Rapport Mirror Verse

Table des matières

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 rencontre 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 bien être chaotique. On considèrera qu'une trajectoire est chaotique si, après n réflexions (n dépendant du cas étudié), on ne constate aucune répétition.

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.

La simulation pourra par la suite être enrichie, en considérant, par exemple, plus de dimensions ou en intégrant une plus grande variété de miroirs.

Création de l’outil

Stratégie générale

Pour répondre à cette problématique de simulation de réflexion du rayon lumineux sur des miroirs, nous nous sommes tout d’abord appuyés sur la loi de Snell-Descartes sur la réflexion. Nous avons cependant adapté sa forme « classique » utilisant des angles pour utiliser une forme fondée sur des symétries dans l’espace.
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 pour chaque réflexion va simplement calculer les intersections entre le rayon lumineux (une droite) et tous les miroirs ainsi que la tangente (ou plan tangent). Il choisira ensuite le point d’intersection le plus proche du point de départ du rayon. Enfin, il effectua la symétrie au point d’intersection grâce à la tangente précédemment calculée afin de recommencer le calcul.

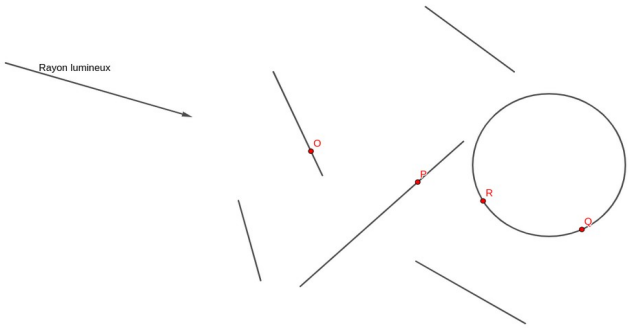

Figure 1: Schéma du calcul des intersections avec un rayon

Organisation du travail

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 7 jours en parallèle chacun respectivement sur l’affichage, l’architecture des miroirs et le calcul des réflexions pour ce miroir.
Ensuite, nous prenions 7 autres jours pour relier toutes ces parties ensemble afin d’avoir un ensemble 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.

Cependant, cette organisation, c'est vue remise en question à la moitié du projet quand nous nous sommes rendu compte que l’architecture de code que nous avions choisie n’était pas optimal.
C’est pourquoi nous avons décidé de mettre en pause le développement pendant 2 semaines afin de réaliser un refactor (réorganisation/refonte du code).

Figure 2: Diagramme de Gantt pour V1

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

- Une première ayant juste pour but de prendre le JSON en paramètre et de lancer la simulation avec celui-ci (run_simulation_json_3d)
- Une deuxième de générant des simulations aléatoires et de les enregistrer dans un nouveau fichier JSON (generate_random_simulation_3d)
- La plus grosse portion réalisant réellement les simulations. (mirror_verse)
- Et, une dernière qui permet via une interface graphique de lier les trois parties pour une utilisation simple (mirror_verse_ui)

Cette réorganisation du code, bien que nous aillant fait temporairement ralentir dans notre développement, nous a permis, une fois celui-ci effectué, de pouvoir être plus efficace pour l’intégration de nouveau miroir et d’outil, par exemple, de détection de boucle de réflexion.
Cela a, de plus, grandement simplifié la création de l’interface graphique globale.

Cependant, le but premier était que dorénavant le code peut être maintenu, mis à jour etc pour normalement plusieurs années et cela ouvre donc notre outil à de futures mises à jour même une

fois que nous ne travaillerons plus sur celui-ci. En effet, comme nous l’avons rendu disponible en open-source sur la platform GitHub, chacun peut le prendre, le modifier et s’en servir comme il le souhaite.

Répartition des taches

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 choisi ces taches, car l’implémentation des miroirs et les tests unitaires (développé en Rust) ne demandait pas de connaissance approfondie en Rust. En effet, celui-ci a découvert/appris le langage lors de ce projet. Il a également choisi de se charger de l’interface utilisateur, comme celle-ci est développée en flutter un framework qu’il maitrise, car utilisé dans plusieurs projets personnels.
- Ensuite, Guillaume a pris en charge le rendu 2D et 3D de chaque miroir dans un monde 3D. Étant déjà familier avec la 3D/2D grâce à ses expériences antérieures sur d'autres projets personnels, il a pu créer des rendus visuels 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 réflexions, ainsi que sur la sauvegarde et le chargement des simulations dans des fichiers, en plus de la génération aléatoire. Sa connaissance avancée de Rust lui a permis de diriger efficacement l’architecture du code, tandis que son expertise en algorithmique a été mise à profit pour développer des algorithmes de réflexions précis et performants.

Cette répartition des taches a été choisie dans le bus de capitaliser sur nos compétences individuelles pour assurer le succès du projet dans le temps imparti.

Formes implémentées

Hyperplan

Nous avons implémenté des miroirs sous forme d'hyperplans.

Ceux-ci sont définis par leur centre, leur base et leurs bornes.

Centre : [0.0,0.0]
Base : [[1.0, 0.0, 0.0],
[0.0, 0.0, 1.0]]
Borne:[3.0,5.0]

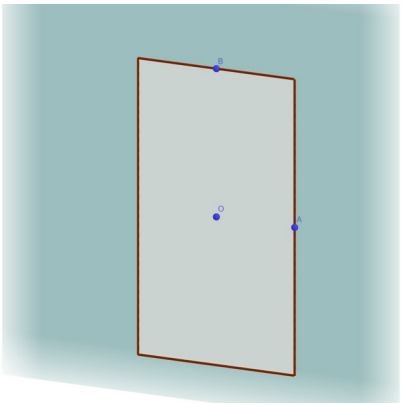

Figure 4: Schéma exemple de mirroir plan

Pour calculer les intersections avec les miroirs, nous réalisons une simple résolution de l'équation :

$$\begin{pmatrix} -\,direction\,rayon\,x & base\,plan\,1\,x & base\,plan\,2\,x & \dots \\ -\,direction\,rayon\,y & base\,plan\,1\,y & base\,plan\,2\,y & \dots \\ -\,direction\,rayon\,z & base\,plan\,1\,z & base\,plan\,2\,z & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

```rust
/// Returns a vector `[t_1, ..., t_d]` whose coordinates represent
/// the `intersection` of the given `ray` and `self`.
///
/// If it exists, the following holds:
///
/// `intersection = ray.origin + t_1 * ray.direction` and,
///
/// let `[v_2, ..., v_d]` be the basis of `self`'s associated hyperplane
///
/// `interserction = plane.origin + sum for k in [2 ; n] t_k * v_k`
pub fn intersection_coordinates(&self, ray: &Ray<D>) -> Option<SVector<f32, D>> {
    let mut a = SMatrix::<f32, D, D>::zeros();

    /* bien vuu le boss
    Fill the matrix "a" with the direction of the ray and the basis of the plane
    exemple
    | -ray_direction.x | plane_basis_1.x | plane_basis_2.x | ...
    | -ray_direction.y | plane_basis_1.y | plane_basis_2.y | ...
    | -ray_direction.z | plane_basis_1.z | plane_basis_2.z | ...
    */

    a.column_iter_mut() ColumnIterMut<f32, Const<D>, …, …>
        .zip(iter::once((-ray.direction).as_ref()).chain(self.basis().iter())) impl Iterator<Item = (Matrix<…, …, …, …>, ….
        .for_each(|(mut i, o)| i.set_column(i: 0, column: o));

    a.try_inverse_mut() bool
        // a now contains a^-1
        .then(|| a * (ray.origin - self.v_0()))
}
```

Figure 3: Implementation du calcul d'intersections avec hyperplans

Hyperphère

Nous avons implémenté des miroirs sous forme d'hypersphère.

Ceux-ci sont définis par leur centre ainsi que par leur rayon.

Afin de calculer les intersections avec les miroirs, nous réalisons une simple résolution de l'équation : $\left(\left(x-x_0\right)^2+\left(y-y_0\right)^2+\left(z-z_0\right)^2=r^2\right)-\left(P=P_0+tD\right)$

```rust
fn append_intersecting_points(&self, ray: &Ray<D>, list: &mut Vec<Tangent<D>>) {
    // TODO: more calculations can be offset to the inside of the if block
    // mental note: Cauchy-Schwarz

    let d = &ray.direction;
    let a = d.norm_squared();

    let v0 = &self.center;
    let v = ray.origin - v0;

    let b = v.dot(d);

    let r = &self.radius;
    let s = v.norm_squared();
    let c = s - r * r;

    let delta = b * b - a * c;

    if delta > f32::EPSILON {
        let root_delta = delta.sqrt();
        let neg_b = -b;

        for t in [(neg_b - root_delta) / a, (neg_b + root_delta) / a] {
            let origin = ray.at(t);
            let normal = Unit::new_normalize(origin - v0);
            list.push(Tangent::Normal { origin, normal });
        }
    }
} fn append_intersecting_points
```

Figure 5: Implementation du calcul d'intersections avec n-sphere

Parabole

Nous avons implémenté des miroirs sous forme de paraboles.

Ceux-ci sont définis par un hyperplan directeur, un point focal et un hyperplan de limite.
Le plan de limite sert à pouvoir « couper » la parabole afin quelle ne se prolonge pas à l'infinie

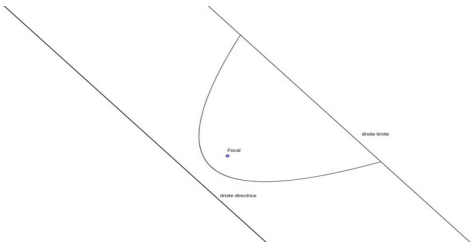

Figure 6: Schéma exemple de mirroir parabolique

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.

(L'implémentation étant trop longue, elle ne sera pas affichée ici)

Fonctionnalités supplémentaires

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.

Une des problématiques rencontrées lors de l'implémentation de cette fonctionnalité était les différents facteurs à appliquer au nombre aléatoire (compris de 0 à 1) afin d'avoir un rendu cohérent.
Nous avons donc particulièrement dû augmenter le facteur pour le centre du miroir dans le but de ne pas avoir tous les miroirs les uns sur les autres.

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.

Conclusion

Malgré le manque de temps, 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.

Bibliographie