



Université Claude Bernard



Lyon 1



Rapport de stage

Intégration de l'éclairage 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	3II
1. Introduction.....	4II
1.1 Présentation de l'entreprise	4II
1.2 Contexte	5II
1.3 Sujet du stage	7II
1.4 Outils et environnement de travail.....	7II
2. Travail réalisé	8II
2.1 Gestion des matériaux	8II
2.2 Intégration de l'éclairage global	9II
2.2.1 Techniques intéressantes.....	9II
2.2.2 Light Probe Volume	13II
2.2.2.1 Baking.....	14II
2.2.2.2 Sampling.....	17II
2.2.3 Résultats.....	17II
3. Conclusion.....	17II
Sources	17II
Table des figures.....	17II
Annexes.....	18II

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 lehl 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 composé de 330 membres, divisés en plusieurs équipes avec leurs thématiques de recherche propres.

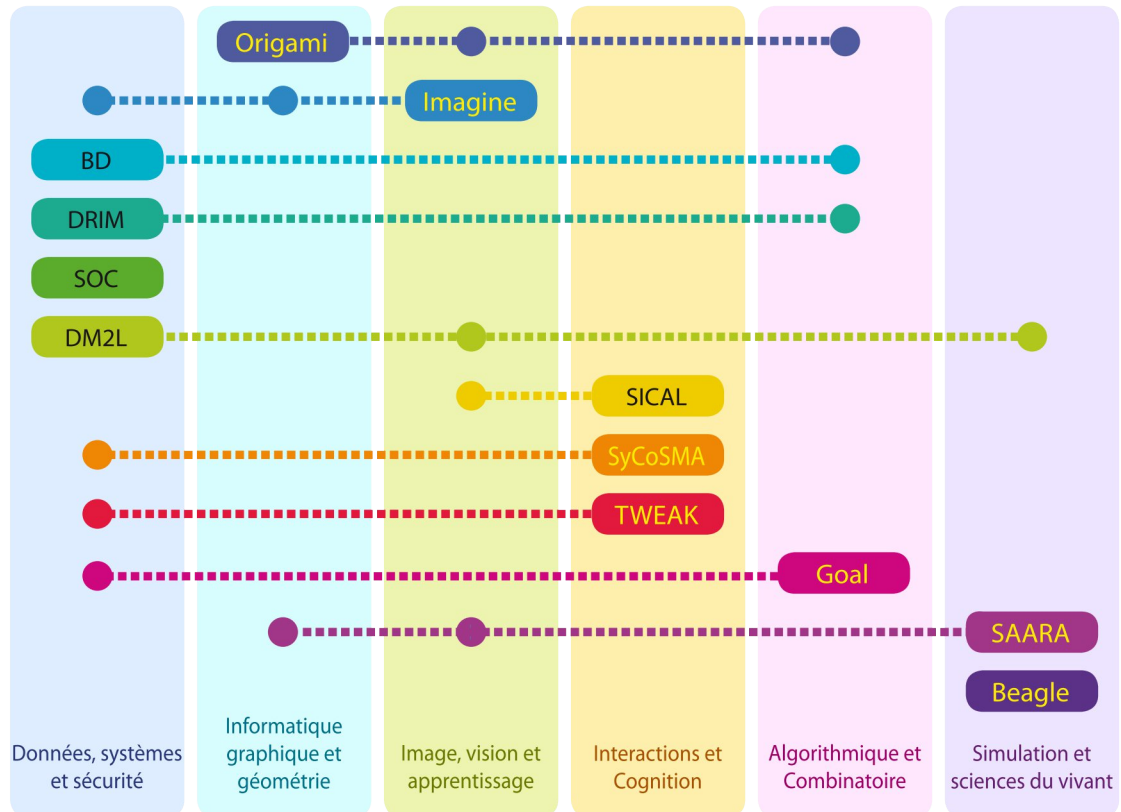


Figure 1 : Équipes et thématiques

Ce stage prend place au sein de l'équipe **Origami**, dont les thèmes de recherches gravitent autour de 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 portable, 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.

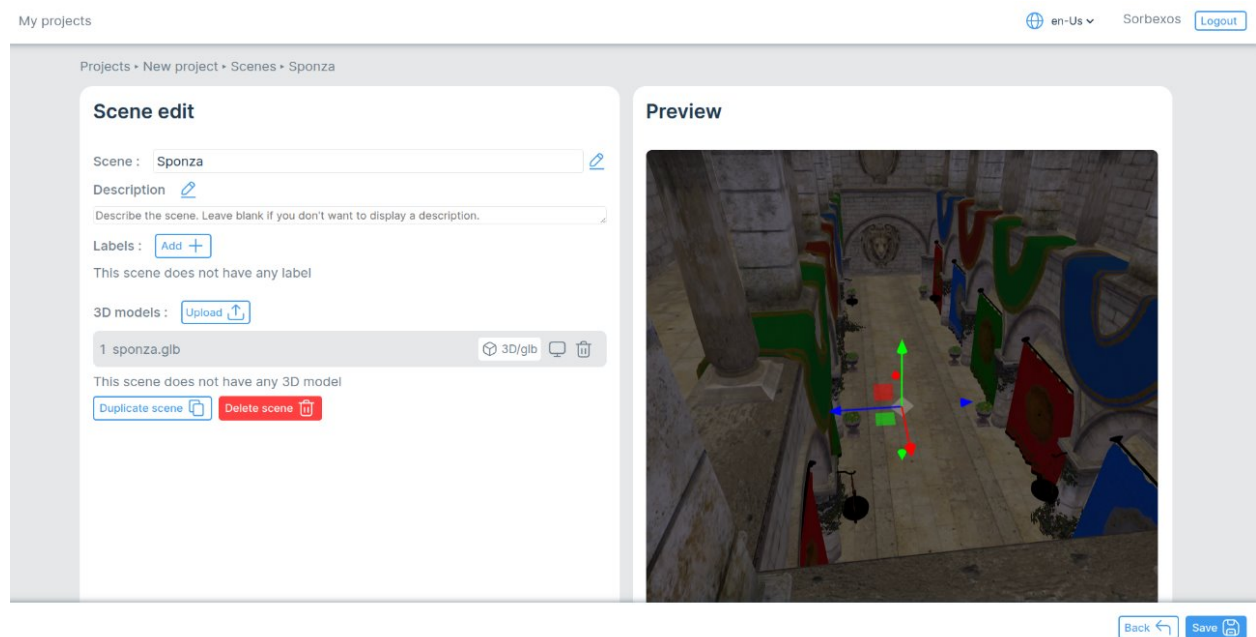


Figure 2 : Partie éditeur de HERA

Un **viewer**, permettant au visiteur de profiter de la visite. Il s'agit simplement de calibrer la scène en plaçant la caméra par rapport à un repère prévu, et ainsi profiter de la scène en réalité augmentée sur son appareil.

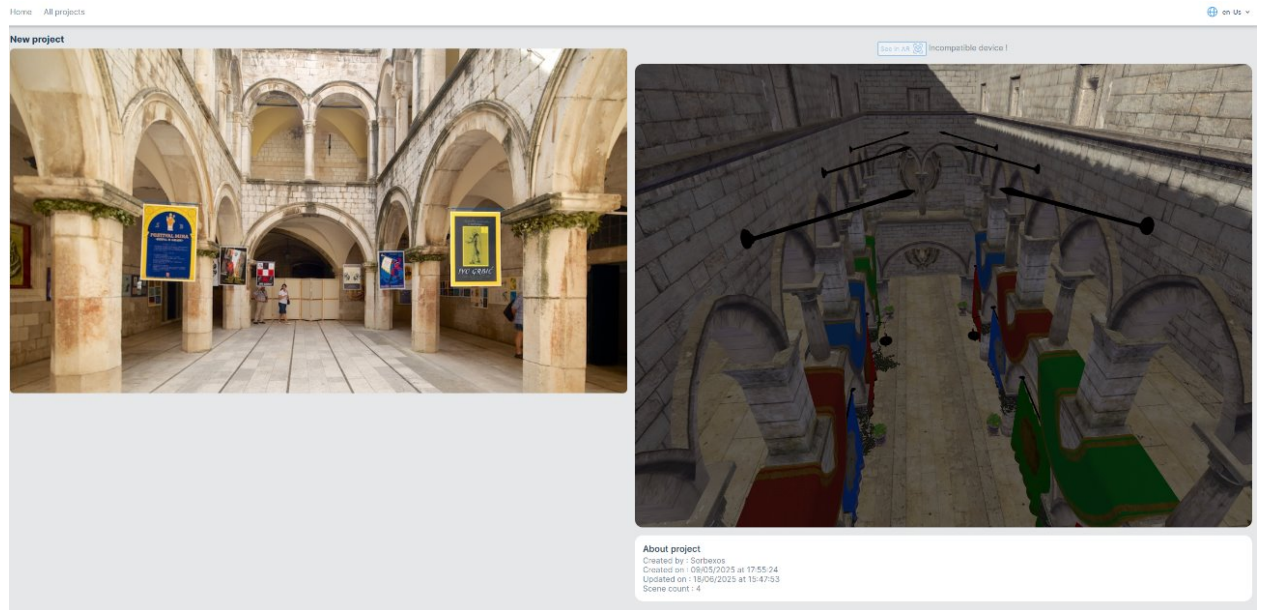


Figure 3: Partie viewer de HERA

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 présets (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.

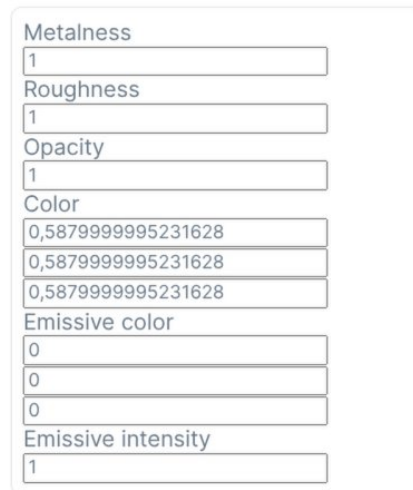
Pour ce qui est des outils informatique, j'ai principalement utilisé VScode, Chrome pour utiliser l'application, ainsi que Gitlab.

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.

Material



Metalness	1
Roughness	1
Opacity	1
Color	0,5879999995231628
	0,5879999995231628
	0,5879999995231628
Emissive color	0
	0
	0
Emissive intensity	1

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.

(rajouter des schémas explicatifs comme t'avais dessiné sur ton carnet)

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 6 : Pot de fleurs gris

Figure 5 : Pot de fleurs rouge

2.2 Intégration de l'éclairage 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](#) 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 plus légère que d'autres, restait trop lourde pour être utilisée sans GPU, et mettait trop de temps à converger.

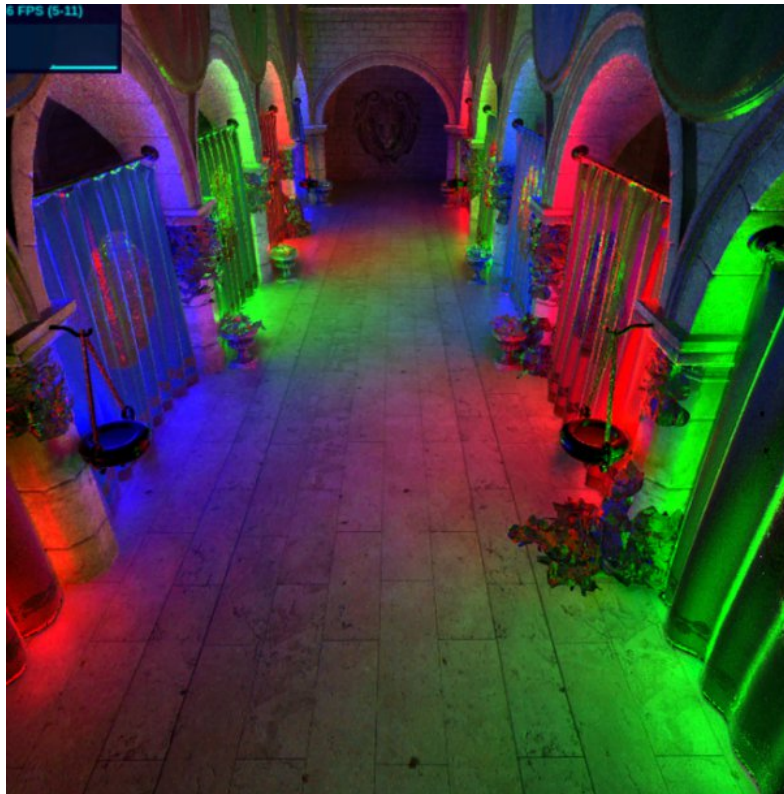


Figure Z : SSGI sur la Sponza

Cette image (**figure 7**) a convergé en ~30 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.

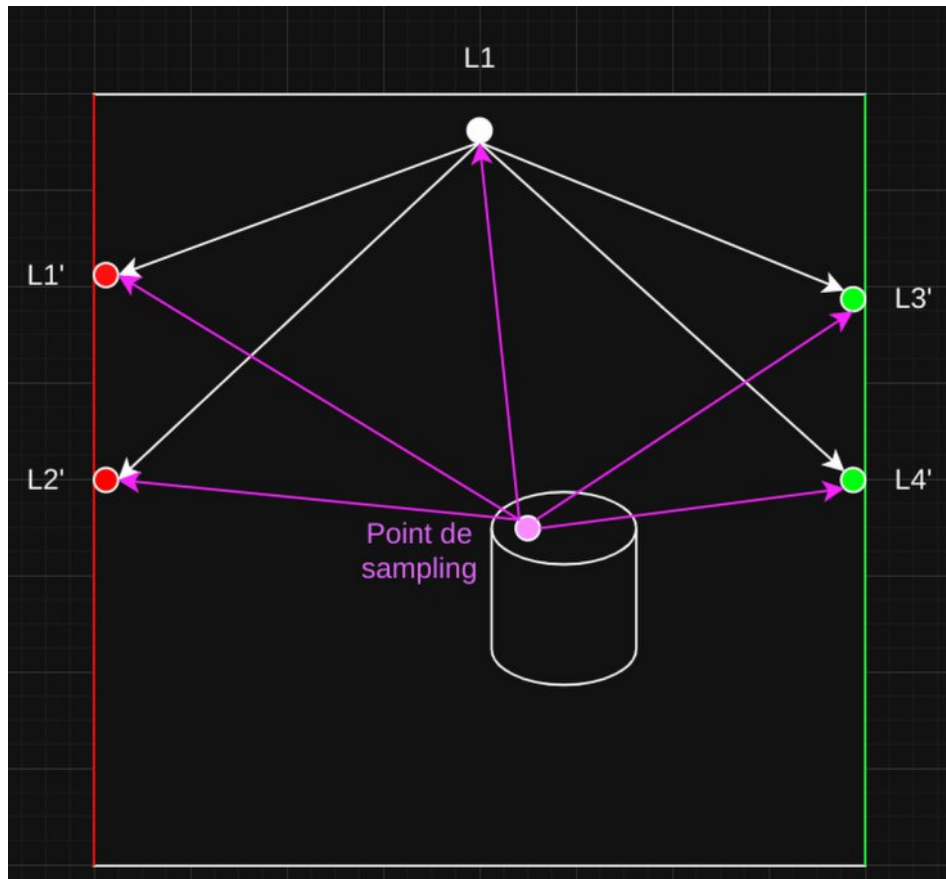


Figure 8 : Instant radiosity

Une fois nos lumières ponctuelles créées, 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.

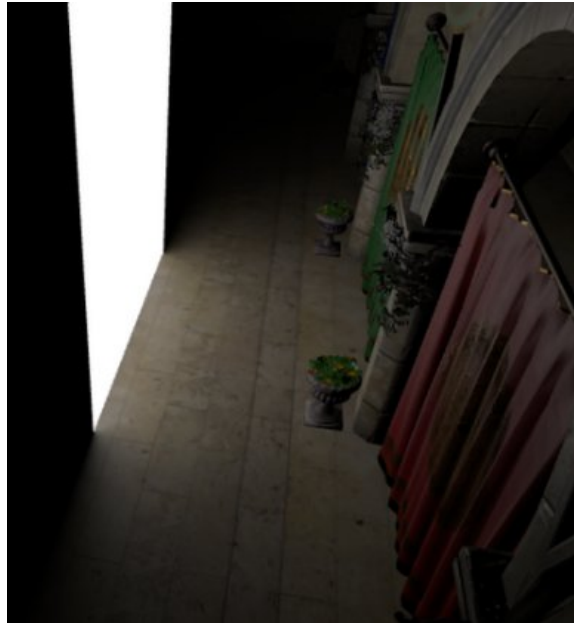


Figure 9 : Rect Area Light de ThreeJS

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.

Il y a deux problèmes principaux :

- L'utilisateur devra placer lui-même les area lights
- Pas d'ombre

Cette solution a donc été abandonnée, même s'il aurait été possible d'approximer de 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 des animations seront rajoutées à 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 correcte, 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.¶

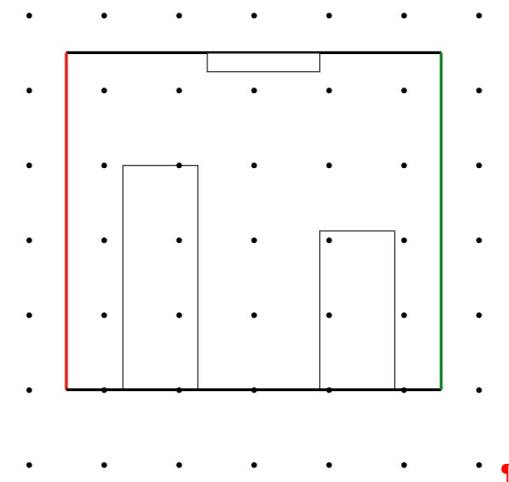


Figure 10 : Grille régulière de probes ¶

2.2.2.1 Baking¶

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

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 rajoute un offset aléatoire pour chaque probe pour éviter d'avoir le même sampling pour chaque probe.¶

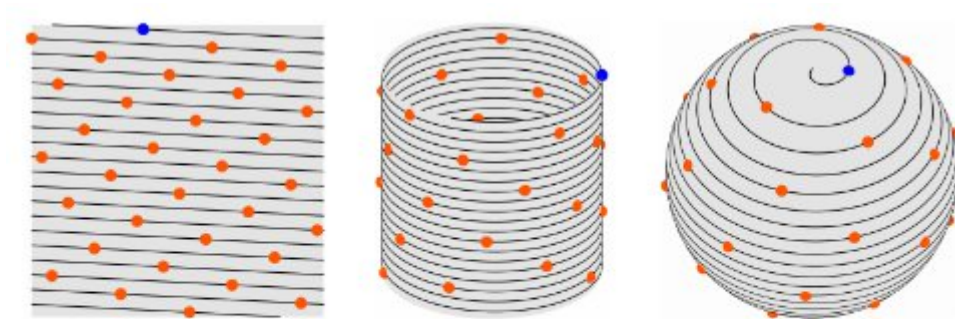


Figure 1: Illustration of the construction of the point set \mathbf{SF}_i^{32} , showing the mapping from the grid through a cylinder to the sphere. The first point of the point set is marked blue.

¶

Figure 11 : Spirale de Fibonacci^{1¶}

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 **invalidé**.

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)¶

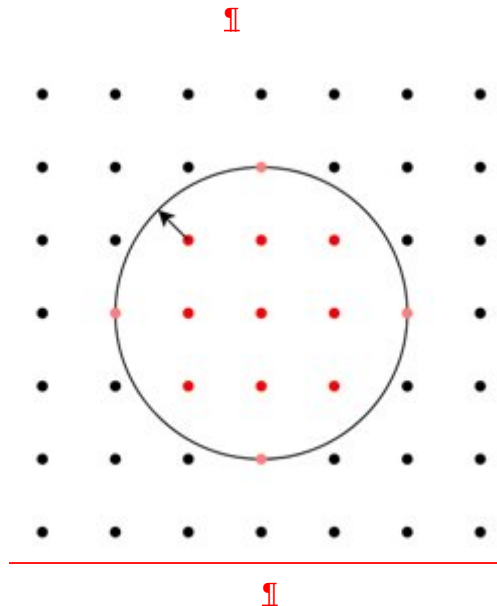


Figure 12 : Déplacement des probes invalides ¶

On calcule un score d'invalidité : chaque rayon qui touche une backface augmente ce score.^{2¶}

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. ¶

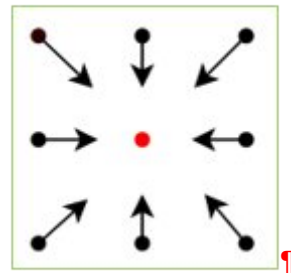


Figure 13 : Dilatation ¶

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.¶

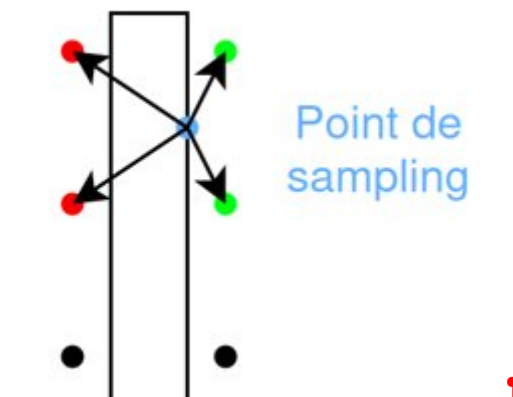


Figure 14 : Occultation des probes ¶

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^{3,4} 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)¶

¶

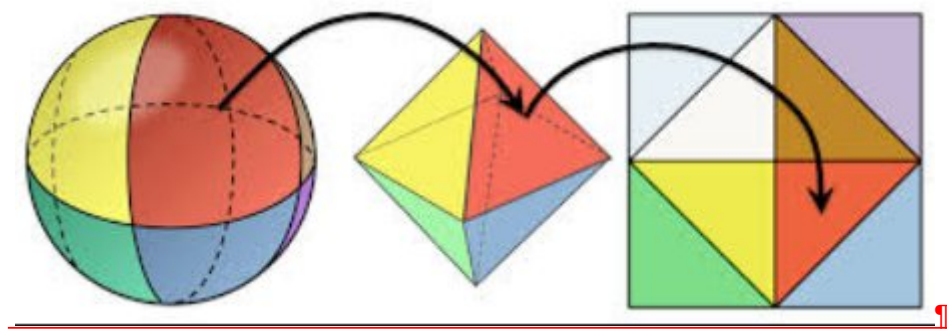


Figure 15 : Octahedral mapping ¶

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.¶

¶

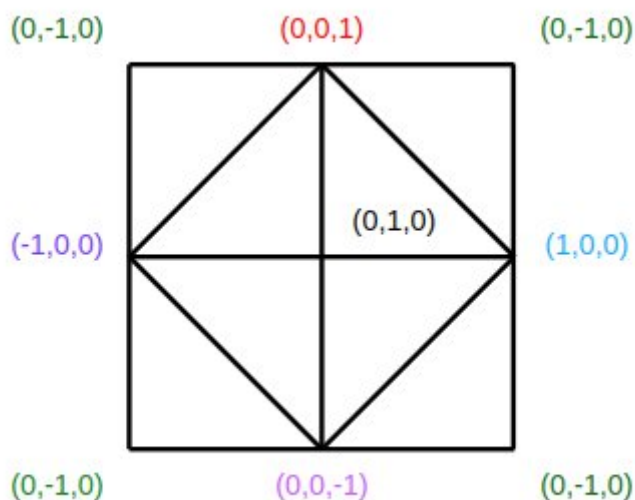


Figure 17 : Directions de l'octaèdre

¶

¶

¶

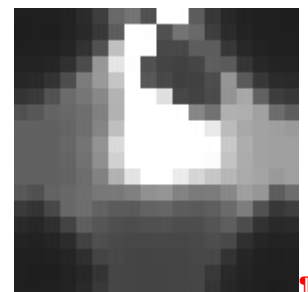


Figure 16 : Exemple d'octmap ¶

¶

Maintenant qu'on a nos directions, on peut remplir nos octmap en lançant des rayons. Pour se 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 (.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.¶

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.

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 lors du requêtage des depth maps.¶

¶

¶

2.2.3 Résultats

3. Conclusion

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>

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>

¶

Table des figures

Figure 1 : Équipes et thématiques	4¶
Figure 2 : Partie éditeur de HERA	5¶
Figure 3 : Partie viewer de HERA	6¶
Figure 4 : Menu d'édition de matériaux	8¶
Figure 5 : Pot de fleurs rouge	9¶

Figure 6 : Pot de fleurs gris.....	9
Figure 7 : SSGI sur la Sponza	10
Figure 8 : Instant radiosity	11
Figure 9 : Rect Area Light de ThreeJS	12
Figure 10 : Grille régulière de probes	13
Figure 11 : Spirale de Fibonacci ¹	14
Figure 12 : Déplacement des probes invalides	14
Figure 13 : Dilatation	15
Figure 14 : Occultation des probes	15
Figure 15 : Octahedral mapping	16
Figure 16 : Exemple d'octmap	16
Figure 17 : Directions de l'octaèdre	16

Annexes