Projet C++ : Moteur d'optique géométrique 2D

Félix Faisant

Le code nécessite la bibliothèque SFML pour l'affichage (paquet `libsfml-dev` sur Mint ou Debian).

1. Objectifs et méthodes

Le but de ce projet est de simuler les lois de l'optique géométrique sur une scène 2D, et ainsi de construire un moteur d'illumination de scènes 2D. Le principe est simple :

→ Les **sources** envoient des rayons lumineux, par exemple dans toutes les directions pour une source isotrope, ou dans une seule direction et de façon étendue pour une source de type "soleil".

→ Les rayons sont interceptés par les **objets** de la scène. Suivant l'objet, le rayon peut être réfléchi, réfracté, diffusé, filtré, absorbé… L'objet ré-émet alors zéro, un ou plusieurs rayons.

→ Des **écrans** accumulent l'intensité des rayons qu'ils interceptent sur une matrice de pixels.

→ Le processus d'interception - ré-émission est répété jusqu'à ce que le rayon soit totalement absorbé.

On peut alors se servir des pixels des écrans comme image, comme un CCD de caméra (1D !) si on a placé une lentille devant, ou comme la luminosité qu'aurait un mur à cet endroit (par exemple pour illuminer une scène d'un jeu de plateforme). Plutôt qu'un long discours, mieux vaut regarder les exemples page 2.

On peut voir ça comme une méthode de Monte-Carlo. C'est très inefficace pour construire des images (on appelle ça alors le *pathtracing*) car une toute petite fraction des rayons émis arrivent au "CCD" (comme pour une caméra réelle)[1]. C'est par contre adapté pour faire une illumination physiquement réaliste (on appelle ça alors l'illumination par *radiosité*).

Les différents objets implémentés ici sont :

- Des lignes ou des arcs de cercles, dont les calculs d'interception sont détaillés dans `lois.pdf`.
- Des dioptres (lois de Snell-Descartes et coefficients de Fresnel, détaillées dans `lois.pdf`), des miroirs, des objets diffusants (diffusion lambertienne ou plus sophistiqué), des écrans, des filtres…
- Ou bien des objets non localisés, comme du brouillard.

Pour implémenter les couleurs, on choisit de rester proche des lois physiques en attribuant un **spectre** discret à chaque rayon (ici 4 longueurs d'onde). De plus, il faut prendre en compte la polarisation TE / TM, qui se comportent différemment pour la réflexion/réfraction[2]. Ainsi, chaque rayon possède *8 composantes*. De plus, on oublie tout phénomène de nature ondulatoire. On travaille alors directement en *intensité lumineuse*, plutôt qu'en amplitude, et on les additionne simplement. Cela simplifie largement l'implémentation.

2. Structure du code

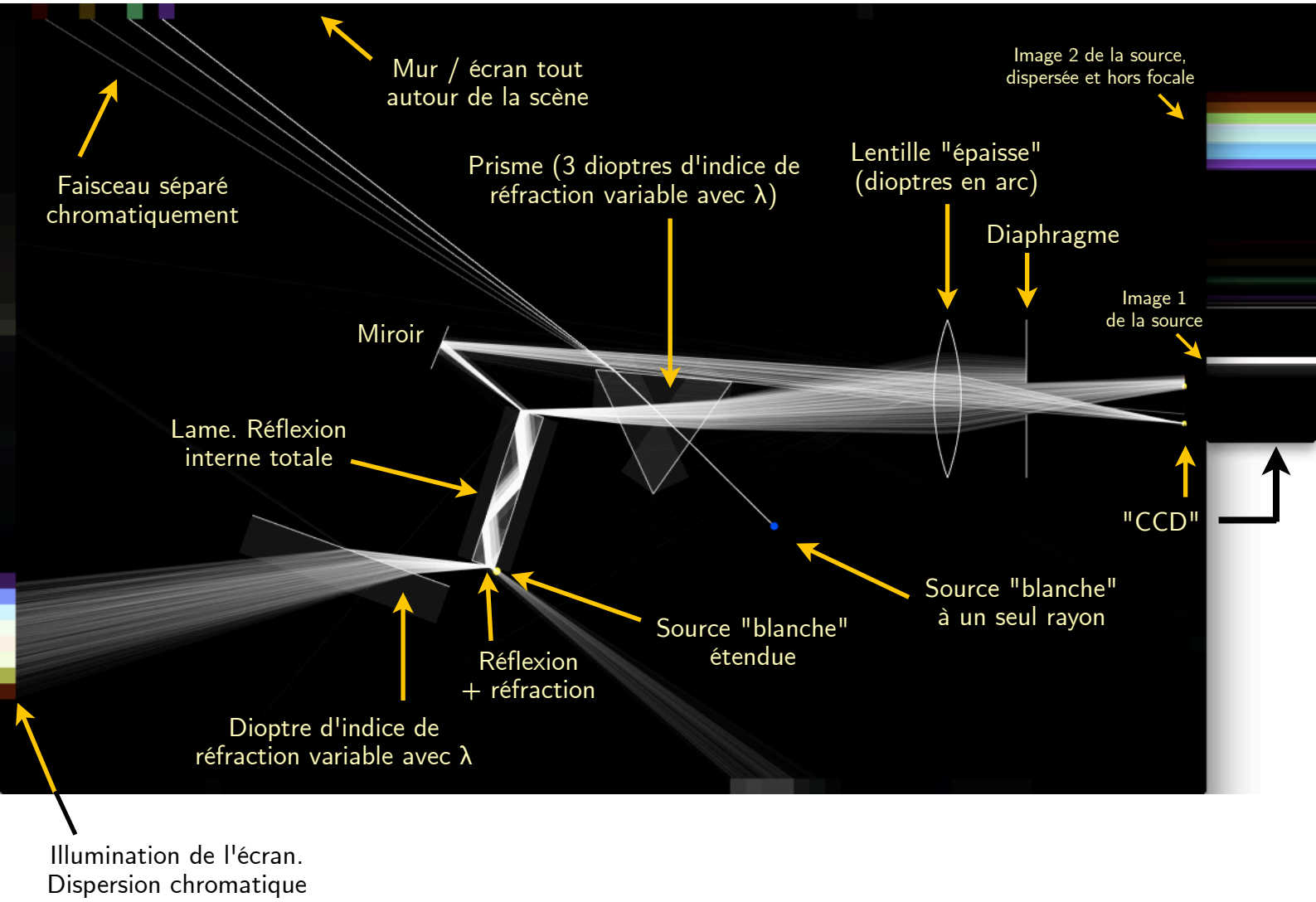
Les fichiers `Util.h`/`Util.cpp` implémentent des structures et fonctions utilitaires classiques : un vecteur 2D `vec_t` et un point 2D `point_t` avec les surcharges d'opérateurs usuelles, les rotations/translations/homothéties, une structure `angle_interv_t` qui implémente un intervalle angulaire dans $\mathbb{R}$ modulo 2π… Le fichier `sfml_c01.hpp` contient des fonctions utilitaires pour l'affichage avec SFML, notamment pour convertir les coordonnées $[0,1]\times[0,1]$ en coordonnées fenêtre.

1. Pour faire du rendu d'image réaliste performant, les rayons sont lancés depuis l'écran et sont propagés dans le sens inverse (on appelle ça alors le *raytracing*). C'est beaucoup moins simple à implémenter pour les processus irréversibles (diffusion…).

2. Le plan d'incidence des rayons est toujours le même : le plan où vivent les objets 2D. Ainsi, les polarisations TE et TM (relatives au plan d'incidence) sont toujours les mêmes, on peut donc séparer la lumière de façon globale en composantes TE et TM, et les traiter séparément.

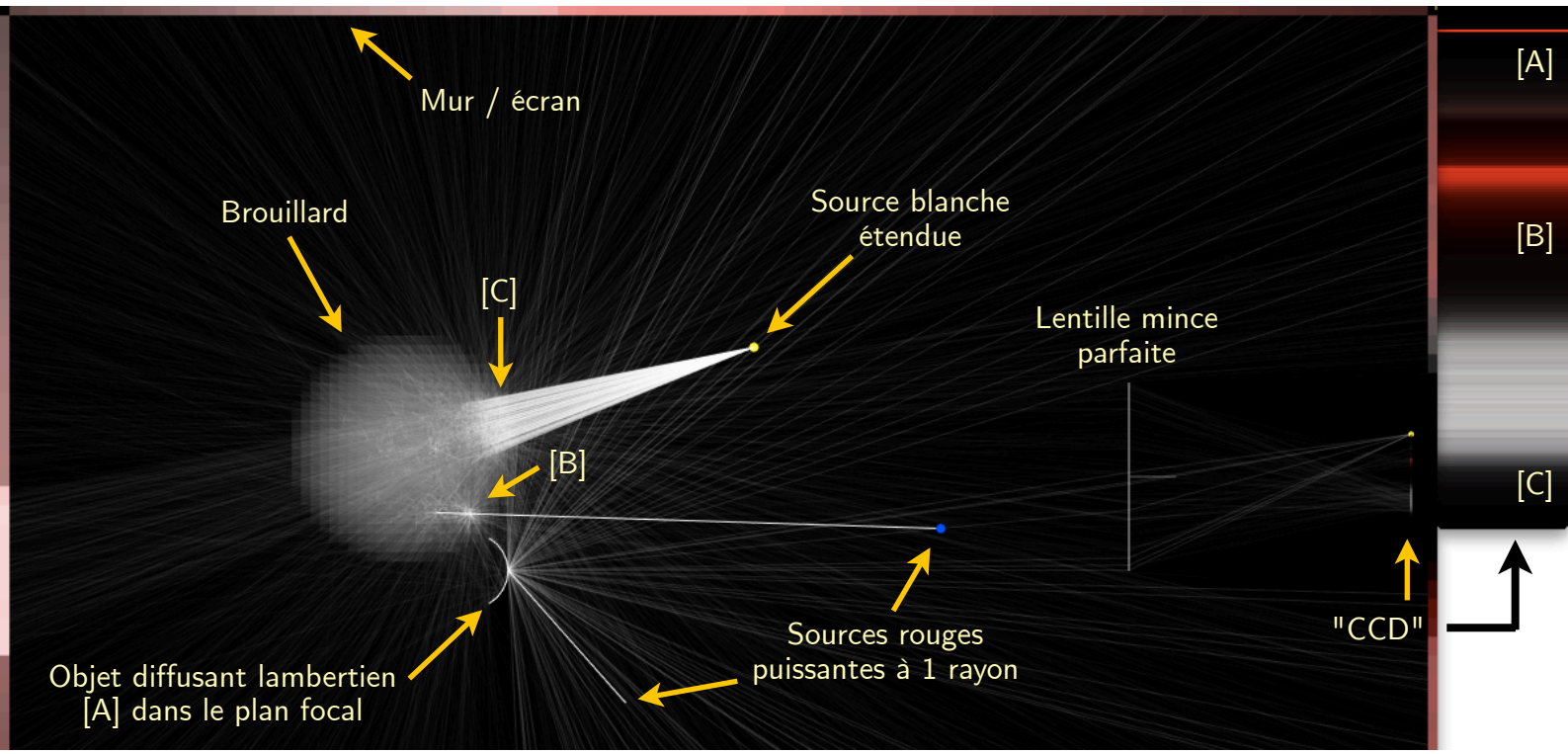
Scène "Réfraction et milieux"

Peut être obtenue avec `make milieux`



Scène "Brouillard"

Peut être obtenue avec `make brouillard`



La structure `Rayon` est implémentée dans `Rayon.h`/`Rayon.cpp`. Un rayon est défini par son point d'origine, sa direction (angle par rapport à l'horizontale), et son spectre de couleur/polarisation en intensité. La structure `Specte` contient les 8 composantes ainsi que des routines de manipulation des composantes.

2.1. Objets

Pour décrire le contenu de la scène, on utilise un arbre de classes (voir diagramme page 4), pensé pour être facilement extensible, éviter toute duplication de code, et utiliser le polymorphisme au maximum.

On distingue les sources, qui ne font qu'émettre des rayons, de tous les autres objets de la scène. Tous les types de sources dérivent de la classe virtuelle `Source`, et implémentent la méthode `genere_rayons` qui renvoie un `vector<Rayon>`. On pourra regarder la classe `Source_SecteurDisqueLambertien` (`Source.h:127`), qui implémente une source étendue lambertienne en forme de secteur.

Tous les autres objets de la scène dérivent de la classe virtuelle de base `Objet`, qui déclare l'interface commune utilisée lors de la propagation (interception puis ré-émission) des rayons, et pour l'affichage. Un objet est capable de **tester l'interception d'un rayon** avec la méthode `essai_intercept(rayon)`, et, le cas échéant, de **ré-émettre le rayon** avec `re_emit(rayon, intercept)` (où `intercept` est une structure opaque renvoyée par `essai_intercept`, conservant les informations de l'interception) :

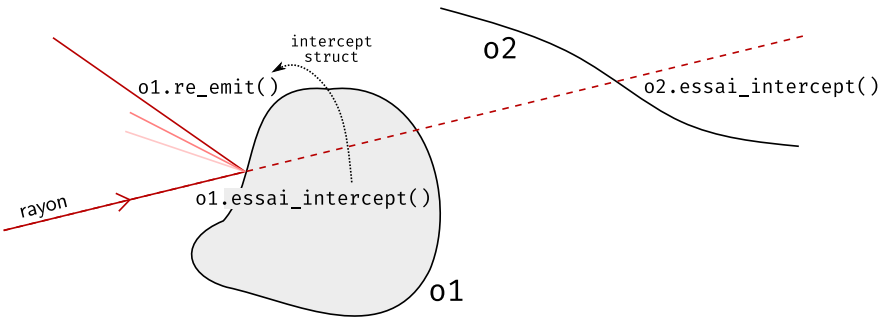


Figure 1. Scénario typique (celui de `Scene::interception_re_emission`) d'utilisation des objets.

Cela semble compliqué, mais l'interception et la ré-émission sont séparés pour des raisons de performance : les calculs de ré-émission peuvent être lourds (e.g. lois de Fresnel) et ne sont pas nécessaires à chaque interception (par exemple si le rayon a été intercepté plus en amont par un autre objet). `essai_intercept` renvoie aussi la distance entre le point d'émission du rayon et l'interception, qui sert à déterminer quel est l'objet le plus proche sur la trajectoire du rayon (voir plus loin pour l'utilisation de ces méthodes).

D'un côté, on définit alors des classes (virtuelles) qui s'occupent uniquement de la **géométrie** et de l'interception des rayons. `ObjetCourbe` décrit une courbe/interface bien définie, et la structure opaque renvoyée par `essai_intercept` contient le point d'incidence, l'angle d'incidence dans le repère de la scène, l'angle à la normale de la courbe, et le côté sur lequel le rayon est intercepté. L'interception est enfin implémentée pour deux géométries particulières : `ObjetLigne` décrit un segment, et `ObjetArc` décrit un arc de cercle. On pourra regarder le header `ObjetsCourbes.h` et les méthodes `ObjetArc::essai_intercept_courbe` (`ObjetsCourbes.cpp:100`) et `ObjetCourbe::essai_intercept` (`ObjetsCourbes.cpp:10`).

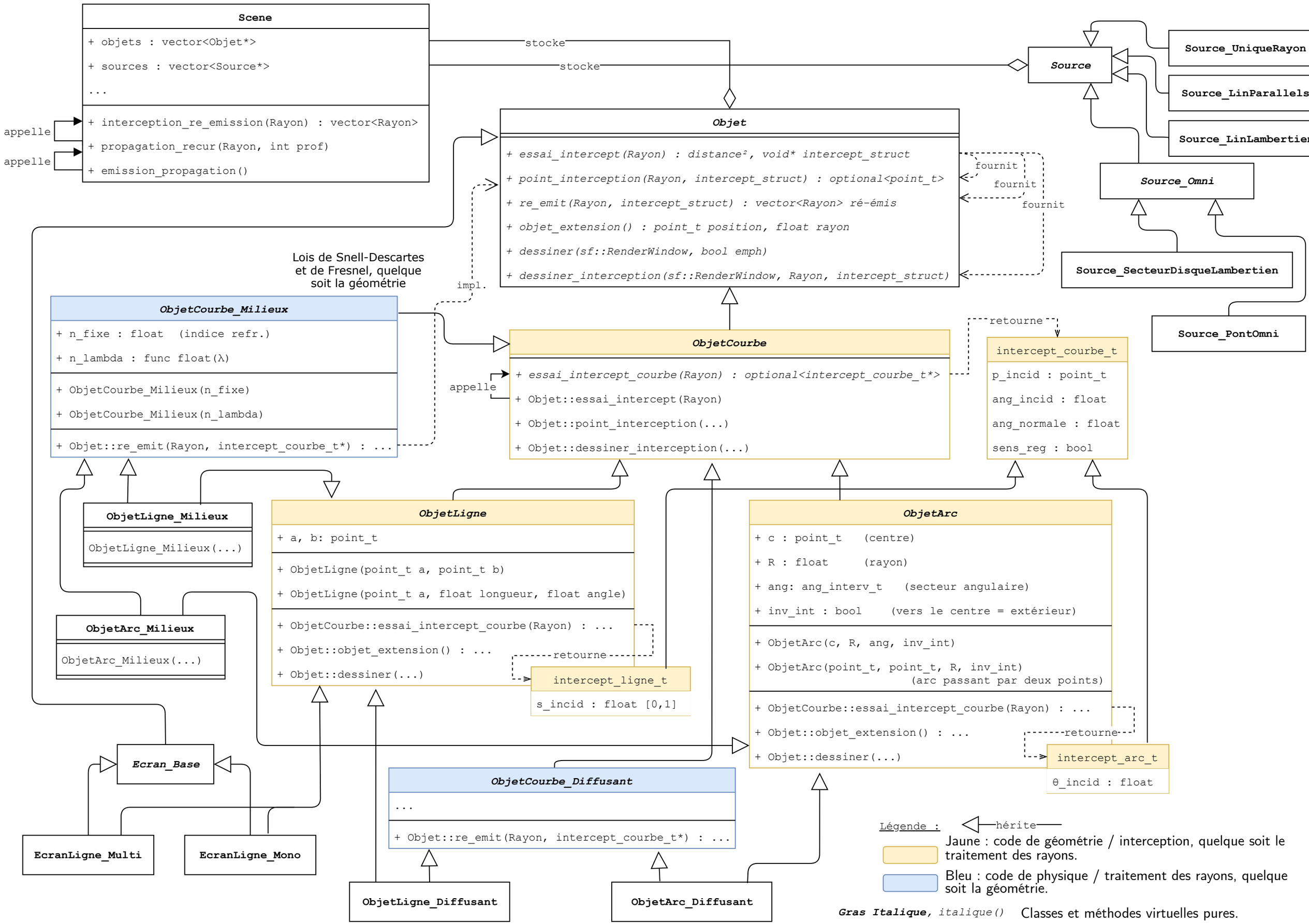
D'un autre côté, on implémente la **physique** et la **ré-émission** des rayons dans des classes virtuelles indépendantes de la géométrie : `ObjetCourbe_Milieux` pour la réflexion/réfraction sur un dioptre, `ObjetCourbe_Diffusant` pour une surface diffusante, `ObjetCourbe_Miroir` pour les miroirs... Ces classes implémentent, elles, la méthode `re_emit()`. On pourra regarder `ObjetCourbe_Milieux::re_emit` (`ObjetMilieux.cpp:20`), qui implémente les fameuses lois

$$\theta_\mathsf{r} = \theta_\mathsf{i}, \quad n_2 \sin(\theta_\mathsf{t}) = n_1 \sin(\theta_\mathsf{i}), \quad R_\mathsf{TE} = \left| \frac{n_1 \cos\theta_\mathsf{i} - n_2 \cos\theta_\mathsf{t}}{n_1 \cos\theta_\mathsf{i} + n_2 \cos\theta_\mathsf{t}} \right|^2, \quad R_\mathsf{TM} = \left| \frac{n_2 \cos\theta_\mathsf{i} - n_1 \cos\theta_\mathsf{t}}{n_2 \cos\theta_\mathsf{i} + n_1 \cos\theta_\mathsf{t}} \right|^2$$

Enfin, comme illustré sur le diagramme, on combine ces classes pour en faire des objets complètements définis : par exemple `ObjetLigne_Milieux`, qui est un dioptre plan, hérite des classes `ObjetCourbe_Milieux` et `ObjetLigne`.

Diagramme des classes partiel

On prend pour exemple les classes `ObjetLigne_Milieux` *et* `ObjetArc_Milieux`*, et seuls elles et les classes parentes sont totalement détaillées. Le reste du code est résumé ou omis.*



Les écrans (définis dans `Ecran.h`) accumulent les rayons incidents et calculent une intensité moyenne pour afficher ou enregistrer le résultat. On utilise surtout `EcranLigne_Multi`, qui est une matrice (1D) de pixels et qui hérite de `ObjetLigne`.

2.2. Scène

La logique de la propagation des rayons est implémentée dans la classe `Scene` (`Scene.h`/`Scene.cpp`), qui stocke les sources et les objets de façon polymorphique (`std::vector<std::shared_ptr<Objet>>`). Pour illuminer / former une image, on appelle la méthode `emission_propagation()` qui lance les rayons à partir des sources. Pour chaque rayon, la méthode `propagation_recur(Rayon)` est appelée. Cette méthode effectue les tests d'interception et la ré-émission de façon récursive. Schématiquement, on a

```
void Scene::propagation_recur (rayon) {
    rayons_re_emis = interception_re_emission(rayon);
    for (ray : rayons_re_emis)
        propagation_recur(ray);
}
std::vector<Rayon> Scene::interception_re_emission (ray) {
    Teste objet->essai_intercept(ray) sur tous les objets de la scène et sélectionne le premier
      objet sur le chemin du rayon.
    objet_le_plus_proche->re_emit(ray, intercept_struct);
}
```

(cf. figure page 3). Les rayons sont ainsi propagés sur la scène jusqu'à ce qu'ils soient absorbés par un écran ou qu'ils soient d'intensité négligeable (`Scene::intens_cutoff`). Une limite est fixée pour éviter les récursion infinies (par exemple avec une réflexion interne totale dans une lame).

Enfin, la méthode `dessiner_scene()` est appelée pour afficher tous les objets de la scène dans la fenêtre. Tant que rien ne bouge dans la scène, on répète ce processus (une *frame*) autant que possible pour avoir un résultat de moins en moins bruité sur les écrans (merci au théorème central limite).

2.3. Divers

La classe `SceneTest` étend `Scene` et regroupe tout le code indépendant du reste et qu'on trouverait typiquement dans les `main()`, mais pas assez générique pour être dans la classe `Scène` : création des fenêtres SFML, de quelques objets courants, d'une lentille et d'un écran pour former une image, ...

Le code est fourni avec 4 scènes démonstratives utilisant `SceneTest` : `main_milieux.cpp` (qu'on pourra regarder), `main_brouillard.cpp`, `main_store.cpp`, `main_diffus_test.cpp`, et qu'on peut faire tourner avec `make milieux`, `make brouillard`, `make store`, et `make diffus_test` respectivement.

Quelques remarques finales :

- Tous les attributs de classes qui ne sont pas associés à des contraintes/invariants et qui sont susceptibles d'être utiles à l'utilisateur de la classe sont publiques. L'encapsulation, ce n'est pas embêter l'utilisateur avec des getters, c'est protéger l'utilisateur contre des bêtises.
- L'extensibilité et la clarté a été préférée à la performance quand le choix se posait.
- Lorsque les constructeurs par copie et d'affectation ne sont pas déclarés, c'est qu'on laisse le compilateur générer celui par défaut, ou, lorsque marqué *deleted* dans une classe parent, aucun.

3. Conclusion

On reproduit tout à fait de nombreux effets d'optique géométrique que l'on connait bien : lentilles épaisses pas tout à fait stigmatiques et avec des aberrations chromatiques, fibres optiques...

On obtient une illumination réaliste (toutefois gourmande en CPU) d'une scène 2D. La formation d'images 1D (par exemple, à droite, un surface bleue diffusante avec reflet de la source ponctuelle l'éclairant) sont correctes mais un peu ennuyeuses. Cela donne envie d'étendre le programme à une scène 3D (ce qui ne serait pas difficile) pour obtenir de vraies images 2D.