éclairage indirect, polyèdre texturé, profondeur de champ, anticrénelage, 100 rendus, 655s

Table des matières

Introduction

Le « Raytracing » est une méthode de rendu 3D, à l'instar des méthodes de « rasterization ». Elle est relativement lente, mais par construction, elle tend à obtenir des résultats physiquement réalistes. L'idée de base est de reproduire la course des rayons lumineux dans la scène, en prenant en compte leur trajectoire, leurs rebonds, en fonction de la nature des matériaux rencontrés. Les méthodes de « rasterization » elles se basent sur un ensemble d'astuces pour obtenir un résultat correct avec pour principal objectif la rapidité de traitement, c'est pourquoi elles sont utilisées dans le domaine des jeux-vidéos qui demandent du rendu en temps-réel. Le « Raytracing » quant à lui peut être utilisé dans des études qui demandent davantage de réalisme, comme par exemple les rendus pour l'architecture, ou encore des modélisations plus exotiques, comme celle des effets d'une lentille gravitationnelle.

Je présente dans ce rapport mon implémentation de la méthode de « Raytracing ». Il s'agit d'un programme développé en C++, disponible sur le dépôt github : https://github.com/NoePeterlongo/RayTracing.git. L'objectif est de réaliser le rendu d'une scène 3D contenant différents éléments, en appliquant diverses subtilités permettant d'améliorer le réalisme.

Principe général de rendu d'image

Pour chaque pixel de l'image, un rayon est émis, avec pour origine la position de la caméra, et dont la direction dépend du pixel et de l'orientation de la caméra. Ce tracé de rayon a pour objectif de déterminer la couleur du pixel correspondant. Cette couleur dépend des objets rencontrés par le rayon, ses rebonds, ses déviations.

Dans la réalité, le rayon arrivant à la caméra est issu d'une multitude de rayons. Tous ces rayons ne pouvant pas être pris en compte, un seul d'entre eux est pris en compte à la fois, et l'image est rendue plusieurs fois, et le rendu final est une moyenne de ces images. Cela permet notamment l'éclairage indirect, l'anticrénelage, la prise en compte de sources de lumière étendues

Le fichier parametres.h détermine des paramètres de rendu, comme par exemple l'exécution de l'anticrénelage et le nombre de rayons à envoyer par pixel.

Dans le programme, la fonction CalculerImage est chargée de déterminer les rayons à tracer, en fonction de divers paramètres de rendus, de la position et l'orientation de la caméra… Le multithreading est réalisé avec la bibliothèque standard <thread> ; chaque « thread » (ici ça s'appelle comme ça, mais ce n'est peut-être pas rigoureux) appelle la fonction CalculerImage pour calculer un certain nombre de lignes. Sont créés plus de threads que de cœurs physiques, car certaines zones de l'image sont plus rapides à calculer que d'autres ; créer un grand nombre de threads permet de s'assurer que le processeur reste pleinement occupé.

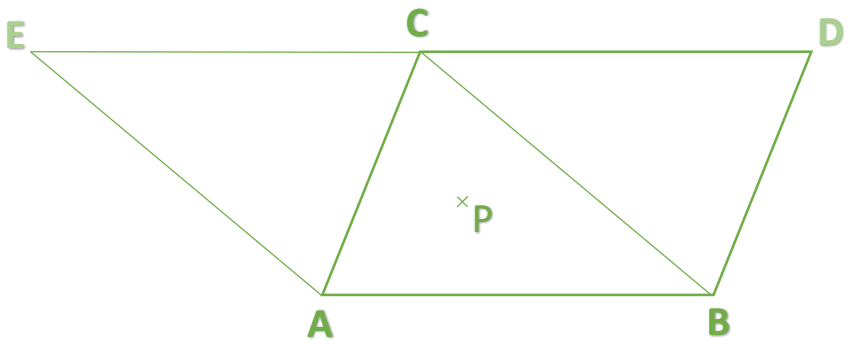
Les différents objets présents (sphères, polyedres, lampes) appartiennent à un objet de type Scene. Cet objet détient aussi la fonction GetColor, qui est chargée de déterminer la couleur renvoyée par un rayon. Afin de prendre en compte les rebonds, la fonction GetColor est récursive.

Les prochaines parties détaillent certains points notables de l'implémentation.

Les polyèdres

Par simplicité, les premiers éléments modélisés sont des sphères, mais nous avons ensuite réalisé des polyèdres. Ayant pris de l'avance sur le cours, et voulant expérimenter par moi-même, j'ai implémenté les polyèdres un peu différemment du cours.

L'objet de base pour modéliser un objet (et ce surtout pour les méthodes de « rasterization ») est le triangle. Les objets 3D (à l'exception ici des sphères) sont ensuite modélisés comme des ensembles de triangles. La première étape est donc de modéliser un triangle, c'est-à-dire déterminer le point d'intersection entre un rayon et un triangle. Pour cela j'ai mis au point (réinventé sans doute) une méthode un peu différente du cours.



Prenons la figure ci-dessus. Nous nous intéressons au triangle ABC. La première étape est de déterminer le point d'intersection du rayon avec le plan (ABC), appelons, s'il existe, ce point P. Déterminer ce point est assez simple, nous ne le détaillons pas ici. Pour que le point P soit dans le triangle ABC, il doit être à la fois dans le parallélogramme ABDC et dans ABCE. Pour savoir si le point P est dans le ABDC, on détermine ses coordonnées dans le repère (A, AB, AC), si ce coordonnées sont comprises entre 0 et 1, alors P est dans ABDC. Un raisonnement identique permet de conclure pour ABCE.

Pour obtenir les coordonnées, on passe par un repère orthonormé. Ces opérations demandent le calcul de divers coefficients, qui restent constants pour un triangle donné. C'est pourquoi, afin de réduire le temps d'exécution, tous les coefficients constants sont calculés à la création du triangle, puis stockés.

Les triangles sont créés par et stockés dans un objet polyèdre. L'objet polyèdre se charge des rotations éventuelles, de la lecture de fichiers .OBJ et .STL, et des intersections de rayons.

Afin d'optimiser l'intersection avec les polyèdres, ils comportent des boîtes d'accélération. Lorsqu'un rayon arrive, l'objet détermine d'abord avec quelle(s) boîte(s) il rentre en contact. Ensuite, sachant quels triangles sont contenus dans quelles boîtes, les triangles pertinents sont interrogés. Ici on utilise deux niveaux de boîtes : une boîte principale qui englobe tout l'objet, et un ensemble de boites non séquentes qui divisent la boîte principale. La boîte principale a deux usages ; le premier est de déterminer si le rayon arrive dans la zone de l'objet ; le second est de contenir les triangles qui ne seraient entièrement inclus dans aucune boîte secondaire. Les boîtes secondaires vides sont supprimées.

Pour réduire au maximum le temps de rendu, il faut bien sélectionner le nombre de boîtes secondaire. Ici, cela est fait au cas par cas, cela dépend de la géométrie de l'objet, et du nombre de triangles. On teste dans un premier temps le rendu d'une image plus petite pour voir quel nombre de boîtes minimise le temps de rendu.

J'ai écrit une fonction pour lire les fichiers .STL, et récupéré une implémentation de la lecture des .OBJ. Cette dernière comprend une prise en charge des textures. Grâce à un jeu de coordonnées associant aux sommets des triangles une position dans l'image de texture, on peut déterminer la couleur du point d'intersection entre un rayon et un triangle.

Fonctionnalités implémentées

Voici une liste non exhaustive des fonctionnalités implémentées.

- Éclairage à partir d'une ou plusieurs sources ponctuelles
- Éclairage à partir d'une source étendue (sphère)
- Transparence, fresnel
- Surfaces spéculaires
- Éclairage indirect
- Spheres, polyèdres
    - Lecture de STL, OBJ
    - Gestion de texture des OBJ
    - Rotation des objets
- Rotation de la caméra
- Profondeur de champ
- Anticrénelage
- Petit moteur physique

Moteur physique

J'ai pris l'initiative d'ajouter un petit moteur physique à la scène afin de réaliser de petites animations un peu réalistes. Ce moteur ne prend en compte que les sphères. Le moteur physique a été programmé de sorte à interférer le moins possible avec le système de rendu graphique : il détient toutes les informations mécaniques sur les sphères, et ne modifie que leur position. Pour chaque sphère de la scène, on crée un objet sphère physique SpherePhy, qui appartient au « MoteurPhysique ». À chaque sphère est associée une capacité à bouger ou non, un coefficient de rebond, et un coefficient d'amorti. Le moteur physique est chargé de calculer les déplacements des sphères entre deux images de rendu. À noter qu'entre deux images, il y a plusieurs itérations du moteur physique, pour améliorer le réalisme.

Concernant les rebonds, différentes méthodes ont été testées. Initialement, les rebonds devaient être déterminés en fonction de l'angle d'incidence, la vitesse de contact. Finalement c'est une méthode inspirée de la réalité physique qui a porté ses fruits. Les sphères peuvent s'interpénétrer, et exercent les unes sur les autres une force qui dépend de la profondeur du contact. Cela est géré par la

fonction ForceSubieDe ci-dessous. L'accélération de chaque sphère mobile est alors mise à jour en fonction de la gravité, et des autres forces.

```
21  Vector3 SpherePhy::ForceSubieDe(const SpherePhy &_sphere)
22  {
23      Vector3 centreACentre = *pPosition - *(_sphere.pPosition);
24      double distanceSpheres = std::sqrt(centreACentre.Norme2()) - rayon - _sphere.rayon;
25
26      if (distanceSpheres > 0)
27          return VECTEUR_NUL;
28      else
29      {
30          double coefAmorti = (_sphere.vitesse - vitesse).Dot(centreACentre) > 0 ? 1 : bounciness;
31          double intensite = coefAmorti*std::exp(-distanceSpheres*20);
32          return centreACentre.Normaliser()*intensite;
33      }
34  }
```

Le moteur physique a permis d'obtenir des vidéos disponibles à ce lien :
https://drive.google.com/open?id=1d3A91IeotOLWflh0j832dqH2QcH-WSXQ

Conclusion et avis personnels

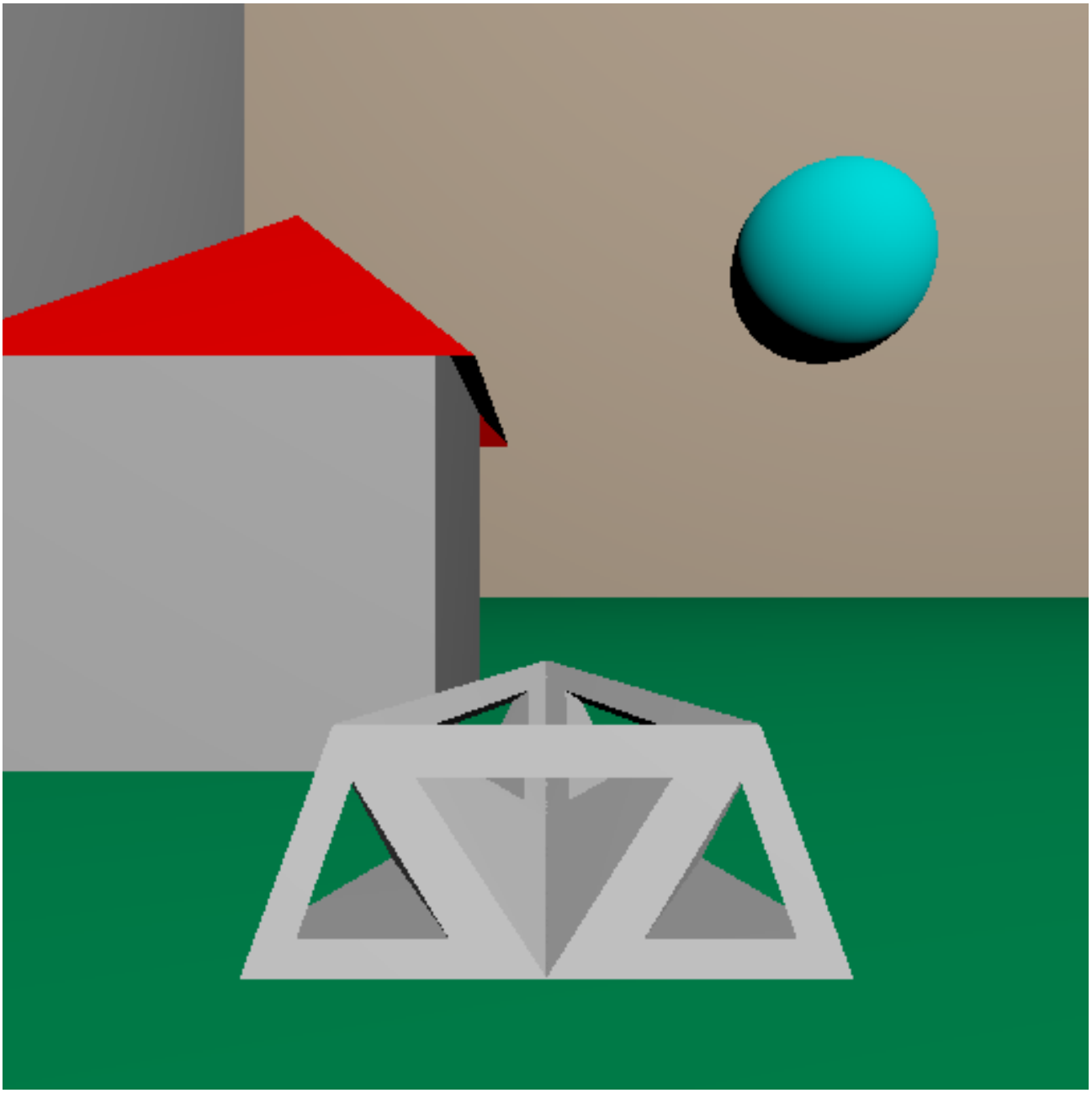
Le cours d'informatique graphique nous a permis de réaliser un moteur de rendu graphique à partir de quasi rien. Je trouve très bonne cette idée de partir de zéro, les autres méthodes pédagogiques ont souvent le don de me dégoûter de la programmation. Le sujet m'intéressant, j'ai pris de l'avance sur le cours, en ne suivant que le cours sur le drive, et parfois sans le lire pour me débrouiller. Il n'était parfois pas évident de concilier mes solutions avec la suite du travail.

"Sometimes science is more art than science, Morty. A lot of people don't get that"

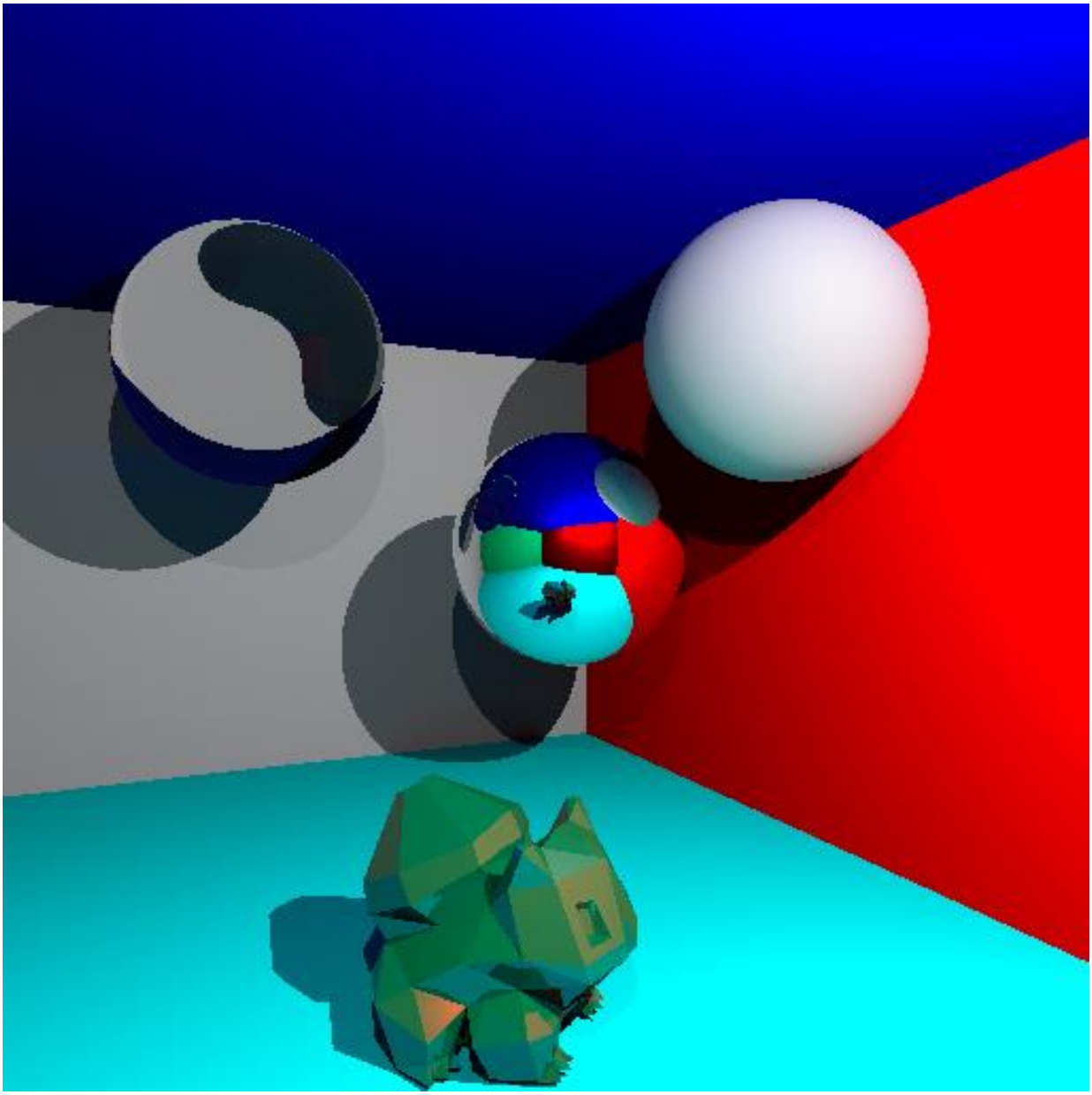
R.S.

Annexe : rendus

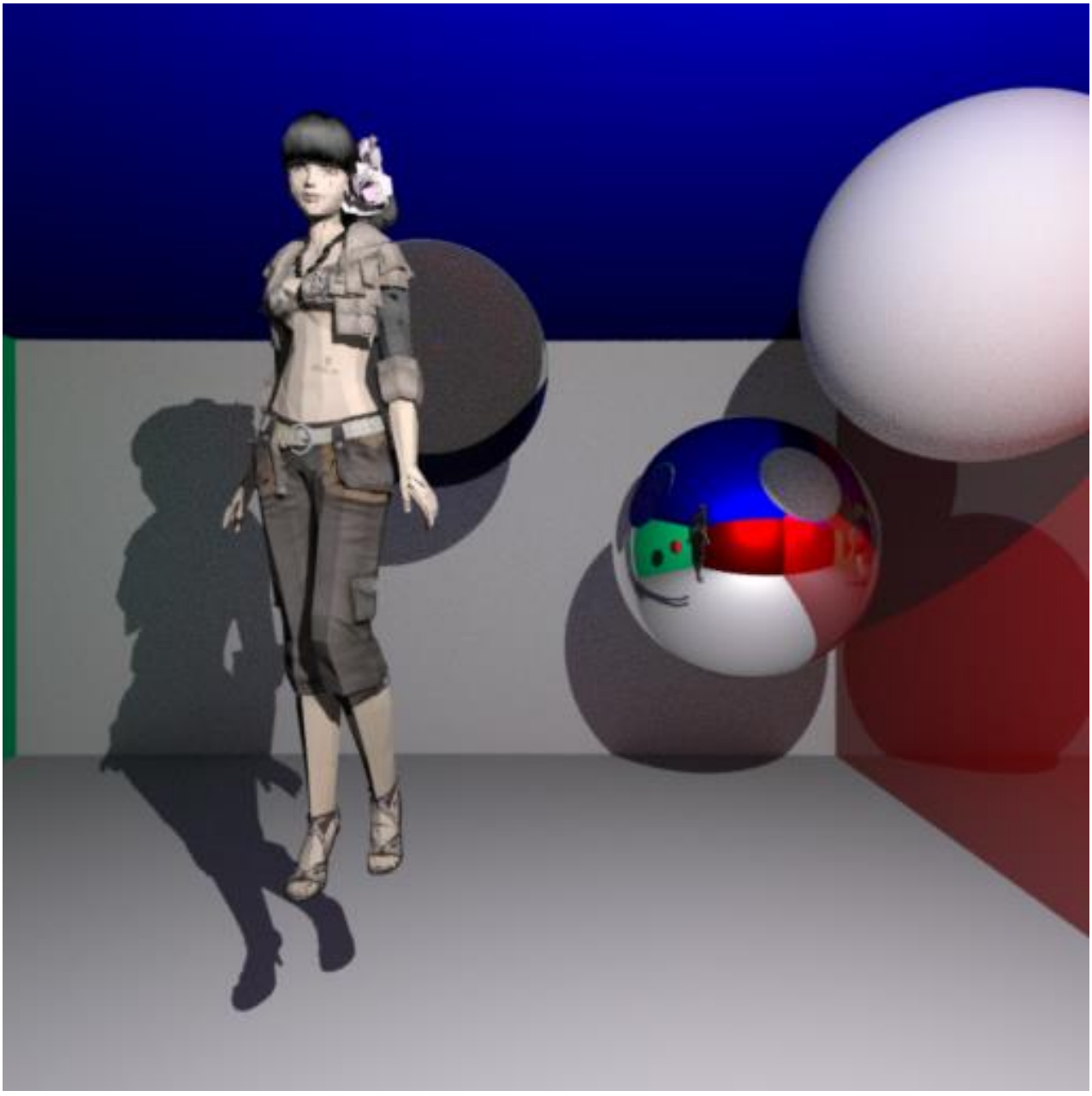
Rendus vidéos accessibles ici :
https://drive.google.com/open?id=1d3A91IeotOLWflh0j832dqH2QcH-WSXQ



Affichage d'un objet de type STL (réalisé sur CATIA au passage, c'est un support pour rubik's cube)



Affichage d’un objet de type STL



Objet OBJ texturé, source ponctuelle

Alduin texturé, source ponctuelle (deux sources)

Alduin texturé, source étendue

Alduin avec profondeur de champ mal réglée