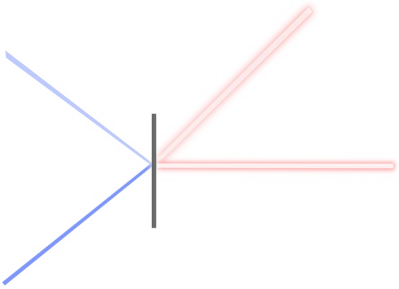

MirrorVerse

Ali Mohammad
Calderon Guillaume
Dechelette Eymeric
2023-2024

Table des matières

I) Introduction

Ce projet fait suite à la demande d'un étudiant, Quentin COURDEROT, en troisième année en spécialité informatique à Polytech. Il a demandé à son enseignant Jérôme Bastien de l'aider à écrire un algorithme pour déterminer la trajectoire d'un rayon lumineux lorsque celui-ci vient frapper un miroir plan fini.

L'objectif de ce projet est donc d'étudier le comportement d'un rayon lumineux lorsqu'il rentre en collision avec des miroirs. Il y a alors deux comportements possibles : le rayon peut être piégé dans le nid de miroirs et se réfléchir à l'infini, ou il peut parvenir à sortir du nid de miroirs. Sa trajectoire, quant à elle, peut suivre un motif ou non.

Dans le cadre de ce projet, nous développerons un outil permettant de simuler et de visualiser le comportement des rayons lumineux lorsqu'ils rencontrent des miroirs. La simulation devra autant que possible être juste physiquement, c'est-à-dire qu'elle devra coller au maximum à la réalité. Elle s'appuiera sur la seconde loi de Snell-Descartes (réflexion) et devra fonctionner au minimum en deux dimensions et avec des miroirs plans.

II) Création du programme

1) Stratégie générale

Pour répondre à cette problématique de simulation de réflexion de rayons lumineux sur des miroirs, nous nous sommes tout d'abord appuyés sur la loi de Snell-Descartes sur la réflexion. Nous avons cependant réadapté sa forme « classique » utilisant des angles pour utiliser une forme fondée sur des symétries orthogonales dans un espace euclidien. Cela nous a permis de rédiger le code de manière générique pour toute dimension afin de simplifier le passage de 2D à 3D.

Ainsi, notre programme demandera, à chaque miroir, les plans tangents à chaque point d'intersection avec le rayon lumineux. Ensuite, il fera avancer le rayon jusqu'au point d'intersection avec le plan le plus proche.
Enfin, il appliquera la symétrie orthogonale du vecteur de direction du rayon par rapport a l'espace directeur de ce plan, calculant ainsi la nouvelle position et la nouvelle direction du rayon. Ce processus se répète jusqu'à qu'il n'y ait plus de points d'intersection entre le rayon et un miroir de la simulation, ou que l'on ait atteint le maximum de reflexions demandé.

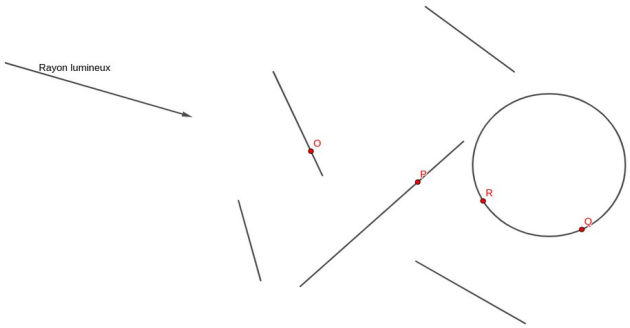

Figure 1: Schéma du calcul des intersections avec un rayon

III) Organisation du travail

1) Organisation temporelle

Nous avons choisi, pour notre organisation en termes de temps, d'adopter un cycle AGILE.
Comme nous travaillons sur ce projet sur notre temps libre, nos sprints (un cycle) était d'une durée de 18 jours.

Nous commencions par travailler pendant sept jours en parallèle chacun respectivement sur l'affichage, l'architecture de l'interface qui définit les miroirs, et l'exécution de la simulation. Ensuite, nous prenions 7 autres jours pour relier toutes ces parties ensemble afin d'avoir un programme fonctionnant de manière cohérente. Nous finissions par 4 jours pour réaliser des tests unitaires et où d'intégration dans le but de s'assurer que nous ne régresserons pas à l'avenir.

Figure 2: Diagramme de Gantt pour V1

Cependant, cette organisation, s'est vue remise en question à la moitié du projet quand nous nous sommes rendus compte que l'architecture de code que nous avions choisi n'était pas optimale. C'est pourquoi nous avons décidé de mettre en pause le développement pendant 1 semaine afin de réaliser un « refactor » (réorganisation/refonte du code).

C'est lors de cette réorganisation que nous avons choisi de séparer le code en 4 portions distinctes.

- Un programme qui lance une simulation à partir de sa représentation dans un format JSON, qu'on récupère dans un fichier dont le chemin est indiqué par l'utilisateur en argument.
- Un second programme qui permet de générer une simulation aléatoire (le nombre de miroirs, le type, et les paramètres de chaque miroir sont générées aléatoirement) pour ensuite la représenter en format JSON, et enregistrer cette représentation dans un fichier dont le chemin est indiqué par l'utilisateur en argument.

- Le moteur principal de calcul, de simulation, et de création de miroirs, sous la forme d'une librairie, qu'utilisent les deux programmes cités ci-dessus.
- Enfin, un programme, qui relie les trois parties précédentes, par le biais d'une interface graphique, permettant de gérer, générer (aléatoirement) et lancer des simulations, sans avoir à utiliser la ligne de commandes.

Cette réorganisation du code a temporairement ralenti notre développement mais nous a permis d'intégrer plus efficacement et plus simplement de nouveaux miroirs, et de nouveaux outils dans la simulation comme par exemple, un outil qui détecte les boucles (infinies) effectuées dans le trajet du rayon lumineux. Cela a, de plus, grandement simplifié la création de l'interface graphique globale.

2) Répartition des tâches

Pour ce projet, nous avons choisi de répartir les tâches de la manière suivante :

- Tout d'abord, Eymeric s'est chargé de l'implémentation des miroirs et de l'interface utilisateur (UI), ainsi que des tests unitaires. Il a également choisi de se charger de l'interface utilisateur, comme celle-ci est développée en Flutter, un outil qu'il maitrise fortement, parce qu'utilisé dans plusieurs projets personnels.
- Ensuite, Guillaume a pris en charge le rendu graphique et l'affichage des simulations. Étant déjà familier avec le rendu 3D grâce à ses expériences antérieures sur d'autres projets personnels, il a pu créer des rendus visuels fluides et de qualité, donnant ainsi une représentation réaliste et claire des miroirs dans l'environnement virtuel de l'application.
- Quant à Mohammad, il s'est concentré sur l'architecture du code, les algorithmes de réflexion, la sauvegarde et le chargement des simulations dans des fichiers, la génération aléatoire, et une partie de l'implémentation de certains miroirs. Sa maîtrise avancée du langage Rust lui a permis de diriger efficacement l'architecture du code, tandis que ses connaissances en mathématiques ont été mises à profit pour développer des algorithmes de réflexions précis et performants.

Cette répartition des tâches a été choisie dans le but de capitaliser le plus possible sur nos compétences individuelles pour assurer le succès du projet dans le temps imparti.

IV) Outils utilisés

1) Le Langage Rust

Le langage Rust est un langage libre, connu pour sa performance, sa fiabilité, et son ergonomie, mais également sa difficulté, en effet, une de ses particularités est que c'est un langage de programmation système (comme C/C++) mais qui ne permet pas l'écriture de programmes contenant des erreurs de sécurité de mémoire grâce à un ensemble de règles supplémentaires qu'applique le compilateur. On se réjouit donc de toute la performance et la puissance d'un langage de programmation système, sans se soucier de la catégorie de bugs (particulièrement difficiles à déboguer) qu'on y retrouve. Il nous fallait un langage performant (pour effectuer un grand nombre de calculs en temps réel) mais agréable à utiliser. De plus, un membre sur 3 connaissait déjà Rust en profondeur, et les deux autres souhaitaient renforcer leur savoir-faire dans ce langage à travers un projet intéressant. C'était donc le candidat parfait !

Une autre particularité qu'on y retrouve, c'est l'existence de types « somme ». Dans la majorité des langages, il y a souvent une manière de créer des types de données qui contiennent plusieurs autres types de données. On retrouve cette fonctionnalité le plus souvent sous la forme de *structures* ou *classes,* cependant on retrouve rarement une manière de définir un type qui se comporte, comme $A \Delta B$, c'est à dire un type de données qui contient soit un type soit un autre. On retrouve cette fonctionnalité en Rust, et elle a été particulièrement utile dans notre projet :

Un hyperplan affine,
représenté par un point « centre »,
puis soit un vecteur normal, soit une base.
Variante « Plan »: contient une base.←
Variante «normal » : contient un vecteur normal.←

```rust
pub enum Tangent<const D: usize> {
    Plane(Plane<D>),
    Normal {
        origin: SVector<f32, D>,
        normal: Unit<SVector<f32, D>>,
    },
}
```

Figure 3: Exemple d'utilisation de type somme

Ce qui nous permet d'écrire du code comme ceci :

```rust
pub fn orthogonal_symmetry(&self, v: SVector<f32, D>) -> SVector<f32, D> {
    match self {
        Tangent::Plane(plane) => 2.0 * plane.orthogonal_projection(v) - v,
        Tangent::Normal { n, .. } => v - 2.0 * v.dot(n) * n,
    }
}
```

Figure 4: Exemple de code utilisant les types somme

Cette fonction effectue la symétrie orthogonale de v par rapport à l'hyperplan directeur de *self* (qui est de type *Tangent*). Ici, on distingue les deux cas possibles de manières de le représenter l'hyperplan directeur : si c'est une base, on effectue la symétrie grâce à la formule $2p - Id$. S'il contient un vecteur normal, on utilise la formule $Id - 2p$.

2) Le format JSON:

Une autre question qui s'est posée était celle de la sauvegarde et de la réutilisation des simulations. Il nous fallait un format de fichier lisible et compréhensible par un être humain, et facilement utilisable dans le programme. Le format JSON était donc parfaitement adapté ! Omniprésent dans le web, il fonctionne avec un système de clé/valeur : chaque clé est associé à une liste ou un objet qui peut, lui aussi, contenir des paires de clés/valeurs, etc. De nombreuses bibliothèques rust servant d'interface avec ce format sont disponibles. Voici un exemple de simulation représente avec ce format :

Figure 5: Exemple de JSON

V) Formes implémentées :

Dans ce qui suit, $E=\mathbb{R}^n$ qu'on munit de sa structure euclidienne canonique.

Le rayon lumineux, sera représenté par la demi-droite d'équation :

$P+tD$ Avec $P, D\in E, \|D\|=1$, et $t\in\mathbb{R}^+$

1) Plan

Intuitivement, ils représentent une portion d'un hyperplan affine de E.

Soient C ∈ E et {b_1, ..., b_{n-1}} une famille libre de n – 1 vecteur.

Un point M appartient au miroir plan de centre C et de base B si et seulement si :

$$\exists(\mu_n,\dots,\mu_{(n-1)})\in[-1;1]^{n-1}, M=C+\sum_{i=1}^{n-1}\mu_i b_i$$

On remarque, ainsi, qu'en dimension 2 ce miroir a la forme d'un segment, en dimension 3 celle d'un parallélogramme, en dimension 4 celle d'un parallélotope, etc...

a) Intersection avec le rayon

En remplaçant donc V par l'équation du rayon : $P+tD$,

On a donc : $P-C=\sum_{i=1}^{n-1}\mu_i b_i - tD$

En considérant A la matrice de la famille $(-D, b_1,\dots,b_{n-1})$ (dans la base canonique), l'équation devient donc :

$$P-C=A\begin{bmatrix} t \\ b_1 \\ \vdots \\ b_{n-1}\end{bmatrix}$$

Ainsi, t est la première coordonnée du vecteur $A^{-1}(P-C)$, On remarque A^{-1} n'existe pas si et seulement si le rayon est parallèle au plan (et donc que $D\in Vect\{b_1,\dots,b_{n-1}\}$)

b) Plans Tangents

Le miroir en question étant plan, son plan tangent, en tout ses points, est… lui-même.

2) (Hyper)Sphère

Soient $C \in E$, et $r \in \mathbb{R}$.

Soit $f_{C,r} \in \mathbb{R}^E, f_{C,r}(V) = \|V - C\|^2 - r^2$

a) Intersection avec le rayon

Un vecteur V appartient à l'(hyper)sphère de centre C et de rayon r si et seulement si:

$\|V - C\|^2 = r^2$, C'est-à-dire $f_{C,r}(V) = 0$

En remplaçant donc V par l'équation du rayon : $P + tD$, l'équation ci-dessus devient une équation polynomiale (en t) de degré 2, qu'on peut facilement résoudre par la méthode du discriminant.

b) Plans tangents

On considère maintenant que V appartient à la sphère

Le vecteur $\nabla f_{C,r}(V) = 2(V - C)$

Est orthogonal a T, plan tangent a $f_{C,r}$ en V.

Ainsi, le vecteur $N = \frac{v - c}{\|V - C\|}$ est un vecteur normal au plan T

3) Parabole (2D uniquement)

Ceux-ci sont définis par une ligne directrice, un point focal et une ligne limite. La ligne limite sert à « couper » la parabole afin qu'elle ne se prolonge pas à l'infini.

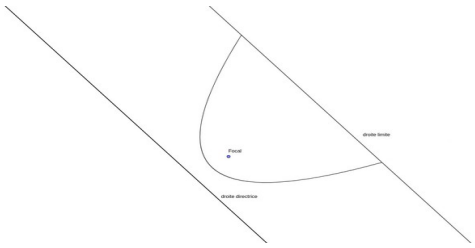

Figure 6: Schéma exemple de miroir parabolique

Nous avons implémenté le calcul d'intersection en deux dimensions en calculant la distance entre le point du rayon et la parabole. Nous avons ensuite implémenté l'algorithme de Newton-Raphson afin d'approcher la parabole. Par sécurité, nous finissons par vérifier que nous avons bien trouvé un point qui appartient à la parabole avec la distance entre le point et le focus, et la distance entre le point et la directrice.

Cependant, cette méthode n'a été réalisée qu'en 2D et n'a pas réellement put être testé par manque de temps.

De plus, l'affichage des paraboles, c'est révélé plus complexe que prévu et n'a pas pu être fini à temps, c'est pourquoi elles n'ont pas été activées dans le programme final.

VI) Fonctionnalités supplémentaires

1) Génération aléatoire

Nous avons également intégré un système de génération d'ensemble de miroirs aléatoires.
Cela nous permet donc facilement de préciser le nombre de miroirs plan et sphérique que l'on souhaite et aléatoirement les miroirs seront générés.

2) Analyse de trajectoire

Ces fonctionnalités étant des fonctionnalités que nous n'avions prévues qu'en fin de projet, nous n'avons eu que très peu de temps pour les implémenter.

C'est pourquoi nous n'avons implémenté que la détection de boucle.
Ainsi, si le rayon repasse au même endroit, notre programme le détecte automatiquement et arrête la simulation, car nous savons que les prochaines réflexions seront les mêmes que les précédentes.

Cependant, avec plus de temps, nous aurions pu imaginer nombre d'autres possibilités d'analyse.

VII) Exemple de simulations

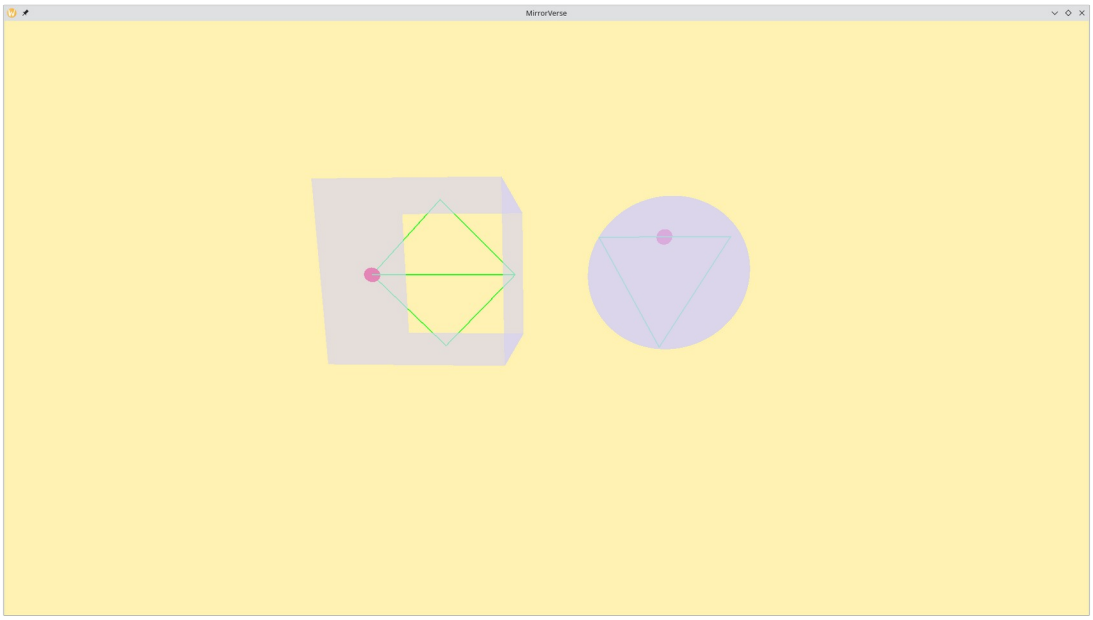

Figure 7: Exemple de simulation simple avec en vert, la detection de boucle de réflexions

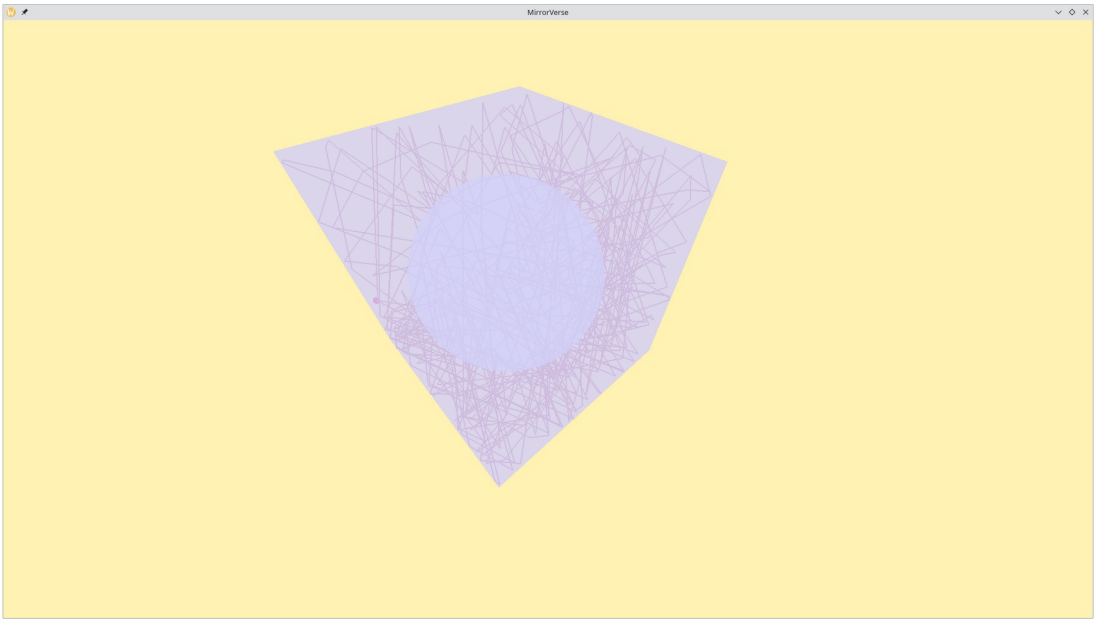

Figure 8: Exemple de simulation avec une sphère et plusieurs plans

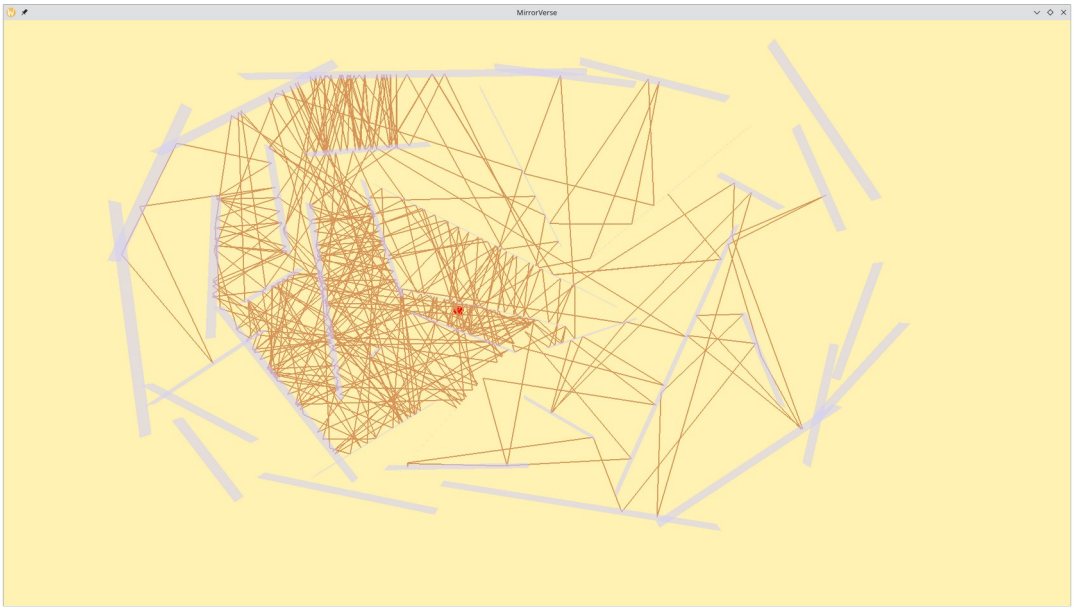

Figure 9: Exemple de simulation complexe avec des mirroir plan

VIII) Conclusion

Nous sommes satisfaits des résultats obtenus tout en maintenant un très haut niveau d'exigence en termes de qualité de code. Notre outil, bien que déjà fonctionnel et puissant, pourrait bénéficier d'améliorations supplémentaires, notamment dans sa convivialité d'utilisation et dans la gestion de plus de types de miroirs.

Enfin, une intégration avec d'autres logiciels ou plateformes de simulation pourrait élargir les possibilités d'utilisation de notre outil et favoriser la collaboration entre différentes équipes de recherche.

En somme, notre projet ouvre la voie à de nombreuses explorations dans le domaine de l'optique et de la simulation de phénomènes physiques, et il pourra certainement continuer à être développé dans les années à venir.

Index des figures

IV) Bibliographie

https://nalgebra.org/docs/user_guide/cg_recipes/

https://docs.rs/cgmath/latest/cgmath/

https://glium.github.io/glium/book/

1) Notions Mathématiques

https://www.math.univ-toulouse.fr/~guedj/fichierspdf/GeomDiff2015.pdf

2) (Hyper)sphère

https://en.wikipedia.org/wiki/N-sphere

https://fr.wikipedia.org/wiki/N-sph%C3%A8re

3) (Hyper)plan

https://fr.wikipedia.org/wiki/Hyperplan

https://en.wikipedia.org/wiki/Hyperplane

4) Parabole(oid)

https://en.wikipedia.org/wiki/Parabola

https://fr.wikipedia.org/wiki/Parabole

https://fr.wikipedia.org/wiki/Parabolo%C3%AFde

https://en.wikipedia.org/wiki/Paraboloid