Rapport de projet
GemStudio – IN55 P2019

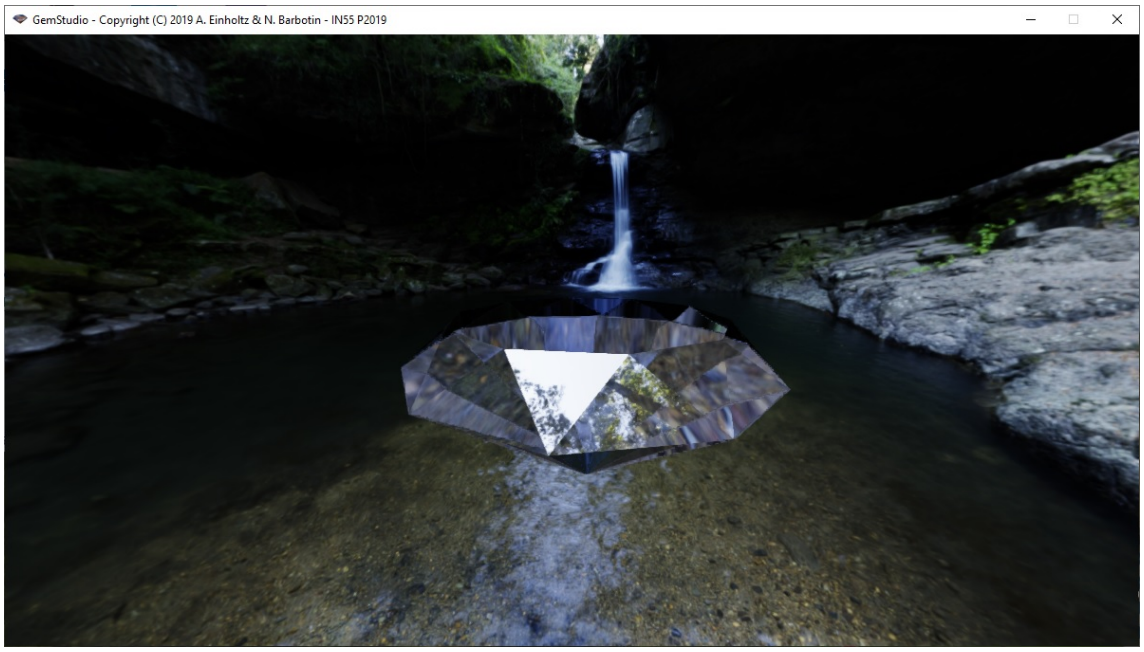


Table des matières

1 Introduction

1.1 Présentation du projet

Dans le cadre de l'UV IN55 lors du semestre de printemps 2019, nous avons été chargés de réaliser un projet basé sur OpenGL. Le sujet que nous avons sélectionné visait à faire le rendu de pierres précieuses en temps réel. Pour cela, nous avons utilisé C++ avec les bibliothèques GLFW pour la gestion de la fenêtre et GLEW pour le chargement d'OpenGL. Les mathématiques et l'interface graphique sont gérées par des bibliothèques maisons, soit MGPCL et AISO.

1.2 Utilisation

Lorsque *GemStudio* est lancé, l'utilisateur est accueilli avec le rendu d'une gemme en fond, et deux fênetres. La première permet de contrôler la vue et la seconde permet de modifier l'apparence de la gemme. Ces fenêtres peuvent être fermées pour profiter de la vue, mais peuvent être réouvertes à tout moment à l'aide des touches F1 et F2.

1.2.1 Fenêtre de vue

Les paramètres ajustables via cette fenêtre sont les suivants :

- Champ de vision : il s'agît du FOV
- Vitesse d'adaption de l'exposition : permet de contrôler la vitesse à laquelle l'œil s'adapte à la luminance de la scène
- Seuil de bloom : luminance à partir de laquelle l'utilisateur est considéré comme ébloui, résultant en un flou léger
- Vitesse de la caméra : vitesse à laquelle la caméra orbitale tourne autour de la gemme
- Effets : permet d'activer ou de désactiver les effets utilisés pour le rendu
- Affichage : propose de contrôler les IPS et d'afficher des informations de débogage
- Changer la skybox : lorsque les environment maps supplémentaires ont été téléchargées, permet de passer à l'environment map suivante

1.2.2 Fenêtre de customisation

Cette fenêtre permet d'ajuster l'apparence de la gemme :

- Couleur : permet de changer la couleur de la gemme
- Indice de réfraction : permet de changer l'indice de réfraction
- N : nombre de côtés de la gemme
- H : hauteur de la partie inférieure
- R : rayon de l'interface inférieure/supérieure

- dH : surélévation de la partie supérieure
- dR : rapport entre R et le rayon de la face du haut

1.2.3 Scène

En plus de pouvoir changer la vitesse de la caméra, l'utilisateur peut contrôler celle-ci. En effet, il peut utiliser (en dehors des fenêtres) la molette de sa souris pour changer l'altitude de la caméra (la distance à laquelle celle-ci orbite autour de la gemme), ainsi que la longitude et la latitude en bougeant la souris tout en gardant le clic gauche enfoncé. Il peut aussi utiliser la touche C pour passer de la caméra orbitale à la caméra libre, et vice-versa. Cette dernière se contrôle avec les touches ZQSD pour la déplacer, et la souris (aussi en maintenant le clic gauche enfoncé) pour l'orienter.

1.3 Répartition des tâches

Amaury s'est chargé la boucle principale, la gestion de la fenêtre, des évènements, et des objets du jeu (dont la gemme et le procédé utilisé pour la générer). Nicolas s'est concentré sur les effets visuels : réfraction et réflection, exposition automatique, etc. . .

1.4 Choix techniques

Deux effets principaux interviennent dans le visuel des pierres précieuses : la réflection et la réfraction. Ces effets étant difficiles à mettre en œuvre en temps réel (c'est-à-dire, sans raytracing), nous avons décidé de se limiter à un seul et unique objet dans toute la scène : la pierre précieuse. Celle-ci est placée dans un décor reposant sur une *environment map* ; il s'agît d'une texture qui est considérée comme « à l'infini » par rapport à la caméra. Cela permet d'accélérer de manière importante le rendu et permet d'atteindre une centaine d'images par seconde sur une Intel HD Graphics 620, toute en gardant une qualité décente.

En plus de la réflection et de la réfraction, nous avons décidé d'implémenter les effets suivants :

- Pipeline *High-Dynamic Range* (HDR), pour assurer la qualité des réflections
- Exposition automatique : contrôle automatiquement la luminosité de la scène de manière à percevoir un maximum de couleurs différentes
- Bloom : effet d'éblouissement lorsque la caméra est exposée à une lumière importante
- Lens flare : imitation des artéfacts visuels produit par les imperfections des lentilles, perçus généralement lorsque le soleil fait directement face à la caméra
- Fast Approximative Anti-aliasing (FXAA) pour réduire « l'aliasing »

2 Fonctionnement

2.1 Classes C++

Le projet est séparé en 10 classes principales :

2.1.1 MainApp

C'est elle qui gère la fenêtre, les évènements (position de la souris, entrée clavier, etc...), tout le rendu de la scène (skybox, objets du jeu, etc...), et la pile de matrices modèles. Pour ce faire, elle fait appel à l'entièreté des classes décrites dans cette section. Elle gère aussi l'IHM, mais nous avons choisi de ne pas détailler cette partie dans ce rapport.

2.1.2 Camera, FreeCamera et RotatingCamera

La classe Camera est une interface utilisée pour décrire une caméra ; c'est-à-dire un objet capable de déterminer la matrice de vue. En plus de cette matrice, l'interface Camera se doit de retourner un vecteur contenant la position de ladite caméra, et ce afin de pouvoir toujour placer le centre de la skybox en cette position.

Le projet propose deux types de caméras :

- FreeCamera : la caméra libre habituelle, que l'on peut contrôler avec la souris et les touches ZQSD
- RotatingCamera : une caméra orbitale qui tourne toute seule autour de l'origine. L'utilisateur peut en prendre le contrôle afin d'ajuster l'alititude, la longitude et la latitude à sa guise

2.1.3 GameObject et Gem

GameObject est une interface représentant un objet du jeu ; elle avait été créée dans l'optique de pouvoir supporter différents objets dans une même scène, mais au final nous n'avons fait qu'un seul objet : la gemme.

La classe Gem implémente GameObject et est chargée de générer, selon quelques paramètres, les vertices et les indices nécéssaires au rendu d'une gemme. 5 paramètres sont supportés lors de la génération :

- Le nombre de côtés
- La hauteur de la partie inférieure
- Le rayon de la séparation inférieure/supérieure
- La hauteur de la partie supérieure
- Le rayon de la face du haut

La couleur peut, quant à elle, être changée via la méthode `changeColor`.

2.1.4 Skybox

Cette classe gère 4 choses :

- Le chargement d'une environment map HDR au format RGBE
- La texture associée à cette environment map (pour la skybox)
- La cubemap associée à cette environment map (pour la réflection et la réfraction)
- Les vertices de la skybox (c'est-à-dire, les vertices d'un cube texturé)

Le chargement de l'environment map étant très long (1-5 secondes), la classe Skybox propose deux moyens de chargement : synchrone et asynchrone. Une skybox étant nécéssaire pour faire fonctionner l'application, la première environment map est chargée de manière synchrone au démarrage de l'application. Les suivantes sont chargée de manière asynchrone pour limiter le gel de l'écran.

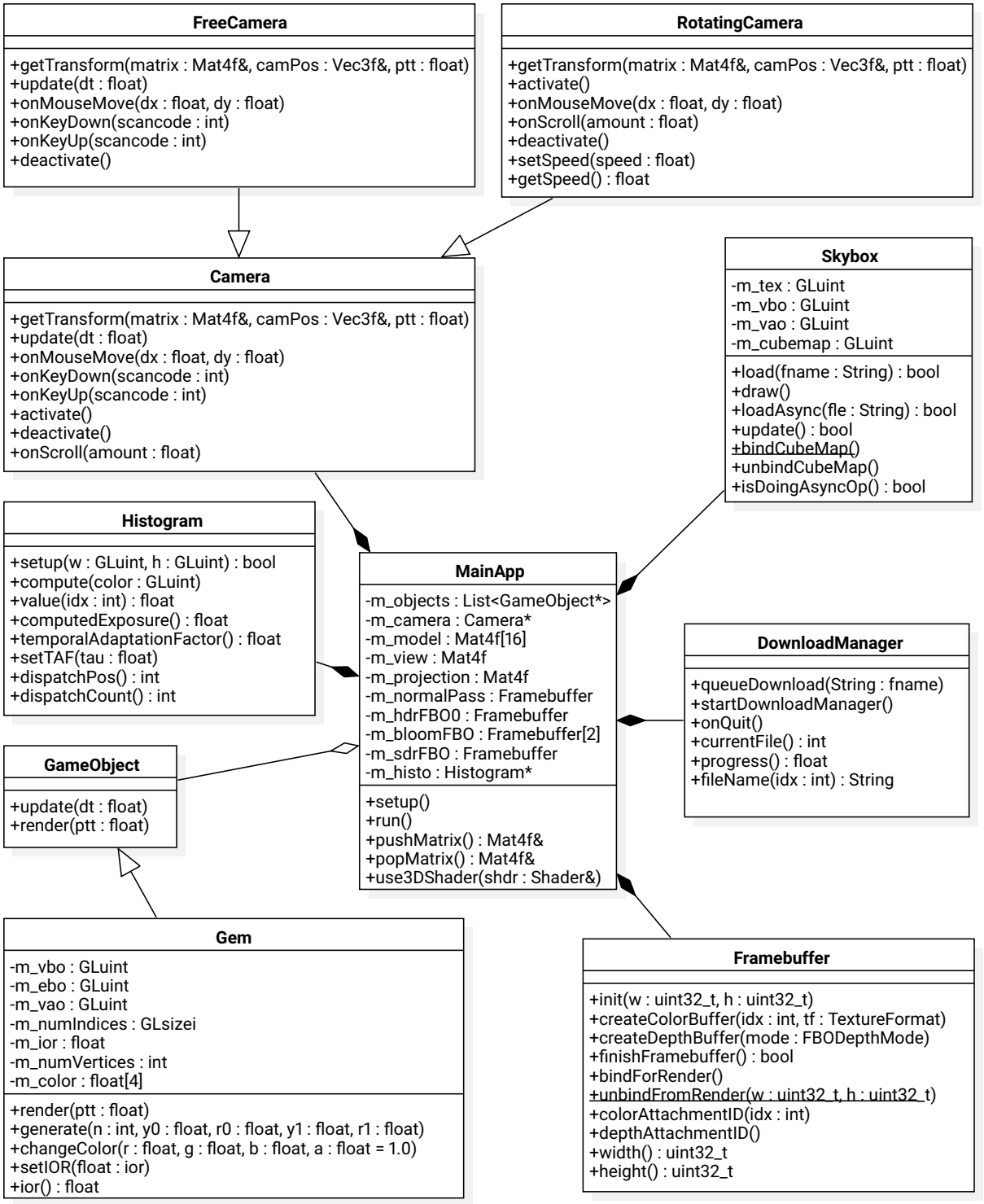
2.1.5 Autres classes

Les classes restantes sont les suivantes :

- Framebuffer : une classe utilitaire pour gérer la création de Framebuffer OpenGL
- DownloadManager : sert à gérer le téléchargement des environment map. Celles-ci étant très lourdes (24 Mo chacune), nous avons choisi de n'en fournir de base qu'une seule de basse qualité. Lors du premier lancement de l'application, cette classe s'occupe justement de télécharger les autres
- Histogram : gère tout ce qui concerne l'exposition automatique : le calcul de l'histogramme avec des *compute shaders*, puis le calcul de l'exposition de la prochaine trame

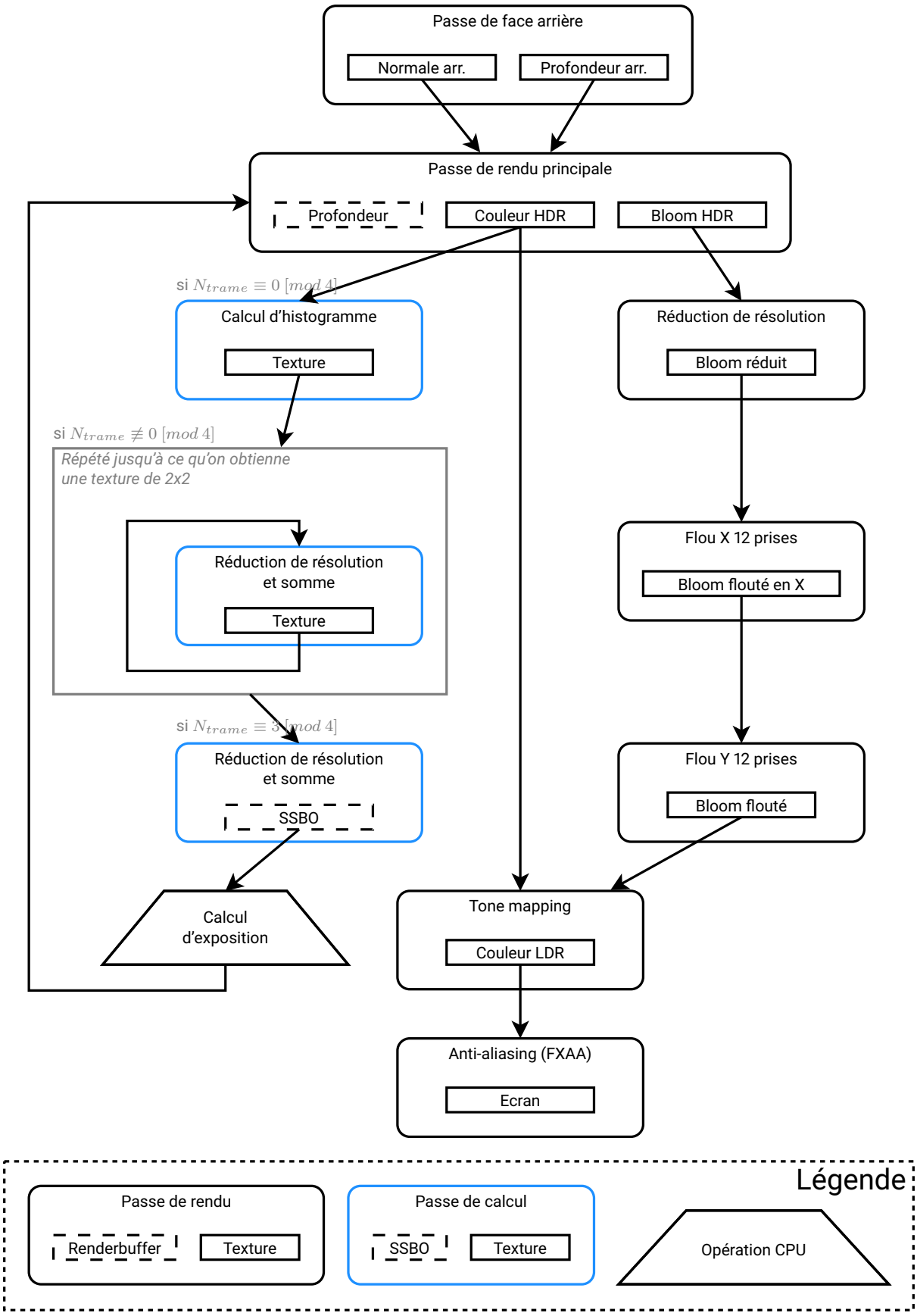
2.1.6 Diagramme de classe

Voici le diagramme de classe résumant le tout :



2.2 Pipeline de rendu

Le rendu de la scène est réalisé avec la pipeline suivante :


Passe de face arrière
Normale arr.
Profondeur arr.
Passe de rendu principale
Profondeur
Couleur HDR
Bloom HDR
si $N_{trame} \equiv 0 \ [mod\ 4]$
Calcul d'histogramme
Texture
Réduction de résolution
Bloom réduit
si $N_{trame} \neq 0 \ [mod\ 4]$
Répété jusqu'à ce qu'on obtienne une texture de 2x2
Réduction de résolution et somme
Texture
Flou X 12 prises
Bloom flouté en X
si $N_{trame} \equiv 3 \ [mod\ 4]$
Réduction de résolution et somme
SSBO
Flou Y 12 prises
Bloom flouté
Calcul d'exposition
Tone mapping
Couleur LDR
Anti-aliasing (FXAA)
Ecran
Légende
Passe de rendu
Renderbuffer
Texture
Passe de calcul
SSBO
Texture
Opération CPU

2.2.1 Passe de rendu principale

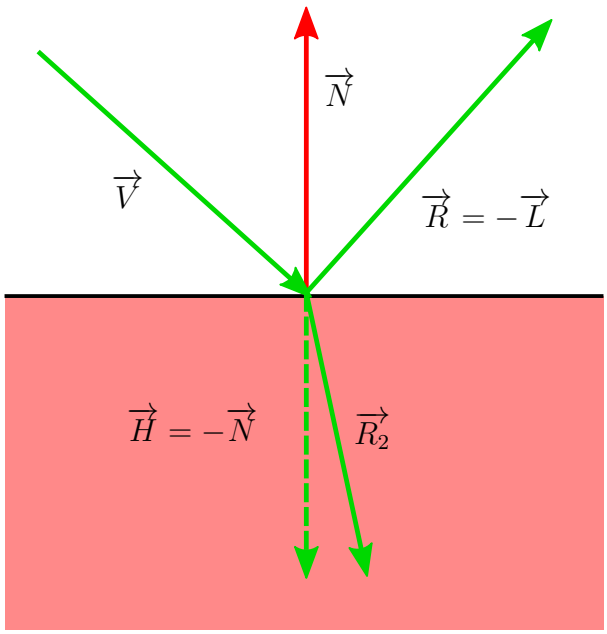
La passe de rendu principale est celle durant laquelle les éléments visuels principaux sont dessinés ; c'est-à-dire la *skybox* et la gemme. Comme il s'agît d'une passe de rendu HDR, cette passe de rendu prend en compte l'exposition déterminée par le système d'exposition automatique, décrit plus bas. Enfin, cette passe de rendu génère en plus du color buffer habituel, un second buffer HDR qui sera plus tard utilisé pour le *bloom* (l'effet d'éblouissement). Ce buffer est princialement noir, sauf aux endroits où la couleur dépasse un seuil de luminance défini par l'utilisateur. Dans ce cas, le buffer de bloom contient la même valeur que le color buffer. Ceci se résume par le code GLSL suivant :

```
float luma = dot(final, vec3(0.2126, 0.7152, 0.0722));

if(luma > u_BloomThreshold)
    out_Bloom = vec4(final, 1.0);
else
    out_Bloom = vec4(0.0, 0.0, 0.0, 1.0);
```

La *skybox* est un simple cube texturé avec l'*environment map* décrite plus haut. Normalement, les faces de la skybox doivent être placées à une distance infinie de la caméra. Comme nous ne pouvons pas vraiment faire cela, nous simulons cet effet en placant le centre du cube à la position de la caméra, et en désactivant l'écriture dans le buffer de profondeur (`glDepthMask`). Ainsi, tout objet déssiné après la skybox, qu'il soit devant ou derrière celle-ci, sera tout de même déssiné à l'écran. Puisque c'est elle qui émet la lumière de toute la scène, le dessin de ce cube est fait sans aucun calcul d'illumination.

La gemme, en revanche, subit quelques calculs d'illumination. Les deux effets à prendre en compte sont (comme dit précédemment) la réflexion et la réfraction :



On part du vectur de vue $\overrightarrow{V}$ (vecteur partant du centre de la caméra et pointant vers le fragment), du vecteur normal $\overrightarrow{N}$ et de l'indice de réfraction η_{gemme}. On en déduit le vecteur réfléchi $\overrightarrow{R}$ et réfracté $\overrightarrow{R_2}$, avec lesquels on lit la cubemap pour trouver les couleurs correspondantes. Les deux couleurs sont ensuite mixées ensemble selon le coefficient de fresnel. Celui-ci dépend de l'indice de réfraction et du produit scalaire $\overrightarrow{L} \cdot \overrightarrow{H}$. Or, on constate dans notre cas que le vecteur de direction de la lumière $\overrightarrow{L}$ est égal à $-\overrightarrow{R}$ (puisque la lumière vient de l'environment map). On constate aussi que :

$$\overrightarrow{H} = \frac{\overrightarrow{V} + \overrightarrow{L}}{\left\|\overrightarrow{V} + \overrightarrow{L}\right\|} = \frac{\overrightarrow{V} - \overrightarrow{R}}{\left\|\overrightarrow{V} - \overrightarrow{R}\right\|} = \ldots = -\overrightarrow{N}$$

Ce qui au final nous donne :

$$\overrightarrow{L} \cdot \overrightarrow{H} = (-\overrightarrow{R}) \cdot (-\overrightarrow{N}) = \overrightarrow{R} \cdot \overrightarrow{N}$$

Tout cela se traduit par le shader suivant :

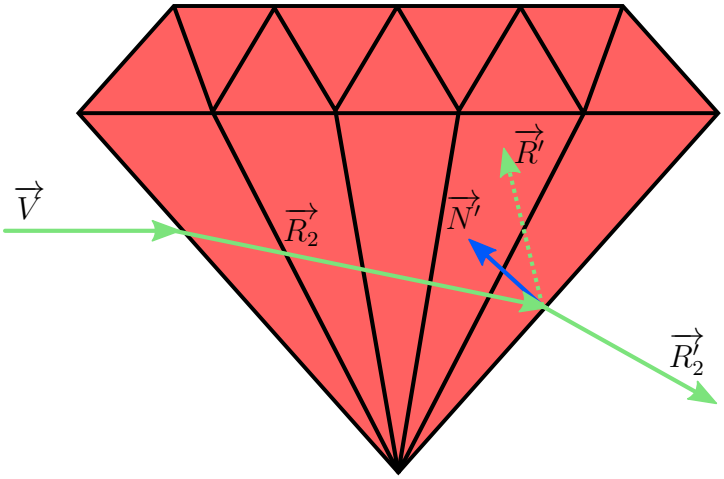
```
vec3 N  = normalize(f_Normal);
vec3 V  = normalize(f_WorldPos - u_CamPos);
vec3 R  = normalize(reflect(V, N));
vec3 R2 = normalize(refract(V, N, 1.0 / u_IOR));

float fresnel = computeFresnel(u_IOR, max(0.0, dot(R, N)));
vec3 diffuse  = texture(u_CubeMap, R2).rgb * f_Color.rgb;
vec3 specular = texture(u_CubeMap, R).rgb;
vec3 final    = mix(diffuse, specular, fresnel) * pow(2.0,
   ↪ u_Exposure);

out_Color = vec4(final, 1.0);
```

2.2.2 Réfraction interne et passe de face arrière

Le *shading* présenté dans la section précédente est correct, mais présente un défaut important : la réfraction ne se produit que sur la face avant. En réalité, le rayon réfracté $\overrightarrow{R_2}$ devrait continuer à l'intérieur de l'objet avant de se « heurter » à la face arrière, sur laquelle il est à nouveau réfléchi et réfracté. Le nouveau rayon réfléchi risque de « rebondir » encore quelques fois dans la pierre, et simuler cet effet serait difficile. Le nouveau rayon réfracté en revanche sort de l'objet, et il suffit alors d'utiliser la cubemap pour trouver la bonne couleur à afficher.



On cherche donc à calculer $\overrightarrow{R'_2}$. Pour cela, on réalise au préalable le rendu d'une texture de profondeur ainsi que d'une texture de normales constituées des faces arrières (`glCullFace(GL_FRONT)`) de la gemme. Ceci nous permettra d'obtenir la profondeur (en espace écran) ainsi que la normale (en espace monde) de tout point sur l'écran. Les étapes ci-dessous sont alors ajoutées au shader utilisé pour le rendu de la gemme :

- Passer $\overrightarrow{R_2}$ en espace écran
- Partir des coordonnées $\overrightarrow{P}$ du fragment en cours de calcul
- Ajouter $\overrightarrow{R_{2,ecran}} \times pas$ à $\overrightarrow{P}$
- Faire de même jusqu'à ce que $P_z \geq \mathcal{D}(P_x, P_y)$ ($\mathcal{D}$ étant la profondeur de face arrière, retrouvée à partir de la texture générée tout à l'heure)
- A cette étape, on dispose des coordonnées espace écran de l'intersection entre $\overrightarrow{R_2}$ et la face arrière
- Retrouver $\overrightarrow{N'} = -\overrightarrow{\mathcal{N}}(P_x, P_y)$ avec $\mathcal{N}$ la normale de la face arrière
- Calculer $\overrightarrow{R'_2}$ avec $\overrightarrow{R_2}, \overrightarrow{N'}$ et η_{gemme}

Reste à déterminer un pas convenable de manière à ne pas sauter trop de pixels et ne pas être trop gourmand en ressources, mais le résultat est convaincant :

2.2.3 Bloom

Le bloom se calcule avec 3 passes de rendu. La première prend en entrée la texture de bloom HDR générée par la passe de rendu principale, et en génère une version basse résolution. La seconde réalise un flou gaussien 12 prises horizontal, et la dernière un même flou mais cette fois sur l'axe vertical. Le résultat sera ensuite combiné à la couleur HDR lors de la passe de tone mapping.

2.2.4 Tone mapping et anti-aliasing

Il s'agît maintenant de convertir les couleurs HDR en couleurs LDR ($\mathbb{R}_+^3 \rightarrow [0;1]^3$), affichables à l'écran. Pour cela, on utilise un *tone mapping*. Il existe une multitude de tone mapping, chacun adapté à une scène différente. Le meilleur tone mapping est celui ajusté par un artiste spécifiquement pour la scène ; mais comme aucun de nous n'a fait les beaux-arts, nous avons juste implémenté un tone mapping sous le nom de *ACES Filmic Tone Mapping*, trouvé sur le blog de Krzysztof Narkowicz.

La dernière étape de rendu est une passe d'anti-aliasing. En l'occurrence il s'agît de FXAA, *Fast Approximative Anti-Aliasing*. C'est un anti-aliasing très basique sous forme de filtre. Il produit un faible flou au niveau des bords, mais est néanmoins très rapide.

2.2.5 Exposition automatique

L'exposition automatique permet d'ajuster l'exposition de la scène de manière à maximiser les détails affichés à l'écran. Si la scène est sous-exposée, les endroits un peu sombre apparaîtrons noir. Si elle est sur-exposée, les endroits bien éclairés seront trop blanc pour y distinguer quelque-chose. Lorsque la scène présente à la fois des zones sombres et claires (par exemple : salle sombre avec fenêtre qui donne vers l'exterieur), il faut choisir la bonne exposition. Cela se fait instinctivement pour un humain. Dans un rendu 3D, en revanche, cela requiert un peu de calcul. Voici la manière dont nous procédons :

- On calcul un histogramme d'indice de lumination à partir du rendu actuel
- On enlève un pourcentage des pixels les plus lumineux
- On enlève un pourcentage des pixels les plus sombres
- On calcul la luminance moyenne de ce qu'il reste
- On en déduit une nouvelle exposition pour la trame suivante
- Pour éviter les changements brusques, et pour simuler le changement progressif de l'œil, on fait varier en exponentielle inverse l'exposition

3 Conclusion

A travers ce projet nous avons pu répondre à la problématique de génération et rendu d'une pierre précieuse en temps réel, et ce en utilisant OpenGL. Ce projet nous aura permis bien évidemment de s'accommoder avec cette bibliothèque graphique, mais

aussi d'en apprendre beacoup sur la synthèse d'image photoréaliste de par les différents effets que nous avons pu implémenter.

3.1 Améliorations possibles

Notre projet pourrait cependant être amélioré, notamment sur les points suivants :

- Implémenter *Interactive Image-Space Refraction of Nearby Geometry*[1] pour pouvoir dessiner plusieurs objets
- Accélérer la recherche de profondeur dans le shader de la gemme en utilisant un algorithme de recherche dichotomique
- Utiliser des couleurs sRGB
- Implémenter un meilleur anti-aliasing (comme le *Temporal Anti-Aliasing*, bien plus efficace que le FXAA)
- Plusieurs rebonds dans le shader de la gemme

3.2 Bibliographie

- *Mathematics for 3D Game Programming and Computer Graphics, Third Edition*, ISBN 978-1435458864
- *ACES Filmic Tone Mapping Curve*, Krzysztof Narkowicz, https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/
- *Automatic Exposure*, Krzysztof Narkowicz, https://knarkowicz.wordpress.com/2016/01/09/automatic-exposure/

1. Chris Wyman, 2005, Interactive Image-Space Refraction of Nearby Geometry