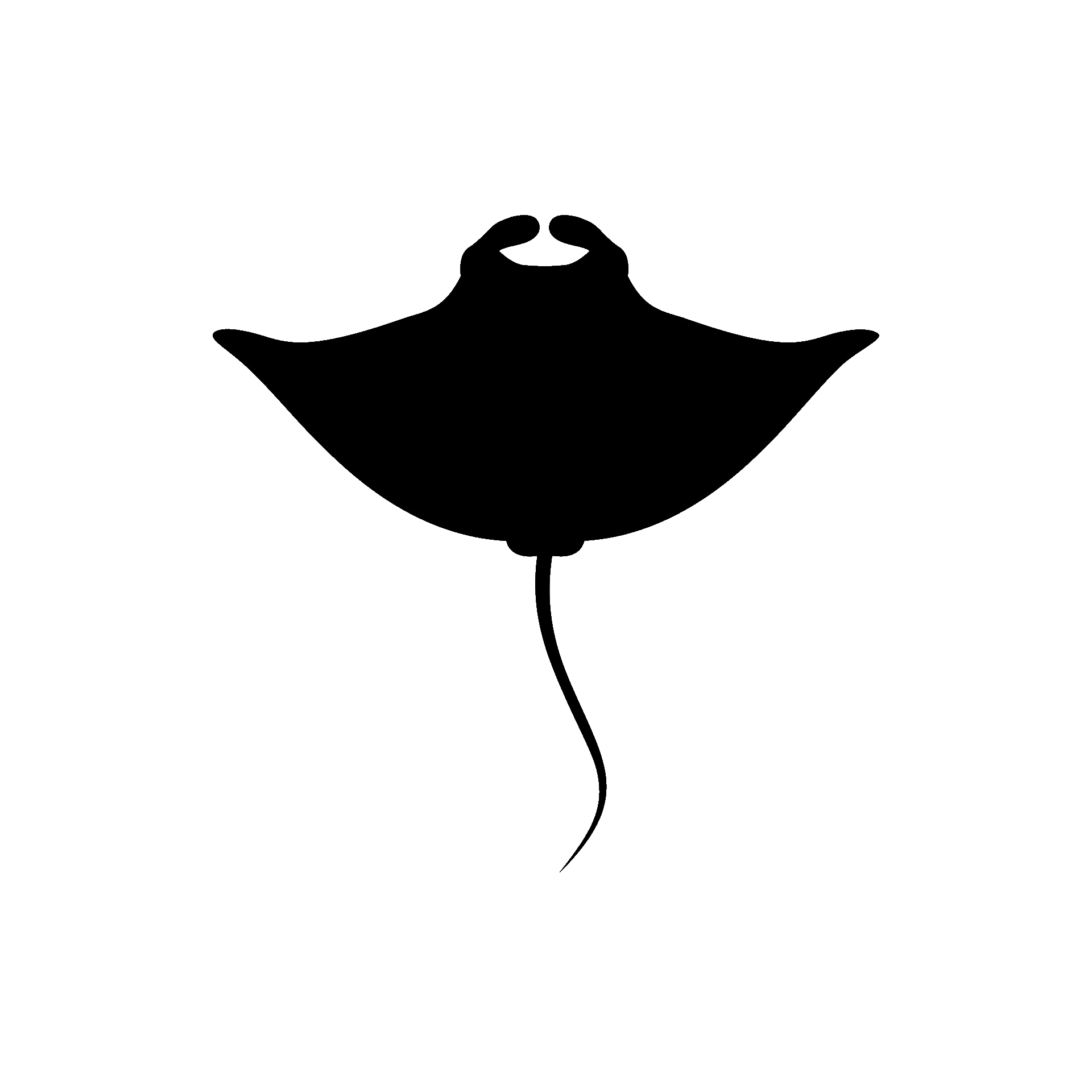
Projet Ray-Tracer

Diagramme de CLasses

#### Par Laurent Louis Lauret



Ray

tracer

Table des matières

[Diagramme de classes 3](#_Toc130904576)

[Explications 4](#_Toc130904577)

[Interfaces 4](#_Toc130904578)

[Classes 5](#_Toc130904579)

[Classes (suite) 6](#_Toc130904580)

[Patrons utilisés : 7](#_Toc130904581)

## Diagramme de classes

Toutes les classes ne sont pas forcément représentées, c’est le cas notamment des classes qui dérivent de **BaseProcessor**.

## Explications

### Interfaces

**IOut** : interface qui représente l’abstraction du flux de sortie. Cette interface dispose d’une méthode permettant l’écriture vers un flux de sortie.

**IScene** : interface représentant l’abstraction qui permet le traitement des rayons. Cette interface retourne une couleur (*Color*), ainsi que les coordonnées du pixel associé.

**IForm** : interface représentant les acteurs (géométries) de la scène qui seront frappés par les rayons, cette interface retourne un *HitResult*

**IProcessor**: interface qui représente l’abstraction qui traite les résultats de l’intersection des rayons (*HitResult*), et retourne à la scène une couleur (*Color*).

**IBuilder** : interface qui représentant le montage de scène. La scène est un objet complexe qui nécessite une configuration.

**IMultiProcManufacture** : interface représentant une manufacture abstraite pour instancier une famille d’objet nécessaire au multiprocessing au cœur de l’application.

**IParallelizer** : Interface représentant la gestion des processus dans le cadre du traitement parallèle.

**IReassembler** : Interface représentant la reconstitution d’une image à partir de morceaux.

**ICleaner** : Interface permettant d’effacer les fichiers temporaires à la suite du traitement parallèle.

### Classes

**Camera** : cette classe représente le point de départ de l’application. Celle-ci est responsable de créer la classe **Viewport** en utilisant la position de la caméra, sa direction, ainsi que la résolution souhaitée.

**Viewport** : cette classe est une réification du champ de vision de la caméra. Il s’agit d’un plan spatial sur lequel sont disposés des points correspondants à chaque pixels (de l’image finale). Cette classe fournit un itérateur pour accéder à l’ensemble des points de la grille.

**IteratorViewport** : itérateur qui permet d’obtenir les points du viewport.

**RayManufacture**: Cette classe permet d’instancier les rayons qui sont lancés dans la scène. Elle utilise la position de la caméra. Sa méthode d’instanciation prend un point, ce qui lui permet d’instancier un objet de type Ray.

**Ray** : Cette classe est une réification d’une demi-droite qui intersecte les géométries de la scène, afin de déterminer de déterminer le chemin emprunté par la lumière vers la caméra

**Picture** : Cette classe représente l’image qui sera générée par l’algorithme de ray-tracing. Elle est responsable de lancer les rayons à travers la scène en utilisant sa manufacture de rayon. Elle dispose aussi d’une méthode pour s’exporter vers l’écran ou encore vers un fichier BitMap.

**Bmp** : Cette classe représente un flux de sortie vers un fichier BMP, elle implémente l’interface *IOut*.

**Screen** : Cette classe représente un flux de sortie vers l’écran, permettant d’avoir un aperçu temporaire de l’image générée par le ray tracer. Elle implémente aussi l’interface *IOut*.

**Scene** : Cette classe permet de coordonner les acteurs de la scène et le lancer de rayon lui-même. Elle est également composée d’une chaine de responsabilité qui permet d’effectuer le traitement de chaque lancer de rayon, et lorsque le traitement est terminé, elle retourne à l’image la couleur calculée pour un rayon. Cette classe implémente l’interface *IScene*

**FormCube** : Cette classe représente un cube dans la scène. Elle implémente l’interface *IForm*.

**FormSphere** : Cette classe représente une sphère dans la scène. Elle implémente aussi l’interface *IForm*.

**HitResult** : Cette classe est une classe de données qui contient les informations nécessaires au traitement et calcul de la lumière pour un rayon donné. Ses attributs sont des propriétés publiques.

### Classes (suite)

**BaseProcessor** : C’est une classe abstraite dont le rôle est de rassembler le code commun aux classes dérivées concrètes. Les classes dérivées sont utilisés dans une chaine de responsabilité permettant le traitement de chaque rayon lancé grâce au *HitResult*.

**MissProcessor** : cette classe, qui dérive de BaseProcessor, permet de traiter les rayons qui n’intersectent aucune géométrie dans le champ de vision de la caméra.

**IlluminationProcessor** : cette classe dérive de BaseProcessor, et permet de calculer l’illumination au point d’impact d’un rayon.

**ShadowingProcessor** : classe dérivée de BaseProcessor, dont le but est de déterminer si un point est dans l’ombre ou directement éclairé.

**ReflectionProcessor** : classe dérivée de BaseProcessor, et qui se charge de traiter les surfaces qui sont réfléchissantes. Elle se sert également de *l’IlluminationProcessor* et du *ShadowingProcessor* pour déterminer l’illumination qui doit être rapportée après une réflexion.

**OcclusionProcessor** : classe dérivée de BaseProcessor, qui permet de traiter l’occlusion de la lumière en se basant sur le nombre de rayons occlus par l’environnement.

**AdaptiveOcclusionProcessor** : classe similaire à OcclusionProcessor qui traite l’occlusion de la lumière en se basant sur la distance des rayons occlus par l’environnement ainsi que sur leur pondération. Elle permet d’obtenir un dégradé dans le rendu de l’occlusion.

**SimpleRefractionProcessor** : classe dérivée de BaseProcessor, qui permet de calculer l’illumination rendue au travers des géométries transparentes, en tenant compte de leur indice de réfraction.

**AdvancedRefractionProcessor** : classe similaire à SimpleRefractionProcessor, mais qui est capable de gérer la réflexion totale à l’intérieur de la géométrie transparente.

**ManufactureHeightMultiProc** : classe qui implémente l’interface *IMultiProcManufacture*, permettant de créer une famille d’objets pour faire le traitement des rayons en parallèle.

**CustomProcess** : classe qui représente le traitement à effectuer à l’intérieur d’un processus lors du traitement parallèle.

**HeightParallelizer** : classe qui permet de découper une tâche, en plusieurs processus pour le traitement parallèle.

**HeightReassembler** : classe qui permet la reconstitution d’une image divisée sur la hauteur.

**Cleaner** : classe qui permet de faire le ménage après le traitement parallèle.

## Patrons utilisés :

Dans ce projet, des patrons de conceptions seront utilisés pour faciliter le développement de l’application, l’ajout de fonctionnalités, ou encore apporter de la flexibilité.

Le premier patron qui ressort, est la **manufacture**.Présent à plusieurs niveaux, la manufacture permet de séparer l’instanciation et l’utilisation de l’objet. Dans ce projet j’utiliserai la manufacture pour créer les rayons ou la scène par exemple.

Le second patron qu’on retrouve, est **l’itérateur**. Dans ce projet, l’itérateur sera utilisé pour obtenir les points du *Viewport*. De ce fait, le reste de l’application n’a pas à savoir comment est organisé la structure du *Viewport*.

On retrouve également la **manufacture abstraite**, lorsqu’il s’agit de créer le multi-processing. En considérant la manière dont le multiprocessing est implémenté, c’est-à-dire, découper le traitement total du viewport en plusieurs morceaux, créer des fichiers indépendants et réassembler les fichiers par la suite, la manière de découper la tâche et le réassemblage sont liés. C’est là que la manufacture abstraite est pertinente car elle permet d’instancier une famille d’objets, des objets qui vont ensemble. Ainsi je l’utilise pour instancier les classes qui traitent le multi-processing en se basant sur la division en hauteur de l’image.

Enfin, le troisième patron utilisé est la **chaine de responsabilité.** Elle intervient au niveau du traitement des *HitResult*, une fois que le rayon a été lancé dans la scène. Il nous permet d’effectuer certains traitements sous certaines conditions et selon un certain ordre, sans que le code qui utilise la chaine ne le sache (la scène en occurrence). De plus, il nous la chaine nous permet d’arrêter le traitement à n’importe quel nœud de la chaine, ce qui en fait un patron idéal pour optimiser le nombre de calculs à effectuer. Ce patron occupe une place importante car il permet de modifier ou d’ajouter des traitements (grâce à l’abstraction), sans devoir modifier le code existant, ce qui est très profitable pour le processus agile.