

**Rapport de stage**Intégration de l'éclairement global dans la plateforme **HERA**  
Du 4 mars au 24 juillet 2025

Amélia Di Martino



**Tuteurs entreprise** :Jean-Philippe Farrugia, Fabrice Jaillet  
**Tuteur universitaire** : Raphaëlle Chaine

**Établissement** : Université Lyon 1  
**Année de formation** : M2 Informatique, ID3D  
**Entreprise d’accueil** : LIRIS

Année universitaire 2024-2025

**Sommaire**

[Remerciements 3](#_Toc1)

[1. Introduction 4](#_Toc2)

[1.1 Présentation de l’entreprise 4](#_Toc3)

[1.2 Contexte 5](#_Toc4)

[1.3 Sujet du stage 7](#_Toc5)

[1.4 Outils et environnement de travail 7](#_Toc6)

[2. Travail réalisé 8](#_Toc7)

[2.1 Gestion des matériaux 8](#_Toc8)

[2.2 Intégration de l’éclairement global 9](#_Toc9)

[2.2.1 Techniques intéressantes 9](#_Toc10)

[2.2.2 Light Probe Volume 13](#_Toc11)

[2.2.2.2 Sampling 18](#_Toc12)

[2.2.3 Résultats 19](#_Toc13)

[3. Conclusion 20](#_Toc14)

[Sources 20](#_Toc15)

[Table des figures 20](#_Toc16)

[Annexes 20](#_Toc17)

# Remerciements

Dans un premier temps, je tiens à remercier mes deux tuteurs de stage, Jean-Philippe Farrugia ainsi que Fabrice Jaillet, de m’avoir encadrée, suivie et conseillée tout au long de ce stage.  
Malgré la distance, vous avez toujours été présents, patients et de bon conseil pour m’aider à avancer lors de ce stage.

Je remercie Raphaëlle Chaine de m’avoir parlé de ce sujet de stage, sujet mêlant plusieurs passions qui me touchent.

Je tiens à remercier Noah Bertholon, avec qui j’ai pu échanger sur certaines problématiques de mon stage, dont la lumière fût souvent rafraîchissante.

De même, je tiens à remercier Jean-Claude Iehl pour ses conseils et les références bibliographiques qui m’ont grandement aidée lors de ce stage.

Ce stage a été pour moi très enrichissant, j’ai eu la chance de beaucoup apprendre, par moi-même et grâce aux autres.

# 1. Introduction

## 1.1 Présentation de l’entreprise

Le **LIRIS** (Laboratoire d’infoRmatique en Image et Système d’information) est une unité mixte de recherche qui mélange CNRS, INSA Lyon, l’UBCL Lyon 1, l’Université Lumière Lyon 2 ainsi que l’École Centrale de Lyon.

Il est ainsi de composé de 330 membres, divisés en plusieurs équipes avec leurs thématiques de recherche propres.

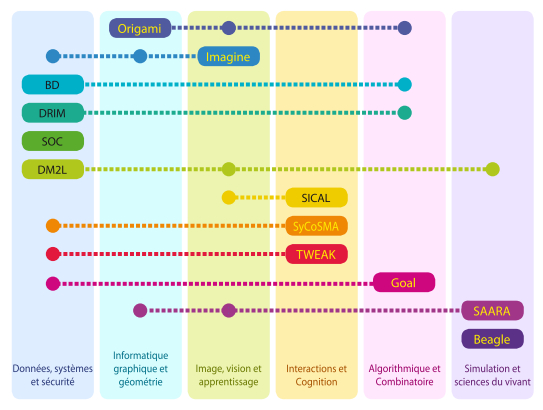


Figure 1 : Équipes et thématiques

Ce stage prend place au sein de l’équipe **Origami**, dont les thèmes de recherches gravitent autour de de l’informatique graphique et de la géométrie.

## 1.2 Contexte

Avant toute chose, il convient de présenter **HERA** (**H**istorically **E**nhanced **R**eality **A**pplication), l’application sur laquelle ce stage porte.  
Il s’agit d’une application web de réalité augmentée et réalité virtuelle, prévue pour des non-informaticiens afin de pouvoir créer et accompagner des visites guidées de sites historiques ou de musées.

L’application est codée en javascript + vue.js, afin de pouvoir l’utiliser de manière portative, sur téléphone, PC, tablette, casques VR...  
La partie affichage 3D utilise ThreeJS.

L’application est divisée en deux parties :

Un **éditeur**, pour permettre au guide de préparer la visite :  
Son objectif n’est pas de recréer Blender, mais il permet quelques opérations simples : rajouter des maillages (au format GLB), les bouger, changer les propriétés basiques des matériaux...  
Il est aussi possible de rajouter des labels (des zones de textes), ainsi que des timings pour gérer le moment où ils apparaîtront lors de la visite.

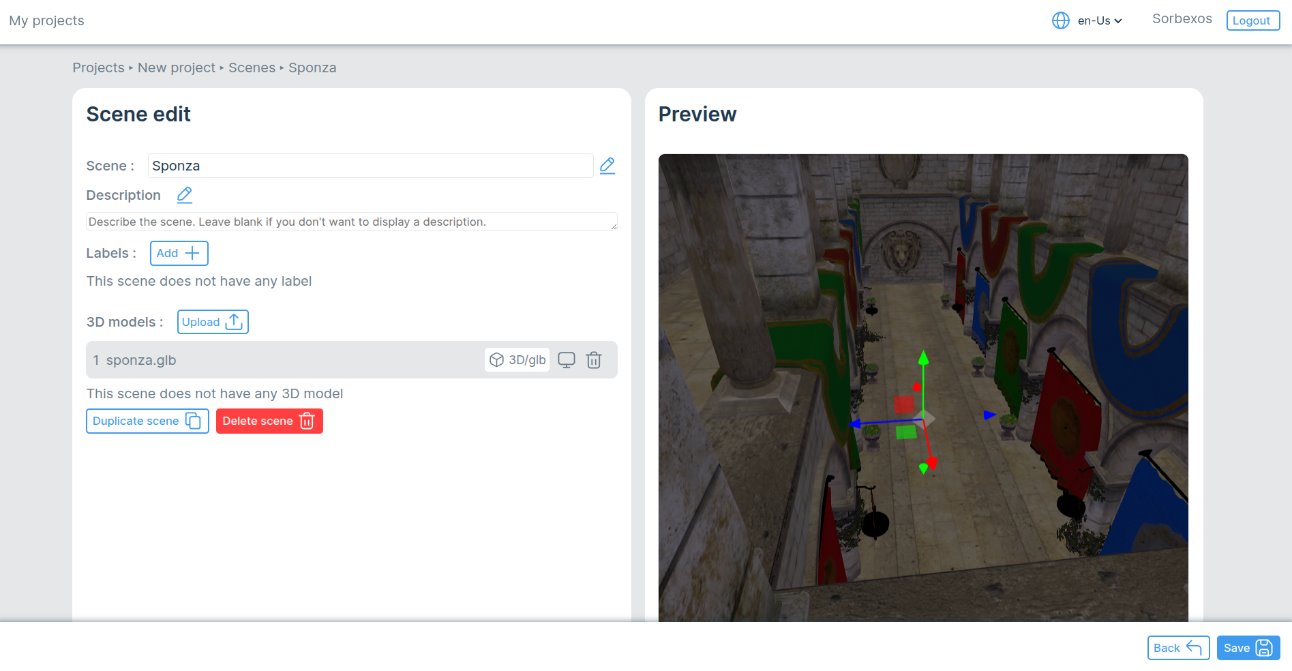


Figure 2 : Partie éditeur de HERA

Un **viewer**, permettant au visiteur d’avoir la scène 3D devant ses yeux, en réalité augmentée ou virtuelle. Il s’agit simplement de calibrer la scène en plaçant la caméra par rapport à un repère prévu, et ainsi pouvoir se mouvoir dans la scène grâce à son appareil.

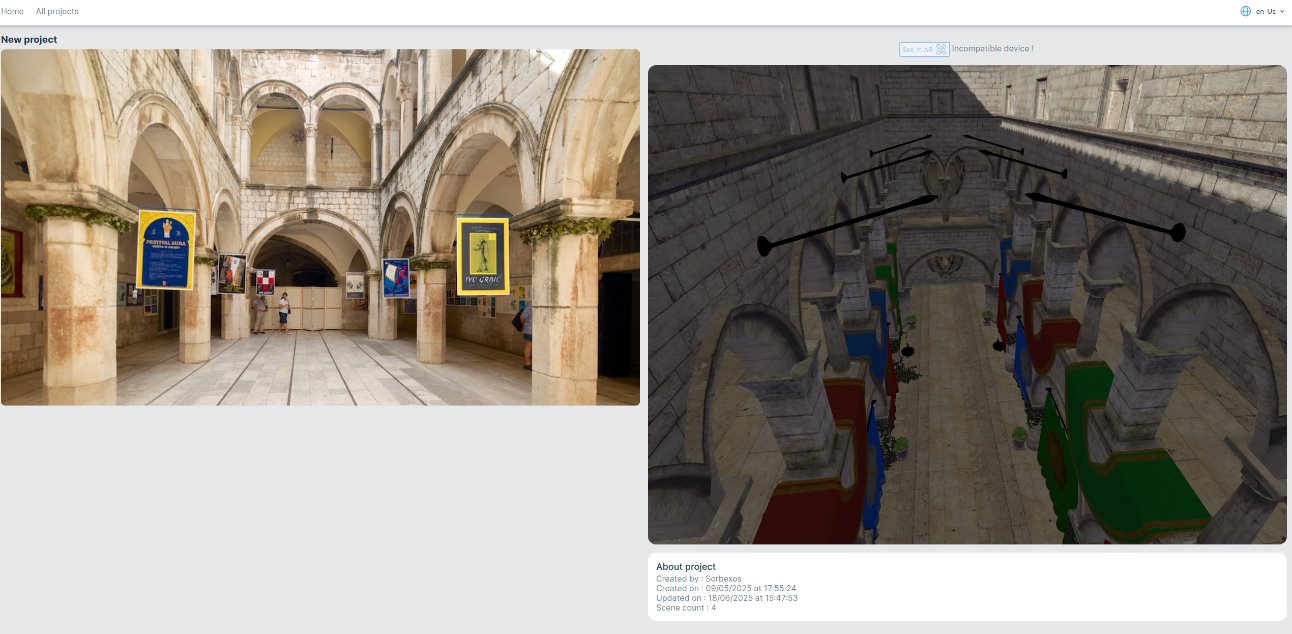


Figure 3 : Partie viewer de HERA

## 1.3 Sujet du stage

L’objectif premier de ce stage était de rajouter des fonctionnalités à l’application.

Plus largement, l’objectif de ce stage est d’utiliser les travaux réalisés afin de pouvoir étudier l’impact des techniques d’**illumination globale** sur la perception humaine.

Avant ce stage, l’éclairage de la scène restait sommaire : une lumière ambiante ainsi qu’une lumière directionnelle pour simuler le soleil.

Le but est donc de pouvoir créer des tests afin de comparer l’impact de l’illumination globale, ainsi que pouvoir avoir différents presets (jour, nuit...) d’éclairage.

## 1.4 Outils et environnement de travail

J’ai passé mon stage dans les boxs prévu pour les stagiaires.  
J’ai utilisé mon propre ordinateur portable, ayant besoin d’un ordinateur avec un peu de puissance de calcul.  
Pour ce qui est des outils informatique, j’ai principalement utilisé VScode, Chrome pour utiliser l’application, ainsi que Gitlab.  
Mes encadrants étant basés à Bourg-En-Bresse, nous utilisions principalement Discord pour communiquer.  
Nous nous voyions au miminum une fois par semaine en présentiel à Lyon pour pouvoir faire des points réguliers sur l’avancement du stage.

# 2. Travail réalisé

## 2.1 Gestion des matériaux

Dans l’objectif de me familiariser avec la base de code, j’ai passé les premières semaines à implémenter la possibilité de changer les propriétés des matériaux des mesh importés sur l’éditeur.

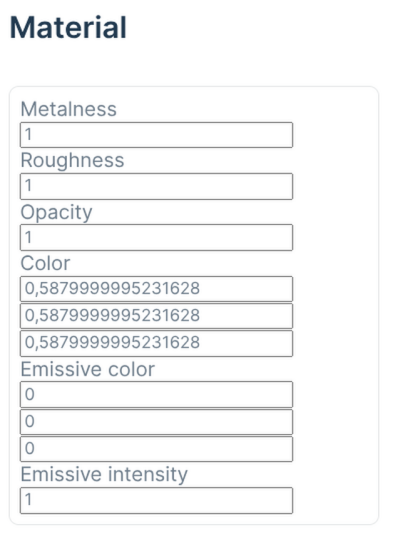


Figure 4 : Menu d'édition de matériaux

Cela était un exercice intéressant, sachant que la base de code n’était pas du tout prévue pour faire cela.  
En effet, chaque objet GLB était importé sous forme d’« asset ». Un asset est une arborescence de différents mesh, qui chacun possède son propre matériau.

Changer le matériau d’un objet impliquait donc de complètement modifier la manière dont les objets étaient importés, afin de pouvoir sélectionner un élément spécifique de la scène, et non toute la scène.

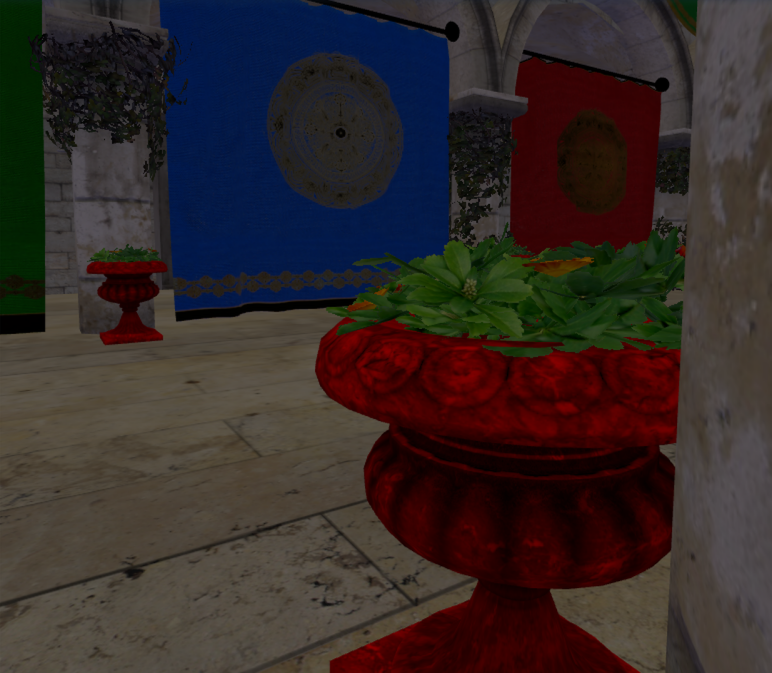
Lorsqu’on importe un asset, il faut traverser son arborescence de mesh afin de les ajouter à la scène, ainsi qu’à la base de donnée, afin de pouvoir stocker leurs spécificités (transformations géométriques et matériau).

J’ai dû me familiariser avec la sauvegarde des éléments de la scène dans la base de données côté backend, ainsi que leur utilisation dans le frontend.

Cette tâche convenait tout à fait à la découverte et la compréhension de la base de code.  
Il m’a fallu m’inspirer de ce qui existait déjà (enregistrement des assets, des labels de texte, des scènes...), et plus globalement du fonctionnement de l’API.  
De même, il m’a fallu comprendre le chemin des objets du chargement de la page à leur affichage dans ThreeJS.

Figure 5 : Pot de fleurs rouge

Figure 6 : Pot de fleurs gris



## 2.2 Intégration de l’éclairement global

Dans un premier temps, il convenait de choisir une technique d’illumination globale.  
En effet, les contraintes sont fortes : l’éclairage doit être réaliste, le tout en temps réel sur n’importe quel support (pc, tablette, téléphone...).

En vue des fortes contraintes, il fallu faire un travail important de recherche afin de cataloguer les différentes techniques pouvant convenir.

Nous présenterons dans un premier temps les quelques techniques qui ont retenu notre attention, et ensuite la technique principale que nous avons choisi.

### 2.2.1 Techniques intéressantes

**SSGI (Screen Space Global Illumination)**

Première technique présentée comme pouvant fonctionner en temps réel.  
Une [lib ThreeJS](https://github.com/0beqz/realism-effects) implémentant cette technique, il nous a paru pertinent de l’essayer.  
Le but n’était pas forcément n’implémenter une technique à partir de zéro : si quelque chose existait déjà et donnait des résultats intéressants, nous l’aurions utilisé.

L’idée est simple : prendre en compte uniquement les objets à l’écran : cela limite évidemment beaucoup les temps de calcul, même si on perd en réalisme, car uniquement ce qui apparaîtra à l’écran influera l’illumination de la scène.

Les résultats ne répondaient finalement pas à nos attentes : la technique, bien que plus légère que d’autres, restait trop lourde pour être utilisée sans GPU, et mettait trop de temps à converger vers une image correcte.



Figure 7 : SSGI sur la Sponza

Cette image (**figure 7**) a convergé en ~10 secondes, ce qui est bien trop long.  
Nous voudrions rester à 30-60 images par secondes, et nous sommes ici à 6 IPS.

Cette solution aurait aussi été avantageuse car elle permettait d’avoir de l’illumination globale sur ThreeJS, sans avoir à bidouiller dans les shaders de ce dernier.  
En effet, ThreeJS ne permet pas vraiment de modifier facilement sa pipeline de rendu.

**Instant radiosity (Virtual Lights)**

Cette technique a l’avantage d’être très simple à comprendre :

Elle remplace la contrainte de lancer beaucoup de rayons par pixel à chaque frame, par le fait d’avoir beaucoup de sources de lumières « secondaires » qui simulent les différents rebonds.

Dans un premier temps de baking, avant le rendu, on va créer des sources de lumières secondaire.  
En partant d’une source de lumière primaire (**L1 sur la figure 8)** on lance des rayons de manière aléatoire.  
Lorsqu’un rayon touche une surface, on va créer une source de lumière secondaire (**L1’, L2’... sur la figure 8)**.  
On peut répéter l’opération autant de fois que souhaité, tant qu’il reste de l’énergie dans notre rayon.

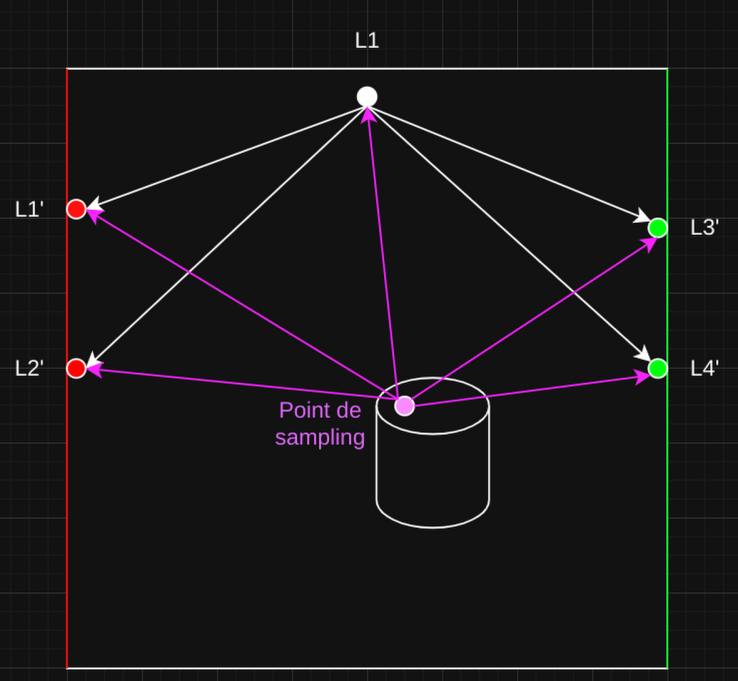


Figure 8 : Instant radiosity

Une fois nos lumières ponctuelles crées, il faut les transmettre au GPU.  
Et là...  
C’est le moment où nous nous arrêtons pour expliquer quelques contraintes liées à ThreeJS et à WebGL 2.0 : il n’y a **pas de buffer** dans l’API de ThreeJS. Cela signifie qu’il faudra passer par des textures afin de transmettre des données en quantité.  
En l’occurrence, ce que nous voulons transmettre ici, ce sont les positions des lumières, ainsi que leur puissance.  
(Par ailleurs, pas de compute shader non plus, même si cela n’est pas embêtant dans ce cas d’utilisation.)

Une fois cela compris, il ne manque plus qu’une étape : tester si oui ou non, sampler à chaque frame un nombre important de lumière ponctuelles est supportable pour l’iGPU de mon PC portable.

La réponse est **non** : on commence à voir nos IPS divisés par deux à partir d’une cinquantaine de lumière, ce qui n’est pas suffisant du tout pour simuler de l’illumination globale dans une pièce de taille respectable.

**Area Lights**

Une dernière solution a été essayée avant de passer à autre chose : les area lights de ThreeJS.



Figure 9 : Rect Area Light de ThreeJS

Cette solution est intéressante car elle permet d’avoir des lumières douces à faible coût.  
De plus, elle est déjà implémentée dans ThreeJS, et représente donc un avantage non négligeable en terme de temps d’implémentation.

Les area lights de ThreeJS fonctionnent grâce aux Lineary Transformed Cosines[5](#cosines).

Cette technique permet d’avoir des lumières en forme de polygones simples en temps réel.

Comme la BRDF des matériaux sont représentés par des fonctions sphériques, il faut donc intégrer le polygone sur une sphère afin de calculer l’éclairage.  
Cela pose problème pour faire du temps réel, car calculer l’intégrale demande de lancer de nombreux rayons par pixels.

Cette technique propose de remplacer cela par une approximation : une distribution en cosinus sur la sphère, linéairement transformée afin de correspondre au polygone.

Cette solution a donc été abandonnée, même s’il aurait été possible d’approximer des ombres en rajoutant des lumières ponctuelles à la surface des Rect Area Lights.

### 2.2.2 Light Probe Volume

Malgré une implémentation ardue, cette solution a été retenue pour plusieurs raisons :

* Tourne sans soucis en temps réel
* Permet des résultats réalistes
* Plutôt simple à comprendre
* Permet de mettre à jour l’illumination reçue par des objets de la scène lorsqu’ils se déplacent (nécessaire car rajouter des animations sera une fonctionnalité de HERA à l’avenir)

Les principaux points faibles de cette solutions sont :

* Pas de mise à jour des probes en temps réel (si un objet bouge, il sera illuminé correctement, mais n’illuminera pas correctement la scène)
* Coût en stockage

L’idée est simple :

Il s’agit de placer des sondes, des **Light Probes**, permettant d’échantillonner la luminosité reçue en un point de l’espace.  
Une fois la luminosité bakée, on enregistre ces informations dans un outil mathématique appelé **harmonique sphérique.**

Il s’agit d’une fonction sphérique qui pour une direction donnée, nous donne une valeur d’éclairage.  
La fonction est paramétrée par plusieurs coefficients : on peut avoir plus de précision avec plus de coefficients.  
Ici, nous utiliserons des harmoniques sphérique à 9 coefficients, chaque coefficient étant une couleur.

Un Light Probe Volume (aussi appelé Irradiance Volume) est donc un volume discret qui en un point, permet d’évaluer la lumière reçue dans une certaine direction.  
Les probes sont placées dans une grille régulière.

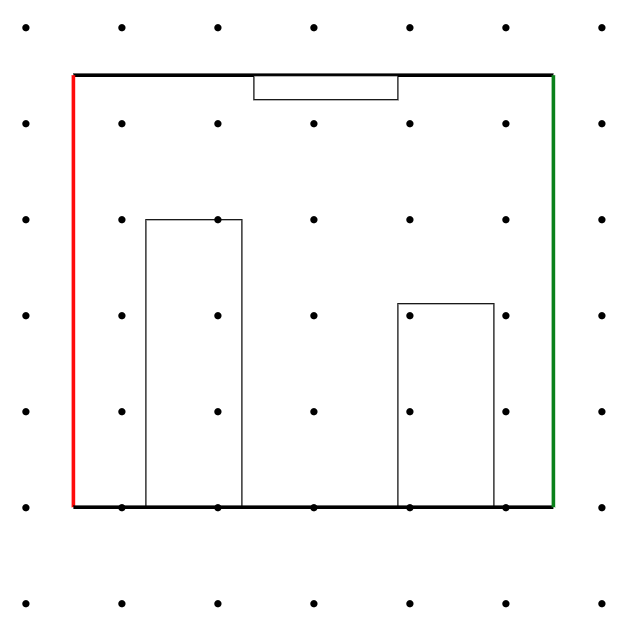


Figure 10 : Grille régulière de probes

#### 2.2.2.1 Baking

Nous utilisons la librairie de Jean-Claude Iehl gKit afin de pouvoir baker les probes.

Dans un premier temps, on traverse tous les triangles de la scène afin d'enregistrer les triangles émissifs.

Pour chaque probe, on calcule la lumière directe et indirecte reçue.

Pour l’illumination **directe**, on lance des rayons de manière aléatoire sur les sources de lumières de la scène. Le poids de ces sources est fixé en fonction de leur taille et de leur puissance.

Pour l’illumination **indirecte**, on lance des rayons autour de la sphère de façon régulière, autour d’une « spirale de Fibonacci ». Cela permet de couvrir correctement toutes les directions.  
Afin d’éviter un biais, on rajouter un offset aléatoire pour chaque probe pour éviter d’avoir le même sampling pour chaque probe. (en résumé, on se décale légèrement sur la spirale)

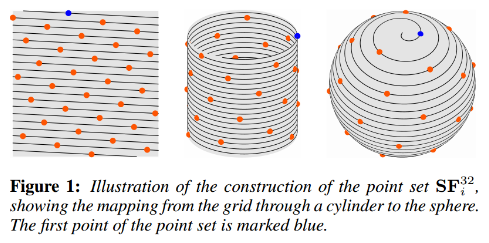


Figure 11 : Spirale de Fibonacci [1](#Fibonacci)

Un premier problème lié au placement des probes : certaines probes vont tomber dans la géométrie.   
Les rayons ne pourront donc pas atteindre et les lumières : cette probe est donc **invalide**.  
On va essayer de trouver le chemin le plus court pour sortir de la géométrie.  
On profite pour cela des rayons lancés pendant le calcul de l’éclairage indirect : on sauvegarde le chemin le plus court, et on s’y déplace, pour ensuite recalculer une nouvelle fois. (figure 12)

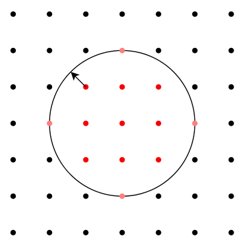


Figure 12 : Déplacement des probes invalides

On calcule un score d’invalidité : chaque rayon qui touche une backface augmente ce score.[2](#Unity)

Ce score nous sert un tout petit peu plus tard : si jamais on ne réussit pas à sortir de la géométrie (si on est entre deux objets), on remplit cette probe avec les données des probes valides voisines.

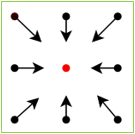


Figure 13 : Dilatation

Ensuite, dernier problème mais pas des moindres : il faut gérer les occultations !  
En effet, dans ce cas (figure 14) on veut sampler les probes de droite, mais pas celles de gauches car elles sont derrière un mur.

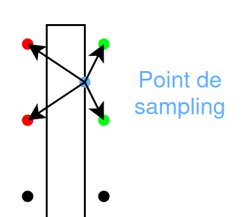


Figure 14 : Occultation des probes

Pour avoir cette information, il n’y a pas mille solutions... Chaque probe a besoin d’une depth map !  
En effet, lors du sampling (qui sera détaillé plus tard), nous avons besoin de connaître la distance entre la probe et ses alentours, et cela dans chaque direction.

Il faut donc enregistrer une nouvelle information sphérique, oui, mais pas dans des harmoniques sphériques.  
En effet, les différentes sources utilisées[3](#Implem),[4](#Nvidia)préfèrent utiliser des octmap. A l’usage, les harmoniques sphériques ne sont pas faites pour une information de profondeur.

Simplifier une sphère en octaèdre permet facilement de l’enregistrer sur une texture 2D, ce qui aide grandement son enregistrement ainsi que son sampling. (figure 15)

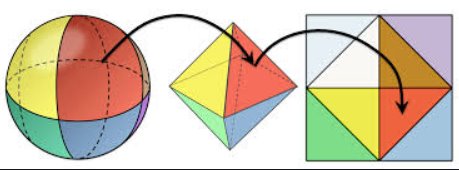


Figure 15 : Octahedral mapping [3](#Implem)

Pour ce qui est de générer des directions, c’est assez simple.  
Chaque point de l’octmap représente une direction. Lorsqu’on remplit la texture, on interpole entre les 3 directions du triangle dans lequel on se trouve.

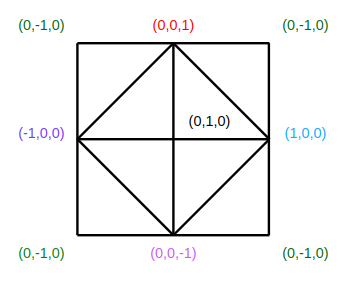


Figure 16 : Exemple d'octmap

Figure 17 : Directions de l'octaèdre

Maintenant qu’on a nos directions, on peut remplir nos octmap en lançant des rayons. Pour ce faire, on envoie plusieurs rayons par pixel, on calcule la moyenne et on enregistre également la variance, qui sera utilisée plus tard lors du sampling.

Octmaps comme harmoniques sphériques sont enregistrées dans des tableaux 1D (sous forme d’atlas) (.csv), car nous en avons besoin afin d’utiliser l’API de ThreeJS pour transmettre des textures 3D au fragment shader.  
Le fait d’utiliser des textures 3D simplifie grandement le sampling, pour passer simplement du repère monde au repère texture.

Pour pouvoir profiter de l’interpolation matérielle des textures, nous avons besoin de « recoudre » les bords des octmap, afin que les voisins qui serviront à l’interpolation soient bien des directions voisines sur la sphère.

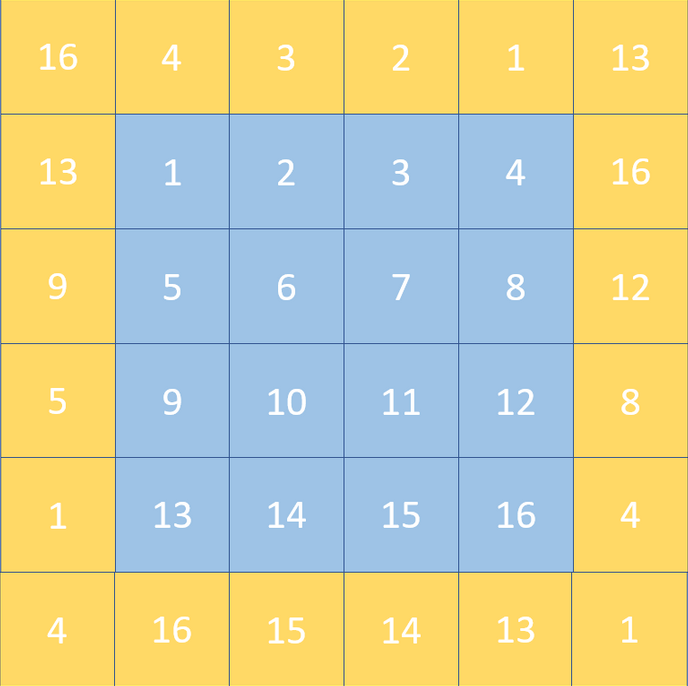


Figure 18 : Octmap en bleu, couture en jaune [3](#Implem)

#### 2.2.2.2 Sampling

Nous voilà maintenant avec nos probes bakées, équipées d’harmoniques sphériques et de depth map.

Nous sommes dans le fragment shader, avec un point de sampling, et une « cage » de 8 probes à sampler.

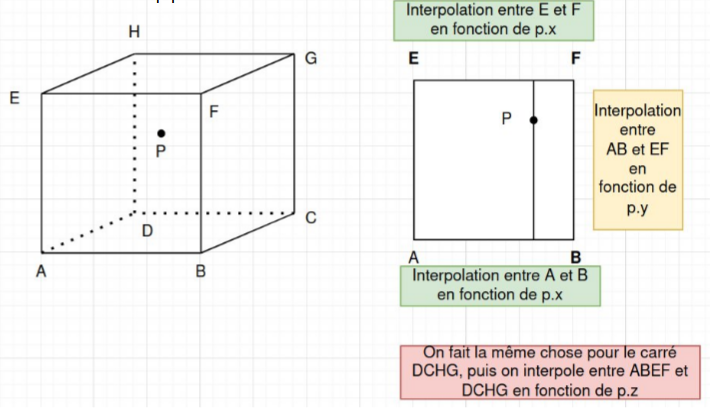


Figure 19 : Interpolation trilinéaire

L’objectif est de réaliser une simple interpolation trilinéaire, avec une contrainte : donner un poids à chaque probe en fonction de sa disponibilité (si elle est obstruée ou non)*.*

Afin d’éviter les changements brutaux lors de forts changement de profondeur d’un pixel à l’autre de la depth map, on utilise la variance pour modifier le poids de la probe obstruée.

Nous profitons bien-sûr de l’interpolation matérielle lorsqu’on requête les depth maps.

Obtenir le bon pixel de la depth map en fonction de la direction n’est évident.   
Il faut dans un premier temps passer du repère monde dans lequel est le point dont on cherche à connaître l’illumination reçue au repère texture.

Une fois dans le repère texture, il est assez simple de récupérer les 8 probes avoisinantes ainsi que les coefficients dépendant de la position du point dans le cube, qui serviront à l’interpolation.

Avec ça, on a tout ce dont on a besoin pour sampler l’irradiance, mais il nous reste l’octmap !  
Comment faire ?

L’idée est la suivante :

On se place d’abord en bas à gauche de notre depth map.  
Ensuite, en fonction de la direction entre le point et la probe, on cherche à savoir dans quel triangle on se trouve.  
Cela se fait assez simplement en découpant le triangle en fonction des axes. (figure 20).

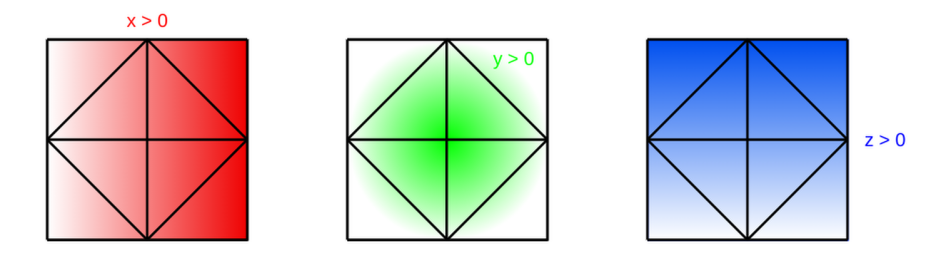


Figure 20 : Recherche du triangle en fonction de la direction

Maintenant qu’on a notre triangle, on peut facilement obtenir son « origine », qui ici sera toujours le point le plus en bas à gauche.

On a ensuite besoin de connaître les UV sur ce triangle, pour savoir de combien se décaler en fonction de notre direction.  
On calcule assez simplement l’intersection entre la direction et le triangle placé à l’origine.

Maintenant qu’on a notre triangle, son origine et ses UV, on peut finalement retrouver notre pixel sur la texture !

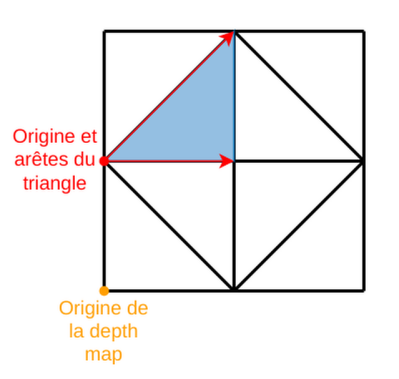


Figure 21 Ici, on a une direction avec x < 0, y > 0 et z > 0

Dans un second temps, il nous paraît pertinent d’expliquer que ThreeJS n’est pas prévu pour ce type d’utilisation.  
Nous avons ici besoin d’avoir un shader custom permettant de réaliser nos opérations.

Dans ThreeJS, les shaders sont stockées par matériau. Chaque mesh a un matériau, et chaque matériau a son propre shader.  
L’idée est donc de récupérer chaque shader, et de le modifier afin d’avoir une version customisée du shader de base de ThreeJS.

On y transmet donc pour chaque matériau nos uniforms (taille du volume de probes, densité...), nos octmap ainsi que nos harmoniques sphériques.

### 2.2.3 Résultats

Cette techinique se montre efficace, et permet comme prévu un affichage en temps réel grâce aux probes bakées.

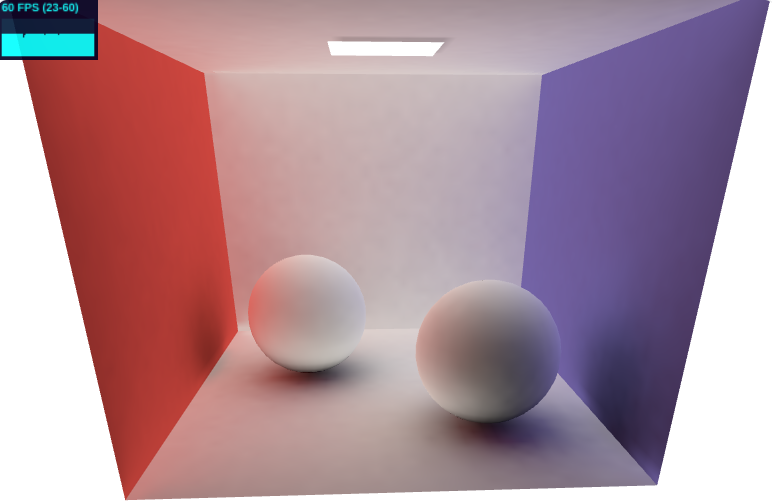


Figure 22 : 80000 probes, 512 rayons pour l'indirect

Il serait possible de baisser le nombre de probes bakées prenant en compte uniquement celles qui sont proches de la géométrie.  
Cela permettrait de baisser les coûts de stockage, ainsi que le temps de baking.  
De même, utiliser une structure de BVH pour le lancer de rayons accélérerait grandement les calculs.

Cette technique montre sa faiblesse au niveau des forts changement de lumière.

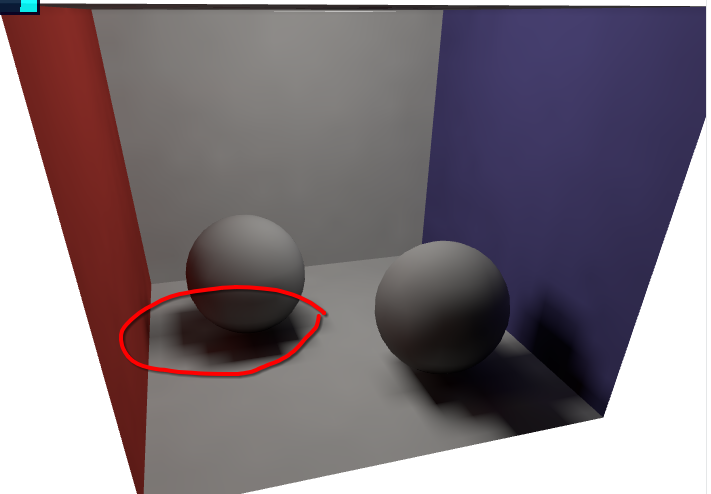


Figure 23 : 4096 probes, les ombres sont crénelées

Une solution possible serait de mélanger des shadow map classiques (éventuellement avec des lumières ponctuelles) et de profiter les lights probes uniquement pour l’éclairage indirect.

# 3. Conclusion

Ce stage a été ma première expérience dans le monde de l’informatique graphique.  
Bien qu’un stage applicatif, j’ai dû réaliser de nombreuses recherches pour résoudre au mieux les problématiques de ce stage, et cela fût très enrichissant.

Être entourée d’enseignants chercheurs et de doctorants m’a permis d’obtenir plus facilement des références intéressantes.

Bien qu’accompagnée par Jean-Philippe Farrugia et Fabrice Jaillet, c’est une première pour moi d’avoir eu autant d’autonomie sur un projet, et confronter ma compréhension du sujet à leurs attentes et connaissances fût très intéressant.

Les présentations de papiers régulièrement organisées par l’équipe Origimi ont été une occasion enrichissante pour s’ouvrir sur d’autres sujets.

# Sources

Figure 1 : <https://liris.cnrs.fr/liris>

1 : <https://www.lgdv.tf.fau.de/uploads/publications/spherical_fibonacci_mapping.pdf>

2 : [https://advances.realtimerendering.com/s2022/SIGGRAPH2022-Advances-Enemies-Ciardi%20et%20al.pdf](https://advances.realtimerendering.com/s2022/SIGGRAPH2022-Advances-Enemies-Ciardi et al.pdf)

3 : <https://handmade.network/p/75/monter/blog/p/7288-engine_work__global_illumination_with_irradiance_probes>

4 : <https://www.jcgt.org/published/0008/02/01/paper-lowres.pdf>

5 : <https://eheitzresearch.wordpress.com/415-2/>

# Table des figures

[Figure 1 : Équipes et thématiques 4](#_Toc1)

[Figure 2 : Partie éditeur de HERA 5](#_Toc2)

[Figure 3 : Partie viewer de HERA 6](#_Toc3)

[Figure 4 : Menu d'édition de matériaux 8](#_Toc4)

[Figure 5 : Pot de fleurs rouge 9](#_Toc5)

[Figure 6 : Pot de fleurs gris 9](#_Toc6)

[Figure 7 : SSGI sur la Sponza 10](#_Toc7)

[Figure 8 : Instant radiosity 11](#_Toc8)

[Figure 9 : Rect Area Light de ThreeJS 12](#_Toc9)

[Figure 10 : Grille régulière de probes 13](#_Toc10)

[Figure 11 : Spirale de Fibonacci 1 14](#_Toc11)

[Figure 12 : Déplacement des probes invalides 14](#_Toc12)

[Figure 13 : Dilatation 15](#_Toc13)

[Figure 14 : Occultation des probes 15](#_Toc14)

[Figure 15 : Octahedral mapping 16](#_Toc15)

[Figure 16 : Exemple d'octmap 16](#_Toc16)

[Figure 17 : Directions de l'octaèdre 16](#_Toc17)