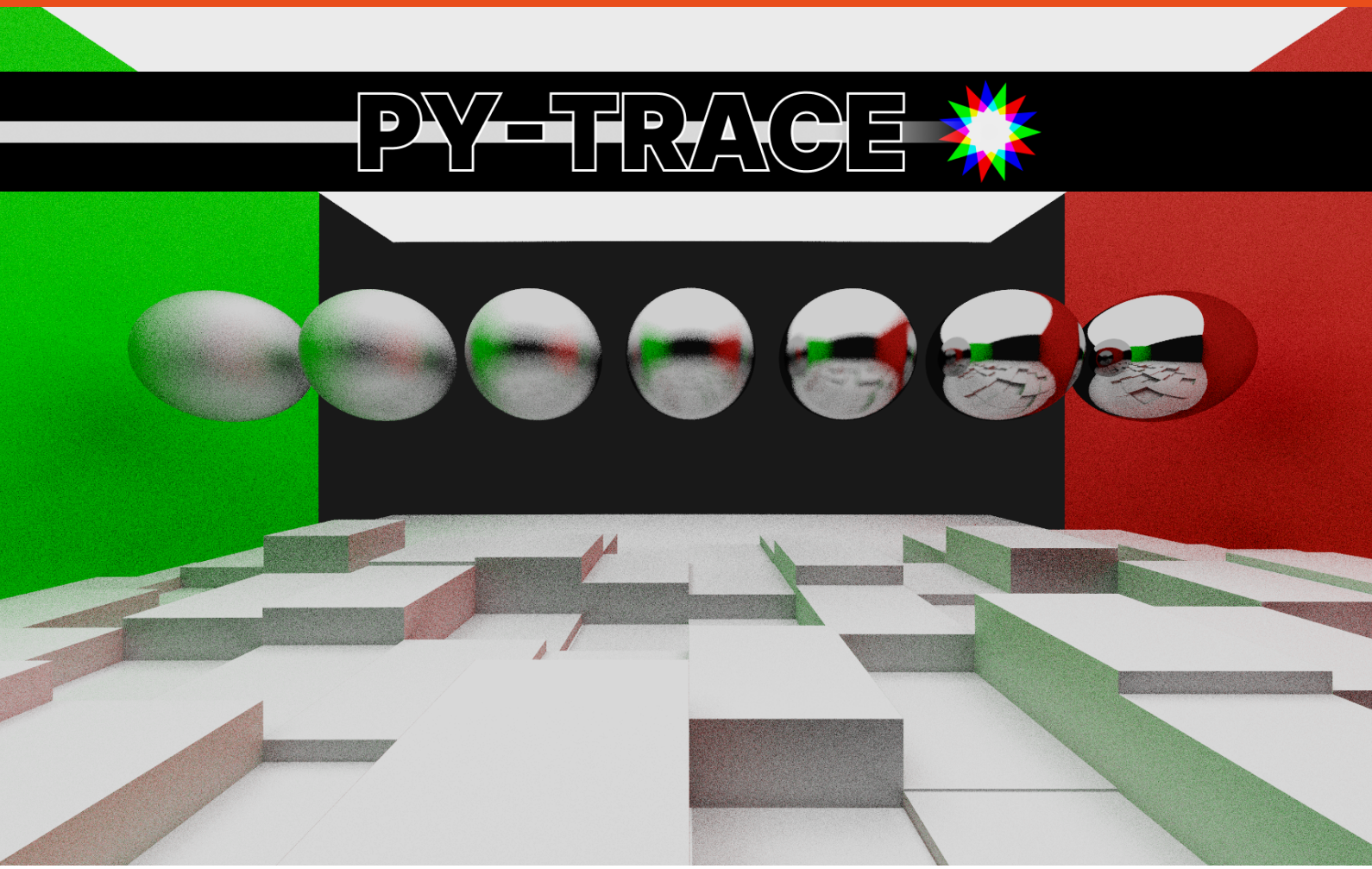
LES TROPHÉES NSI

Édition 2023

DOSSIER DE CANDIDATURE
PRÉSENTATION DU PROJET



Ce document est l'un des livrables à fournir lors du dépôt de votre projet : 4 pages maximum (hors documentation).

Pour accéder à la liste complète des éléments à fournir, consultez la page **Préparer votre participation**.

Vous avez des questions sur le concours ? Vous souhaitez des informations complémentaires pour déposer un projet ? Contactez-nous à **info@trophees-nsi.fr**.

NOM DU PROJET : PY-TRACE

> PRÉSENTATION GÉNÉRALE :

La lumière est un des phénomènes les plus difficiles à reproduire dans le domaine de la 3D. Nombreuses sont les technologies qui ont essayé de reproduire ce phénomène si particulier et qui y sont arrivées telles que le ray-tracer ou plus communément appelé RTX. (que l'on peut voir dans minecraft RTX par exemple, ou encore cyberpunk 2077).

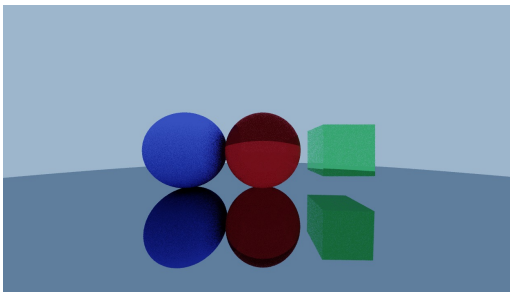
Mais est-ce si compliqué que ça ?

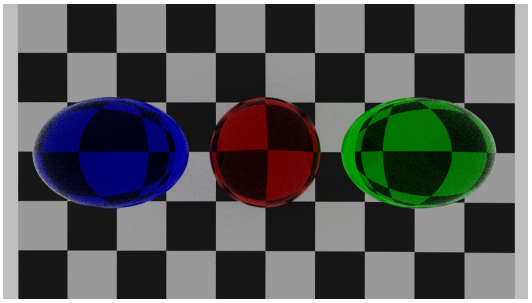
Le raytracing c'est une technologie qui n'a pas encore été démystifiée, beaucoup de gens connaissent le principe du raytracing mais peu en connaissent vraiment le fonctionnement et les nombreuses notions qui rentrent en jeu.

Le projet est alors né quand on a découvert que le raytracing n'était pas un algorithme si compliqué que ce que l'on pouvait croire. Et que l'on pouvait rendre de belles images avec (comme ça, on peut faire des fonds écran).

Mais finalement, qu'est-ce que cette technologie apporte de plus ?

Cette technologie permet très simplement de décrire des matériaux compliqués tel que du verre ou du métal sans devoir avoir recours à la triche, ou à des alternatives limitées. Par exemple, dans les jeux, les miroirs et vitres sont limités et compliqués, or avec le raytracing, c'est très simple de l'utiliser. De plus, ce rendu réaliste de la lumière est également utilisé dans les productions, le dernier avatar (et tous les films qui sont sortis ces 30 dernières années) utilisent cette technique pour produire des images de synthèses réalistes.





Ici, à gauche, le rendu d'un miroir réaliste et à droite le rendu de sphères de verre pareillement réalistes avec notre raytracer.
Le but de notre projet est donc, outre le fait de créer un ray-tracer, d'en synthétiser le fonctionnement et de le retranscrire en python donc d'une façon qui se vaut beaucoup plus simple et plus compréhensible.

> ORGANISATION DU TRAVAIL :

Notre équipe est constituée de Sanjay SAGET-GOBARDHAN, Clément BELAISE , Mathis LEDA, Cyprien BOUVIER.

On travaillait souvent en groupe durant les cours de NSI ce qui nous permettait de programmer ensemble et ainsi être plus rapide et productif. Comme ça, lorsqu'il y avait un problème on pouvait facilement le régler en coopérant.
De plus, on a appris à utiliser l'extension de VS Code: "Live Share" qui nous permettait de partager une même machine et de programmer ensembles en direct.

Le code source quant à lui était stocké de manière libre et open source sur GitHub ici: (https://github.com/Supercip971/py-trace) . Cela nous permettait de versionner notre code, de le stocker et qu'il soit simplement accessible.

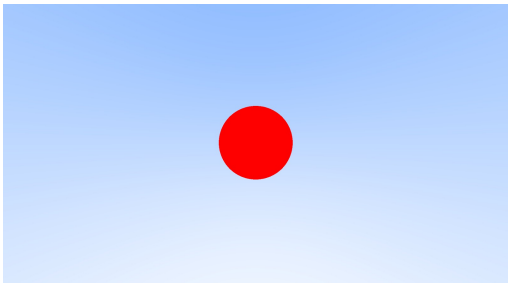
Pour communiquer lorsque l'on était dans l'impossibilité de se voir, on utilisait WhatsApp et Discord pour échanger. On utilisait également Trello pour gérer les tâches.

L'équipe était alors "divisée" en 4:

- Sanjay: Gère le dossier technique, a aidé à la programmation d'une partie des matériaux, modélisateur 3D.
- Clément: A programmé le code des vecteurs, de la caméra, et de la classe Ray. Il a par ailleurs fait le joli logo.
- Mathis: A aidé à la programmation du code d'intersection entre les formes, a aidé au dossier technique, gère la vidéo et son montage.
- Cyprien: Programmeur en "chef", a assisté à la programmation des matériaux et du code d'intersection entre les formes et le rayon lumineux, a par ailleurs assisté quant à la modélisation 3D de certaines scènes.

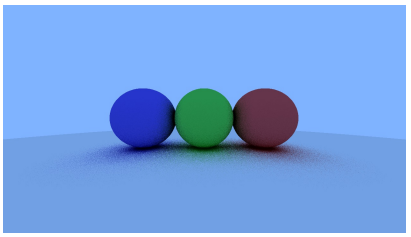
LES ÉTAPES DU PROJET :

Premièrement on devait trouver une idée, donc on a cherché, puis Cyprien a proposé l'idée d'un raytracer. Il s'y connaissait un peu et en avait entendu parler (car il avait cherché ce sujet pour son grand oral du bac). On est ensuite tombé sur une vidéo de quelqu'un qui expliquait comment l'algorithme se déroulait et on s'est senti capable de le faire.

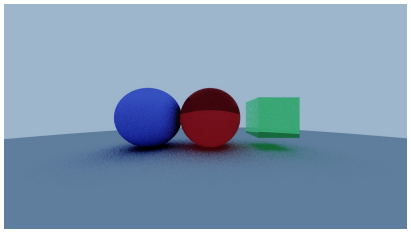


On a alors directement réparti les tâche, mais il fallait beaucoup de code initial avant de faire un rendu basique et donc voir quelque chose : un accès pixel par pixel de l'écran, on a d'abord utilisé tkinter, puis finalement on a changé vers pygame (qui était plus rapide). On a ensuite implémenté un code de math pour les vecteurs 3D, une gestion de scène, un code pour la caméra. Mais on a finalement réussi à faire un rendu basique d'une sphère après une semaine.

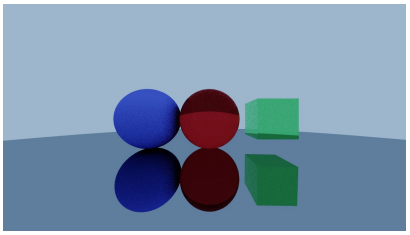
On a ensuite implémenté le système des matériaux et les rebonds des lumières (qui est une part très importante du projet). On a choisi le matériau lambertien (ou orthotrope), car en général, c'est le plus simple à implémenter (il tient en quelque ligne de code).



Le projet commençait déjà à prendre forme, mais on a décidé de “refactor” le code pour le rendre plus lisibles par tous, et éviter de se tirer une balle dans les semaines à venir.
De plus, la partie la plus compliquée était faite, maintenant c’était juste des maths, de la physique, du Banga et non du Caresse Antillaise.



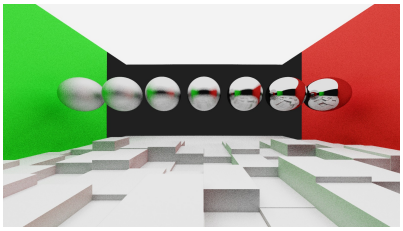
Par la suite, on a décidé d’implémenter un cube (avec Cyprien et Mathis) et en parallèle Sanjay et Clément ont rajouté le verre.

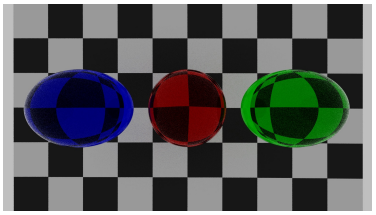


Pour continuer, on a commencé à rajouter le métal, et on a rajouté un tonemapper et du code markdown pour expliquer certains éléments complexes du code (les maths, les idées derrières) qui ne peuvent pas être résumés à un simple commentaire.

On a ensuite essayé de trouver une manière d’optimiser. En effet, le code était extrêmement lent, et sa lenteur commençait à heurter nos capacités, cependant, après avoir essayé, pypy, python 3.11 (et les nouvelles versions qui promettent d’être plus rapide), cython, utiliser numpy… On s’est rendu compte que le gros souci est python en lui-même, car le code tournait 10 000 fois plus lentement que dans un exemple en C ou sur le GPU. (avec une simple sphère).
Rien que remplir une fenêtre de pixels d’une couleur (sans rien d’autre en plus) est rendue à 20 fps, donc on a dû se rendre à l’évidence : on ne pourra pas rendre le code plus rapide.

Ainsi, on a également décidé de faire plusieurs scènes, des modèles ou autre, on a premièrement utilisé blender pour certaines puis on les a retranscrites dans le programme en code python.
Par exemple, le modèle du bonhomme de neige à été fait par Sanjay.
En général, on faisait un rendu simple sur place, par la suite, on faisait le rendu final la nuit (car il prend beaucoup de temps) sur un autre pc. Voilà certaines scènes que l’on a pû produire pendant le développement : (le bonhomme de neige s’appelle bob).

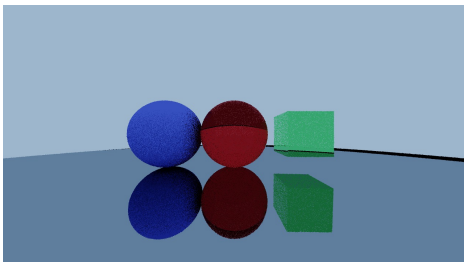




> FONCTIONNEMENT ET OPÉRATIONNALITÉ :

Le projet est, en réalité, terminé, mais c'est le genre de sujet qui sont en fait jamais terminés, de nombreux éléments restent encore à rajouter, mais on se rapproche énormément du hors programme, et cela rendrait PY-TRACE extrêmement complexe.
Mais on aurait aimé, si les performances étaient plus potables, d'éditer les scènes en direct et de pouvoir naviguer dans la scène (mais à 10 minutes par images, c'est impossible).

Quant à la détection de bugs, la scène étant directement décrite python, certaines erreurs sont directement corrigées.
On a pu rencontrer différentes difficultés, par exemple : on a découvert un bug, qui nous a pris du temps à résoudre, en effet, lorsque l'on faisait une réflexion à une certaine distance, on avait une ligne noire visible ici. Ce bug était produit seulement sur un angle (-y, -z) et il nous a hanté pendant un moment. On l'a premièrement mise sous le tapis (comme tout bon développeur) puis finalement, on a dû se résoudre à le régler. En fait, c'était dû à un bug dans le code de l'intersection de la sphère (on avait un problème dans nos maths lorsque l'on avait essayé de trouver la formule pour l'intersection).



> OUVERTURE :

Une idée principale d'amélioration serait d'abandonner totalement le python, qui est trop lent pour ce genre de rendu. On peut également introduire du code C dans le python pour palier aux performances, mais pourquoi écrire du C dans du python quand on peut directement écrire en C ?
Une idée d'amélioration qui serrait bonne serrait de transformer notre raytracer pour qu'il soit plus professionnel. En utilisant par exemple des fonctions de probabilités de densités des matériaux pour pouvoir facilement rebondir vers les lumières plutôt qu'aléatoirement (ce qui produit des images plus rapidement). De plus, on pourrait introduire une structure de données qui diviserait la scène pour un rendu plus rapide (Bvh, octree...). On pourrait par ailleurs implémenter un BSDF qui est une sorte de seul matériau universel qui permet de presque tout faire (verre, plastique, métal, ...). On aimerait aussi rajouter une interface graphique qui permettrait de sélectionner les scènes.

Pour vendre le projet, on pourrait le vendre comme une nouvelle alternative libre et open source dans Blender pour faire des rendus de scène. Car Blender permet d'intégrer des systèmes de rendus simplement, mais il y a peu de bonnes alternatives open sources qui ont un code simple et simplement modifiable. Et on pourrait faire des beaux rendus comme démonstration.

Si c'était à refaire, on le referait, mais on utiliserait probablement un autre langage, ou on l'implémenterait sous forme de rendu dans un shader dans la carte graphique de l'ordinateur. Car elles sont très bonnes dans les rendus parallélisés. Et on aimerait faire une interface qui est simple d'utilisations à la place d'interfacer dans la ligne de commande.

DOCUMENTATION

• Spécifications fonctionnelles (guide d'utilisation, déroulé des étapes d'exécution, description des fonctionnalités et des paramètres)
• Spécifications techniques (architecture, langages et bibliothèques utilisés, matériel, choix techniques, format de stockage des données, etc)
• Illustrations, captures d'écran, etc

Premièrement, il faut installer les librairies pygame et numpy pour pouvoir lancer le programme, de plus on recommande d'utiliser Linux avec python 3.10 ou plus récent, car le programme a été testé sur ces plateformes.

Ainsi, le programme ne dispose pas d'interface graphique, mais doit être paramétré dans la ligne de commande, il faut premièrement entrer dans le dossier *sources*, puis exécuter la commande :

```
$ python ./sources/main.py --scene metal-demo
```

Ici, on a le paramètre –scène qui permet de sélectionner la scène, pour avoir une liste des scènes disponibles, exécutez le programme sans paramètre.

```
$ python ./sources/main.py --scene metal-demo --width 1920 --height 1080
```

De plus, on peut paramétrer la hauteur et la largeur dans les paramètres, car à causes de limitations, on ne peut changer la taille qu'au lancement du programme.
Si vous voulez des informations à propos des paramètres, vous pouvez toujours exécuter le programme avec:

```
$ python ./sources/main.py --help
```

Pour rajouter une scène, il faut modifier le fichier scenes.py, et rajouter une scène, ici, on va créer une simple scène nommée 'tuto'.

Ici, ça va être un guide sur comment créer et personnaliser une scène basique.

```
import scenes.basic
import scenes.demometal
import scenes.snowman
import scenes.demoglass
import scenes.light_demo
import scenes.tuto


scenes_list = {
   "glass-demo": scenes.demoglass,
   "metal-demo": scenes.demometal,
   "light-demo": scenes.light_demo,
   "basic": scenes.basic,
   "snowman": scenes.snowman,
   "tuto": scenes.tuto,
}
```

On rajoute les lignes en vertes, qui décrivent où chercher la scène nommée ‘tuto’, puis dans le fichier tuto on rajoutes ce code:

```
from camera import Camera
from shapes.sphere import Sphere
from shapes.box import Box
from vector import Vec3
from materials.lambertian import Lambertian
from materials.glass import Glass
from materials.metal import Metal
from world import World
from color import Color


def load():
   return None
```

Ici, le code est basique, on a tout importé et si vous exécutez le code vous aurrez une érreur, car on ne retournes rien du tout.

On va devoir décrire une caméra:
Premièrement on doit prendre sa position, on va prendre (0,0,1), là ou elle regardes (0,0,0) et le dessus (pour son orientation) et le dessus est en (0,1,0) (+y). On doit aussi prendre un FOV (field of view) et un ratio d’image, mais les deux derniers paramètres sont en général à 90° et 16/9:

```
def load():
      # Position, Où il regardes, le dessus, le fov, le ratio d'écran
      camera = Camera(Vec3(0, 0, 1), Vec3(0, 0, 0), Vec3(0, 1, 0), 90,
16.0/9.0)
```

Ensuite, on définis le “monde”, qui est doté d’une caméra et d’un fond, c’est l’objet qui va stocker tout les éléments de la scène :

```
def load():
    # On décrit la caméra, sa position, là ou elle regarde...
    camera = Camera(Vec3(0, 0, 1), Vec3(0, 0, 0), Vec3(0, 1, 0), 90, 16/9)

    # on définit le monde avec sa caméra et la couleur de fond
    world = World(camera, Color(0.5, 0.7, 1))

    return World
```

Avant de rajouter des objets on doit définir quel matériaux on va utiliser, ainsi on va créer deux matériaux: un métal et un lambertien. Ils sont tout les deux des objets dont on doit préciser la couleur. On doit également préciser la rugosité pour le métal:

```
# Un lambertien (matériaux équivalent à du papier, ou du plâtre)
lambert = Lambertian(Color(0.3, 0.3, 0.8))

# Un métal de couleur (0.5,0.5,0.5) et une rugosité de 0.001 (très lisse)
metallic = Metal(Color(0.5, 0.5, 0.5), 0.001)
```

Ici, on aura une sphère métallice (presque-)grise très lisse et une sphère qui ressemblera à du platre bleu.

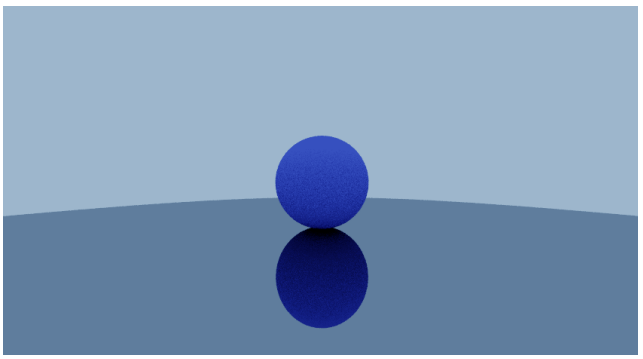
Finalement on peut rajouter les deux sphères à notre monde:

```
# Nos deux sphères, avec respectivement, leur coordonées, rayon et matériaux
sphere = Sphere(Vec3(0, 0, -1), 0.5, lambert)
big_sphere = Sphere(Vec3(0, -100.5, -1), 100, metallic)

world.add_shape(sphere)
world.add_shape(big_sphere)
```

Finalement, on doit retourne le monde dans la fonction load.

Une fois terminé, on peut exécuter le programme et on aura un rendu :
On comprend alors que l’on peut facilement décrire des objets, leurs matériaux, et leurs propriétés à partir d’un fichier python. Si on le souhaite, on peut

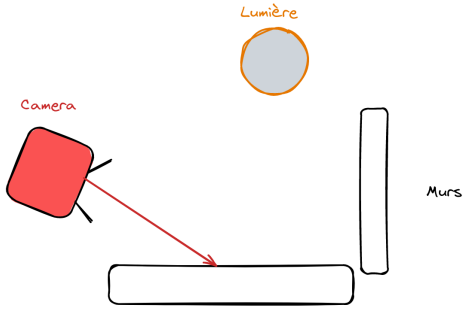


générer des sphères de manière aléatoires.
Vous pouvez par exemple changer la couleur du métal en rouge et vous pourrez facilement voir le changement.

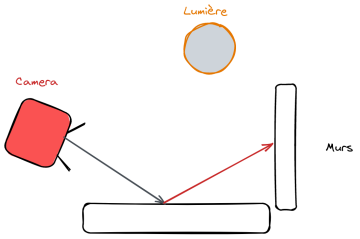
Déroulé d'exécution :

Premièrement, on analyse les arguments passés. Ensuite, on charge la scène correspondante, pour finalement faire le rendu pour chaque pixel.
Le rendu est le cœur du programme :

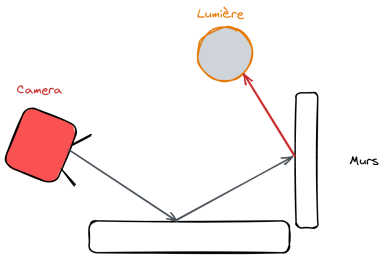
- On sélectionne un pixel,
- On transforme les coordonnées du pixel en coordonnées de 0 à 1 utilisable par la caméra.
- On demande à la caméra un rayon correspondant au pixel.
- On fait un algorithme récursif qui traverse la scène et qui simule le chemin d'un seul rayon. Représenté sur le schéma



Premièrement, on a le rayon de la caméra, il va toucher un objet. Puis à partir du matériau de l'objet, on va demander où le rayon va rebondir ? Du verre pourrait donner une direction interne, un miroir une réflexion, un lambertian une direction aléatoire… Ici, on considère un rebond. On appliquera également une couleur au rayon (un mur bleu, du bleu, etc.)

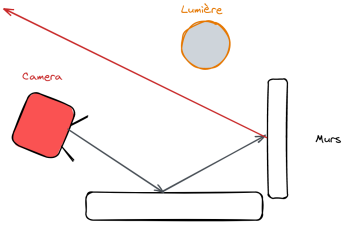


Ici, le rayon rebondis une deuxième fois, le processus se répète, mais ici, on aura deux situations.



Si le rayon rebondi vers une lumière, alors, on colore le pixel et on arrête la fonction.

Mais si elle ne touche rien du tout, ensuite, on applique la couleur du ciel.



Ainsi, on répète le procédé des millions de fois et on fait la moyenne des couleurs des rayons pour un pixel, pour avoir une image qui ressemble à la réalité.

Voir le README.md https://github.com/Supercip971/py-trace/blob/main/README.md

Voir la documentation complète https://github.com/Supercip971/py-trace/tree/main/doc