

Projet IN204
Projet informatique Lancé de rayons

FRANCO TEDESCO Luigi VILELA MOMENTE Gustavo

17 mai 2014

Table des matières

1	Introduction	3
1.1	Notre Démarche	3
1.2	Bref Explication	3
1.3	Possibilité d'étendre le projet	3
1.4	Améliorations proposés à notre projet	4
2	Analyse du Logiciel	5
2.1	Analyse d'usage	5
2.2	Analyse Fonctionnelle	5
2.3	Description précis du modèle	6
2.4	Architecture de Gros Grains	6
2.5	Design Patterns	8
2.6	Directives de préprocesseur	8
2.7	Compilation	9
2.8	Fichier d'entrée	10
2.9	Exemples	10
2.9.1	Format.txt	10
2.9.2	input.txt	13
2.9.3	spec_sphere.txt	13
2.9.4	mirros.txt	14
3	Conclusion	15
4	Annexes	16
4.1	Informations détaillés sur les classes	16

Table des figures

2.1	Analyse d'usage	5
2.2	Diagramme de classes	7
2.3	Schéma d'architecture des Gros Grains	8
2.4	Format sans artefacts	11
2.5	Format avec artefacts dû au parallélisme	12
2.6	Rendu avec input.txt	13
2.7	Rendu avec spec_sphere.txt	13
2.8	Rendu avec mirrors.txt	14

Chapitre 1

Introduction

1.1 Notre Démarche

Notre démarche était basée sur deux points principaux : la première était la création d'un modèle réaliste, au moins le plus proche possible des lois physiques, cependant avec des simplifications "discrètes" pour qu'il puisse être implémenté dans l'ordinateur. D'un autre côté, nous voulions aussi un moteur, pour le lancer, simple et qui avait support à une approche parallèle.

Pour faire face à ces deux contraintes imposées, nous avons établi qu'il serait les objets de la scène qui auraient des méthodes pour traiter les rayons reçus, ce qui a permis une approche simple de l'algorithme du lanceur et une simulation plus réaliste de la physique qui entoure le problème, les propriétés physiques de chaque objet définissent la couleur que l'on voit.

Nous avons étendu le projet à deux propositions suggérées : Support de la réflexion et de la transparence et une stratégie de parallélisation.

1.2 Bref Explication

Tout d'abord un entrée type texte décrit des objets de la scène qui doivent être créés par notre application - le format de cet *.txt* est expliqué dans le **format.txt**. Cette création de objets de la scène est gérée par la méthode *ReadScene* de la classe *Scene*, classe responsable pour la gestion de tout ce qui se passe dans la scène. Ce méthode rend comme résultat un objet *Scene* avec les objets comprises dans des containers (vecteurs) pour chaque type. D'après cette partie de création, le moteur reçoit l'objet *Scene* et la méthode *Tracer* est appelée. Ce méthode est le responsable pour la création, rendu et sauvegarde de l'image finale calculé après l'attribution d'un rayon de base pour chaque pixel de l'image, ce rayon qui sera suivi pour la méthode récursive *Follow* et, après interactions avec la scène, donne la couleur du pixel.

1.3 Possibilité d'étendre le projet

Notre projet, depuis le principe, était conçu et développé pour des modifications et expansions de notre modèle. Les interfaces d'objets étaient bien définies et cohérentes entre eux, ce qui permet l'ajout de nouveaux éléments facilement et garantit le bon fonctionnement du programme après les modifications. Cette notion de solution évolutive

étaient utilisé dans le développement du projet lui même - Création d'objets physiques simples (sphère et plane miroir), a priori, puis conception et implémentation d'objets plus complexe avec réfraction et réflexion diffuse que n'avaient pas intervenu dans le fonctionnement de les anciens objets simples.

1.4 Améliorations proposés à notre projet

- Implémenter vraiment du parallélisme
- Un méthode de vérification de la scène - Dans l'application développé pour nous, elle demande le bon sens de l'utilisateur par rapport à définition de la scène.
- Une interface pour aider dans la création de la scène.

Chapitre 2

Analyse du Logiciel

2.1 Analyse d'usage

Notre choix d'usage du logiciel était fait d'une manière très simple, premièrement on voulait un application simple pour l'utilisateur étant donné que le but principal est la rendu rapide de l'image et non l'aide au positionnement de les objets dans la scène, ce qui pourrait être facilement implémenté, et, en autre un syntaxe d'appel dans le terminal pareil à des autres programmes : “*RayTracer.exe input.txt output.jpg*”.

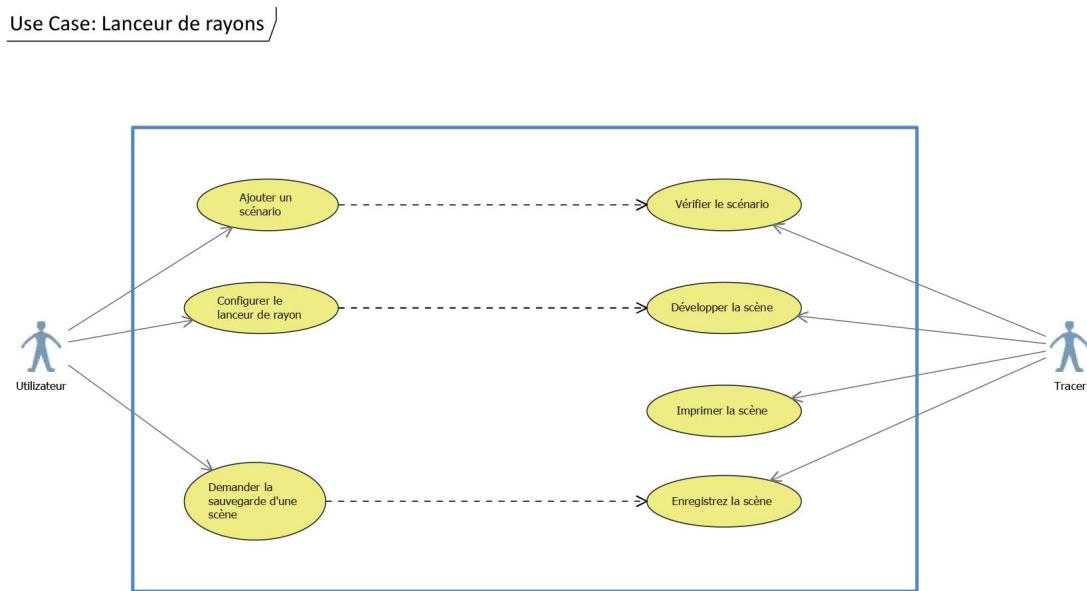


FIGURE 2.1 – Analyse d'usage

2.2 Analyse Fonctionnelle

Le tableau ci-dessous montre une description des composants leurs interactions et fonctions.

Composant	Description	Interactions	Fonction
Ray_tracer	Étant donné une scène quelconque, cette classe produit l'image rendu vers l'information avec la collision entre rayons sortant du observateur et les objets et lumières de la scène	<ul style="list-style-type: none"> - <i>Scene</i> - <i>Ray</i> - <i>CollisionObject</i> 	Le moteur de calcul
CollisionObject	Paquet d'information sur la position et occurrence de collision, donné par l'interaction entre un <i>Ray</i> et un <i>PhysicalObject</i>	<ul style="list-style-type: none"> - <i>light</i> - <i>IPhysicalObject</i> 	Interaction entre <i>Ray</i> et les objets de la scène
Scene	Elle lit les fichier d'entrée et édite la scène. En plus, elle contient les objets de la scène	<ul style="list-style-type: none"> - Fichier d'entrée - <i>Screen</i> - <i>Observer</i> - <i>light</i> 	Gestionnaire des composants de la scène.
IPhysicalObject	Objets qui possèdent des propriétés réflexives, réfractaires et absorbantes	<ul style="list-style-type: none"> - <i>CollisionObject</i> 	Objets de la scène
Observer	Définit l'origine et la direction du rayons de base	-	Point de vue du observateur
Screen	Définit la résolution de l'image finale en pixels et le taille physique de l'écran. L'écran et l'observateur, ensemble, forment le champs de vision capturé par l'image.	-	Représente l'écran
light	Objets luminescentes ponctuels qui possèdent une intensité	<ul style="list-style-type: none"> - <i>CollisionObject</i> 	Le sources lumineux de la scène

2.3 Description précis du modèle

Nous avons défini une grosse arbre d'héritage qui permet non seulement une l'application de contraintes pour l'extension du projet mais aussi la communication entre objets. Cet arbre est représentée dans la Fig. (2.2). En plus, les informations détaillés sur les classes se trouvent dans les annexes (Sec. (4.1))

2.4 Architecture de Gros Grains

D'après l'arbre d'héritage, on peut définir le schéma de l'architecture de Gros Grains que représente les objets et leurs interactions dans le programme. Cette architecture est dans la Fig. (2.3)

On peut voir de manière évidant le division entre deux organismes distinctes du programme, la partie de gestion et création de la scène et de gestion de l'image rendu.

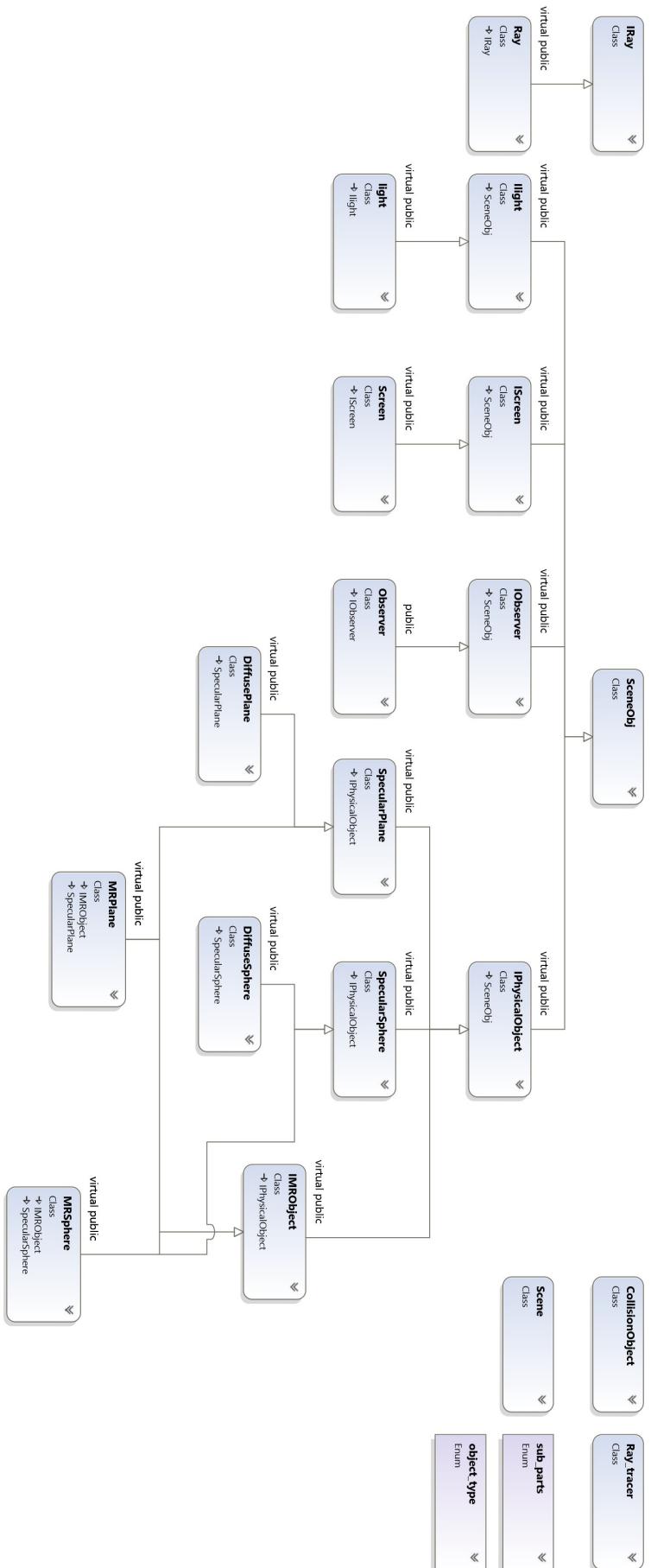


FIGURE 2.2 – Diagramme de classes

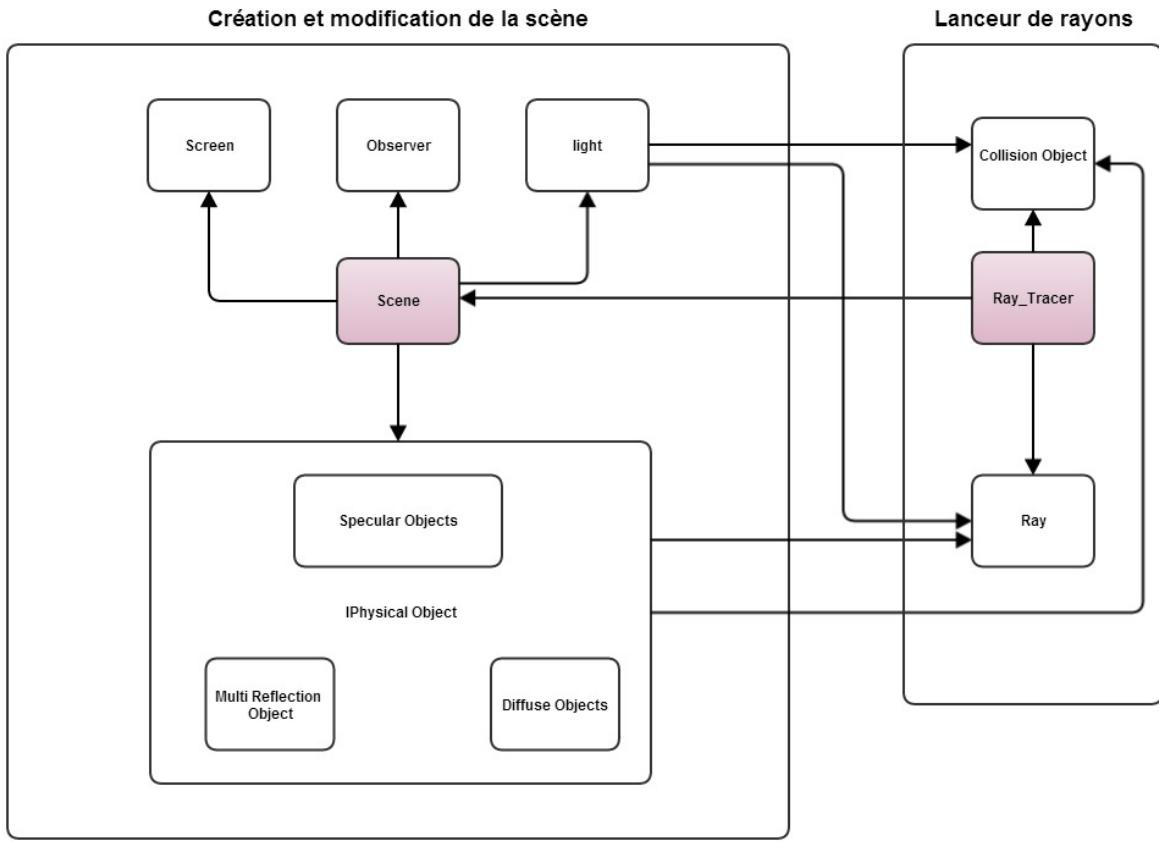


FIGURE 2.3 – Schéma d’architecture des Gros Grains

2.5 Design Patterns

Dans le modèle proposé pour le moteur, l’échange des rayons entre les fonctions est intense. Et donc, avec la finalité de réduire le nombre superflu d’opérations mémoire, c’est-à-dire la copie des données concernant les rayons, la solution usuelle est échanger des pointeurs vers les données. Et donc, dans la première version du moteur on a utilisé ça, mais après quelques essais on a observé que le programme était trop gourmand de mémoire. Tout ça à cause de plusieurs fuites de mémoire, une fois que le contrôler le cycle de vie d’un rayon n’est pas évident.

Ainsi, grâce à le modèle de programmation diffusé par la bibliothèque *Boost* : **smart pointers** on a survécu les problèmes de fuite de mémoire. Il faut dire, que l’on n’ai pas utilisé la version qui était déployé par *Boost* mais la variante introduit par le standard **C++11** avec les *shared_ptr*.

En plus, le diagramme de classes (Fig. (2.2)) met en évidence le fait que l’on a trouvé le notoire *Diamond Problem* dans notre structure de classes. Alors, on a utilisé les classes avec **méthodes virtuelles** avec la finalité de surpasser ce problème là.

2.6 Directives de préprocesseur

On souhaitait que le logiciel après compilation était le plus simple et rapide possible. Et donc, on a recouru à des directive de préprocesseur, c'est-à-dire les *#defines* qui habilitent

ou non quelques boucles `#ifdef`, pour éviter le comportement des processeurs superscalaires agressifs qui pourraient évaluer quelques boucles même quand ils ne sont pas prises, ainsi éviter cette possible perte de temps.

Finalement, toutes les directives sont dans le fichier `defines.h` et leur fonctionnement est dans le Tableau 2.1.

Directive	Description
REFLEX_COUNTER	Définit le nombre d'interactions que un rayon peut faire avant mourir
DEBUG	Sert à activer l'utilisation du module <i>Visual Leak Detector</i> enfin de chercher les fuites de mémoire, si on veut l'utilise il faut penser à l'inclusion de la bibliothèque dans la chaîne de compilation
PARALLEL ¹	Sert à activer l'utilisation des multiprocesseurs
TWOWAY	Si il est défini, des qu'une réflexion diffuse arrive au lieu de rendre un seul rayon deux seront retournés. Les deux auront la moitié de l'intensité normale, mais un d'entre eux considérera les objets réfractaires comme transparents et il ne réfractera pas. Si il n'est pas défini, comme l'algorithme qui vérifie les collisions dans le chemin vers une lumière considère les objets réfractaires comme transparents, une seul rayon sera rendre avec direction vers la lumière
N1	Définit l'indice de réfraction du milieu
SHOW_OUTPUT	Si il est défini une fenêtre apparaîtra avec le rendu de la scène des qu'elle est prêt

TABLE 2.1 – Directives de préprocesseur

2.7 Compilation

D'abord, le projet a été développé complètement sur **Visual Studio 2012**, et donc, le fichier `.sln` concernant sera rendu avec les code-sources, mais pour l'utilise il faut que quelques bibliothèques soient présents et que quelques variables du système(v.d.s.) soient définit. Elles sont :

Eigen 3.1.4

Une bibliothèque pour l'algèbre linéaire. Elle offre en plus des containers que l'on a utilisé pour plusieurs échanges de données.

Dans le projet il faut que la v.d.s.**EIGEN_PATH** soit définit avec le chemin vers le fichier contenant la bibliothèque.

OpenCV 2.4.8

Une bibliothèque pour l'affichage et manipulation des fichiers images. Dans le projet il faut que la v.d.s.**OPENCV_DIR** soit définit avec le chemin vers le fichier `\opencv\build` présent dans le fichier téléchargé.

1. Voir Sec. (2.9.1).

En plus, pour le parallélisme on a utilisé le *OpenMP* et le compilateur doit supporter ça. Finalement, comme on a dit dans la Sec. (2.5), on a utilisé *shared_ptr* donc, un compilateur dans le standard **C++11** est nécessaire.

Enfin, pour utiliser les exécutables fournis il est nécessaire le Visual C++ Redistributable for Visual Studio 2012.

2.8 Fichier d'entrée

Dans le paquet du projet est compris un fichier appelé *Format.txt* avec la description de comment écrire un fichier d'entrée.

2.9 Exemples

Dans le paquet du projet est compris les fichiers qui génèrent les exemples suivants.

2.9.1 Format.txt

Comme on avait dit dans les Sec(1.4,2.6) l'implémentation du parallélisme ne marche pas toujours. Par exemple, la Fig. (2.4) a été rendu sans la directive d'activation du parallélisme et la Fig. (2.5) avec. On voit que des artefact apparaît de façon “aléatoire” quand le parallélisme est activé. Mais ça n'arrive pas toujours, on a observé que le nombre de artefacts est à peu près inversement proportionnel à la complexité de la scène.

En plus, on a observé que le problème arrive plutôt quand on exécute le programme hors *Visual Studio*. Finalement, on pense que le problème est dû à des accès mémoire, mais on n'a pas trouvé la solution. À cause de ça on a laissé l'utilisation du parallélisme comme une option.

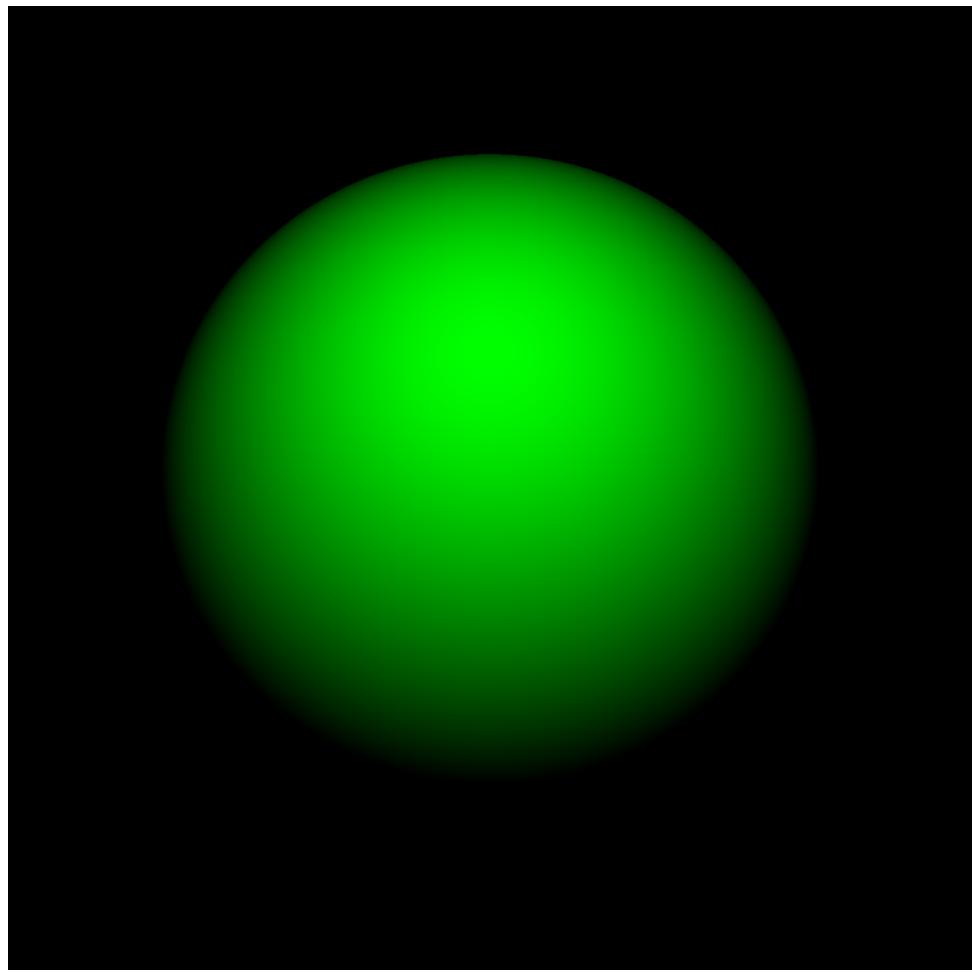


FIGURE 2.4 – Format sans artefacts

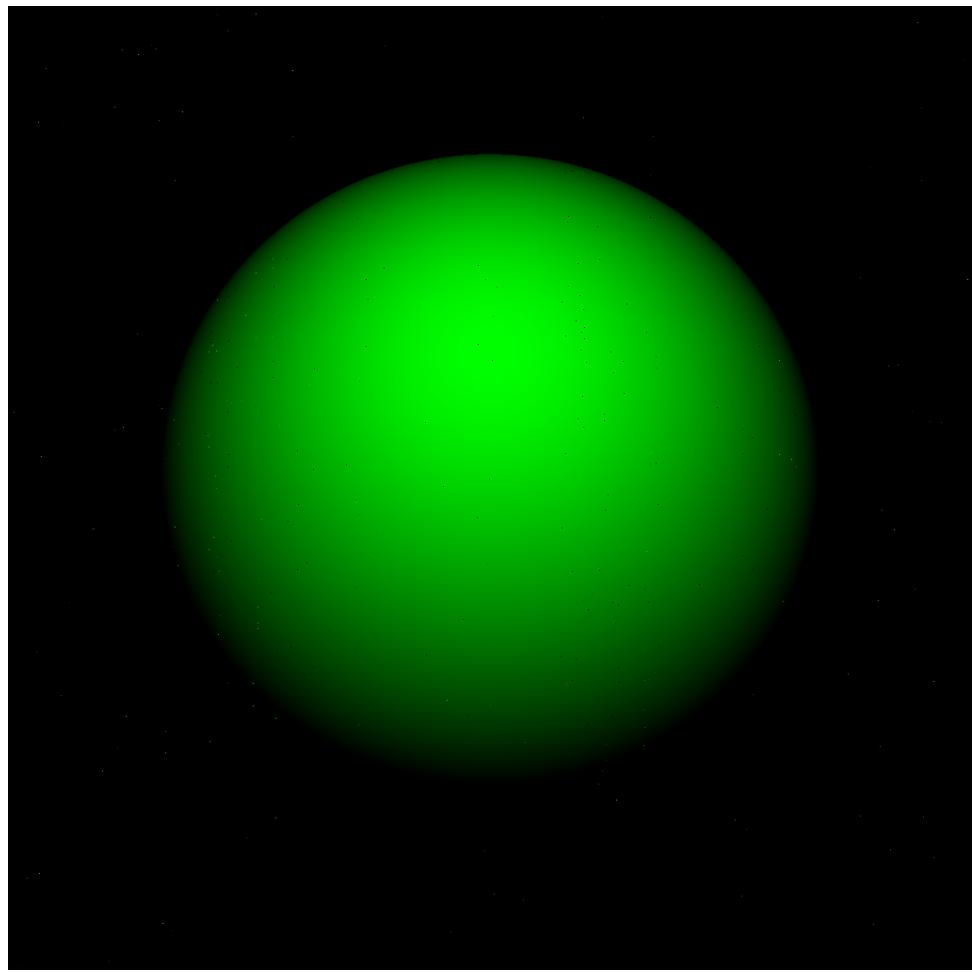


FIGURE 2.5 – Format avec artefacts dû au parallélisme

2.9.2 input.txt

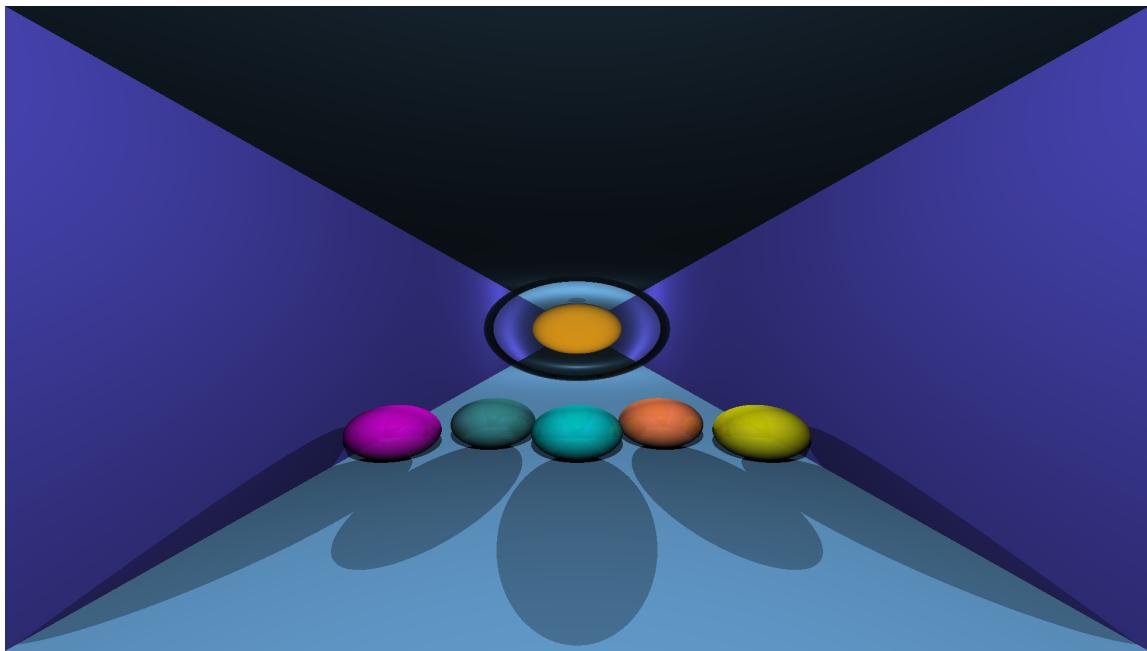


FIGURE 2.6 – Rendu avec input.txt

2.9.3 spec_sphere.txt

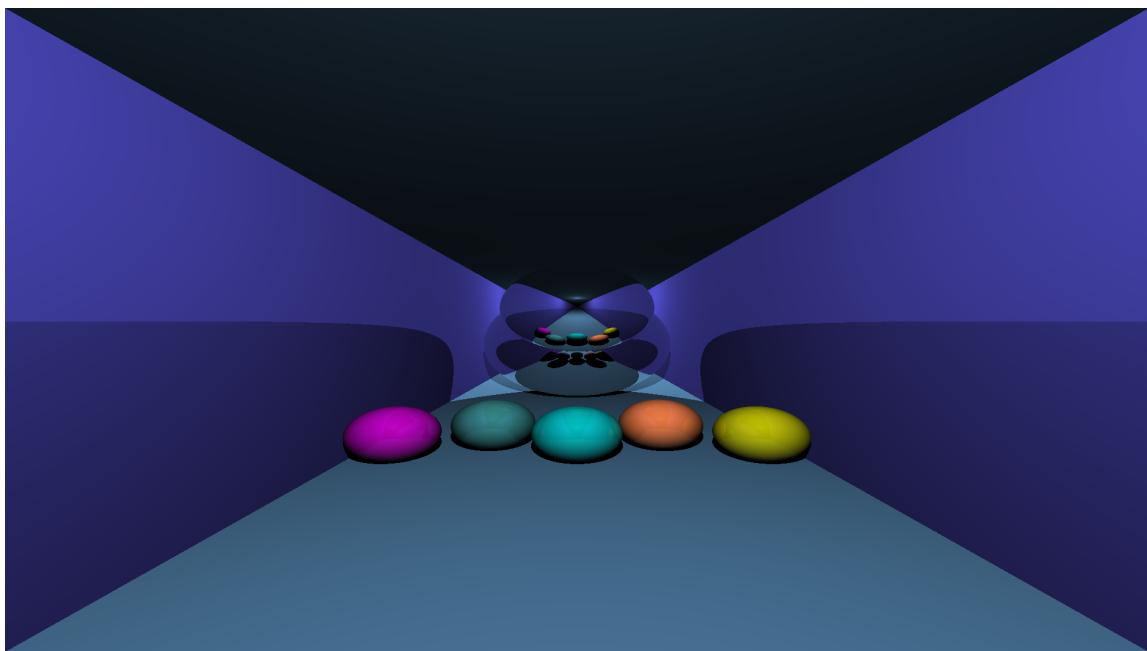


FIGURE 2.7 – Rendu avec spec_sphere.txt

2.9.4 mirros.txt

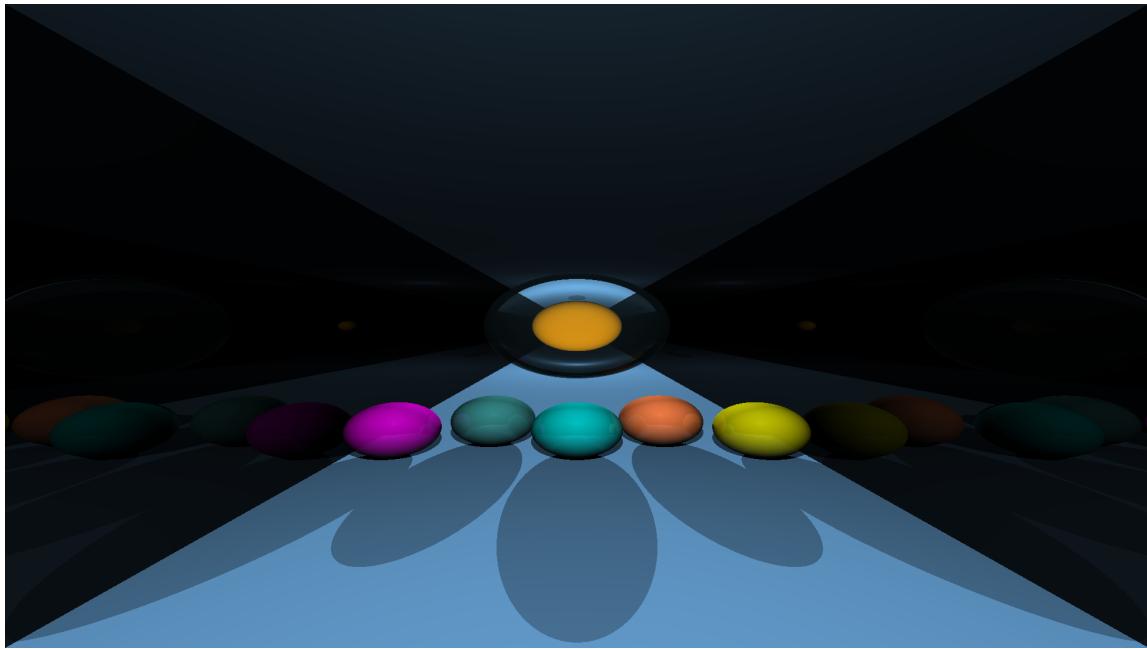


FIGURE 2.8 – Rendu avec mirrors.txt

Chapitre 3

Conclusion

Grâce à ce projet on peut mis en œuvre plusieurs concepts développés dans le cours IN204. En plus, on a profité de l'occasion pour mettre en place un répertoire *Git* et l'utiliser avec la finalité de faciliter le management du code des que les deux auteurs ont contribué au même temps.

Le répertoire de travail n'est pas public, mais la version finale peut être téléchargé dans https://github.com/gustavo-momente/Ray_Tracing ou avec un :

```
git clone https://github.com/gustavo-momente/Ray_Tracing.git
```

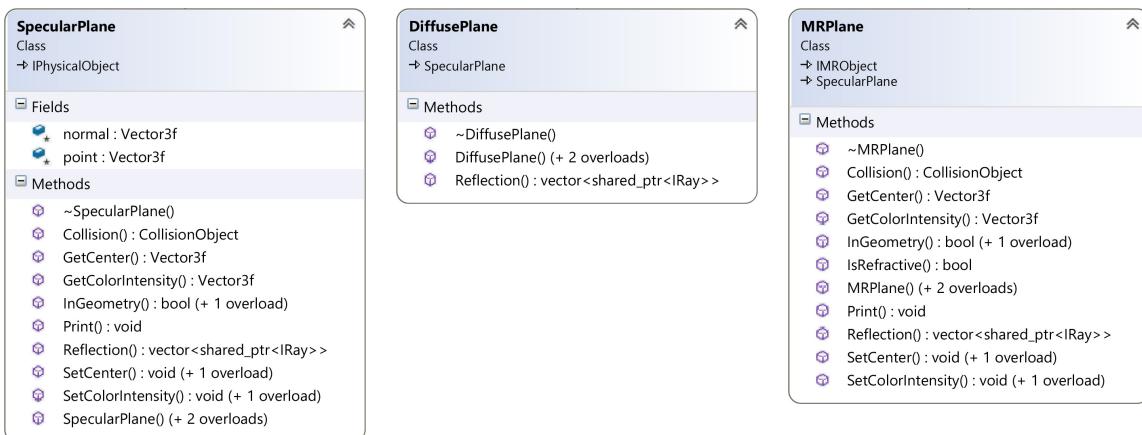
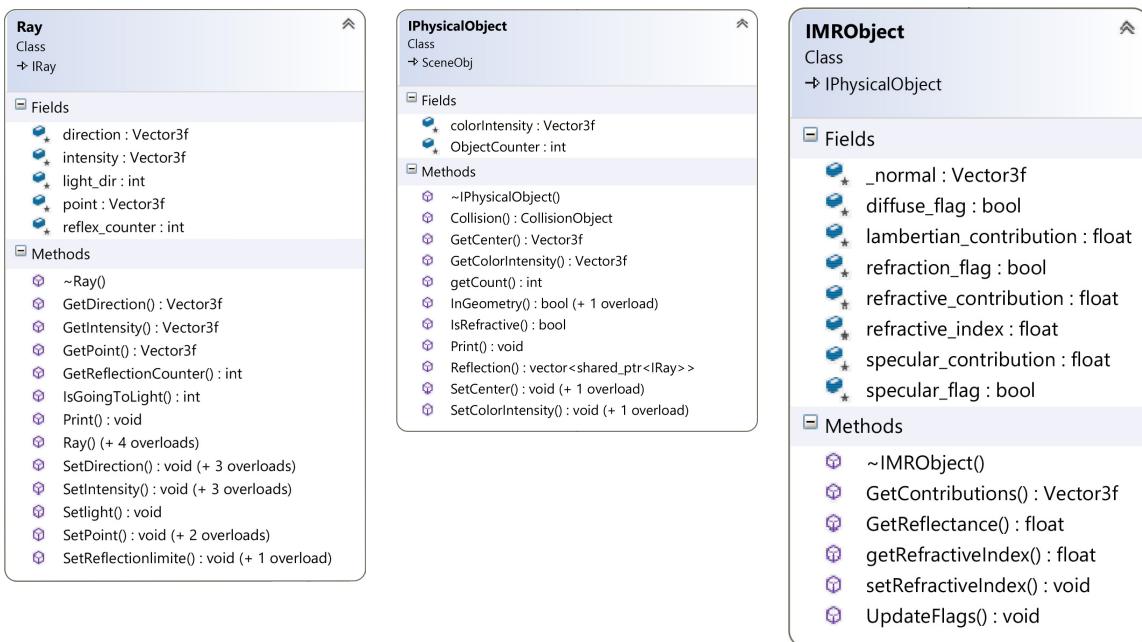
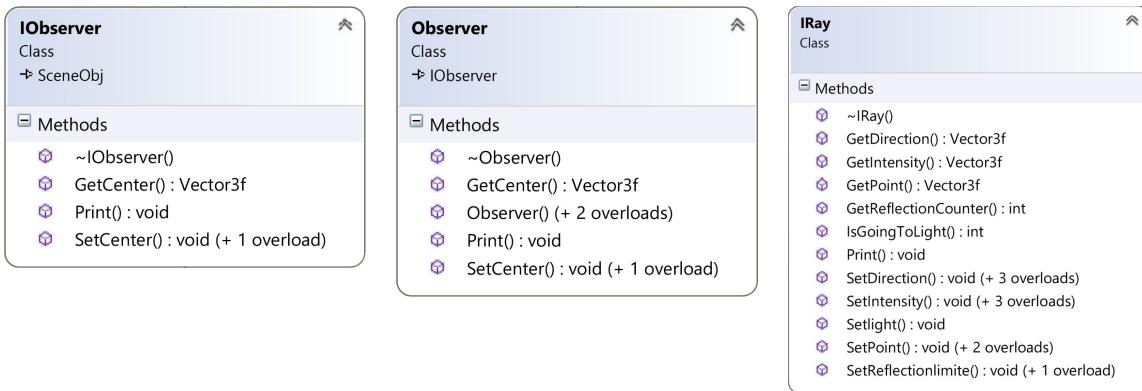
Chapitre 4

Annexes

4.1 Informations détaillées sur les classes

The screenshot displays six class details panels arranged in two rows of three:

- object_type**: An enum with values: comment2, specularPlane, specularRectangle, specularSphere, diffusePlane, diffuseRectangle, diffuseSphere, mrplane, and mrsphere.
- sub_parts**: An enum with values: comment1, screen, observer, Light, object, space, and empty.
- CollisionObject**: A class with fields DoltCollide (bool) and position (Vector3f), and methods ~CollisionObject(), CollisionObject() (+ 2 overloads), GetCollisionPosition() (Vector3f), IsThereCollision() (bool), and Print() (void).
- SceneObj**: A class derived from SceneObj, with fields center (Vector3f), and methods ~SceneObj(), GetCenter() (Vector3f), Print() (void), and SetCenter() (void (+ 1 overload)).
- Ilight**: A class with fields intensity (Vector3f), and methods ~Ilight(), Collision() (CollisionObject), GetCenter() (Vector3f), GetIntensity() (Vector3f), Print() (void), SetCenter() (void (+ 3 overloads)), and SetIntensity() (void (+ 3 overloads)).
- light**: A class derived from Ilight, with methods ~light(), Collision() (CollisionObject), GetCenter() (Vector3f), GetIntensity() (Vector3f), light() (+ 3 overloads), Print() (void), SetCenter() (void (+ 3 overloads)), and SetIntensity() (void (+ 3 overloads)).



SpecularSphere

Class
→ IPhysicalObject

Fields

- * center : Vector3f
- * radius : float
- * sqrd_radius : float

Methods

- ~SpecularSphere()
- Collision() : CollisionObject
- GetCenter() : Vector3f
- GetColorIntensity() : Vector3f
- InGeometry() : bool (+ 1 overload)
- Normal() : Vector3f
- Print() : void
- Reflection() : vector<shared_ptr<IRay>>
- SetCenter() : void (+ 1 overload)
- SetColorIntensity() : void (+ 1 overload)
- SpecularSphere() (+ 2 overloads)

DiffuseSphere

Class
→ SpecularSphere

Methods

- ~DiffuseSphere()
- DiffuseSphere() (+ 2 overloads)
- Reflection() : vector<shared_ptr<IRay>>

MRSphere

Class
→ IMROObject
→ SpecularSphere

Methods

- ~MRSphere()
- Collision() : CollisionObject
- GetCenter() : Vector3f
- GetColorIntensity() : Vector3f
- InGeometry() : bool (+ 1 overload)
- IsRefractive() : bool
- MRSphere() (+ 2 overloads)
- Print() : void
- Reflection() : vector<shared_ptr<IRay>>
- SetCenter() : void (+ 1 overload)
- SetColorIntensity() : void (+ 1 overload)

IScreen

Class
→ SceneObj

Methods

- ~IScreen()
- GetCenter() : Vector3f
- GetDiscretv1() : Vector3f
- GetDiscretv2() : Vector3f
- GetPixel() : Vector2i
- GetSize() : Vector2f
- GetStartCorner() : Vector3f
- Print() : void
- SetCenter() : void (+ 1 overload)
- SetPixel() : void
- SetSize() : void (+ 1 overload)
- UpCorner() : void

Screen

Class
→ IScreen

Fields

- * center : Vector3f
- * corner : Vector3f
- * discretv1 : Vector3f
- * discretv2 : Vector3f
- * pixelDim : Vector2i
- * v1 : Vector3f
- * v2 : Vector3f

Methods

- ~Screen()
- GetCenter() : Vector3f
- GetDiscretv1() : Vector3f
- GetDiscretv2() : Vector3f
- GetPixel() : Vector2i
- GetSize() : Vector2f
- GetStartCorner() : Vector3f
- Print() : void
- Screen() (+ 2 overloads)
- SetCenter() : void (+ 1 overload)
- SetPixel() : void
- SetSize() : void (+ 1 overload)
- UpCorner() : void
- UpDisc() : void

Scene

Class

Fields

- * light_vector : vector<light*>
- * obs_vector : vector<IObserver*>
- * physic_vector : vector<IPhysicalObject*>
- * screen_vector : vector<IScreen*>

Methods

- ~Scene()
- Collision() : bool
- GetLight() : vector<light*> (+ 1 overload)
- GetObserver() : vector<IObserver*> (+ 1 overload)
- GetPhysical() : vector<IPhysicalObject*> (+ 1 overload)
- GetScreen() : vector<IScreen*> (+ 1 overload)
- numbers() : int (+ 1 overload)
- Print() : void
- ReadData() : void
- ReadScene() : void
- Scene() (+ 1 overload)
- SetLight() : void
- SetObserver() : void
- SetPhysical() : void
- SetScreen() : void

Ray_tracer

Class

Fields

- * image_vector : vector<Mat*>
- * output_name : string
- * scene : iterator
- * scene_vector : vector<Scene*>

Methods

- ~Ray_tracer()
- BaseRay() : vector<shared_ptr<IRay>>
- CheckScene() : void
- FirstCollision() : bool
- Follow() : Vector3f
- GetOutput() : string
- Ray_tracer() (+ 4 overloads)
- SaveImage() : void
- SetOutput() : void
- SetScene() : void
- Test_singleRay() : void
- Threat_pixel() : Vec3b
- Tracer() : void