Master GIG — Géométrie et Informatique Graphique
Rapport

Implantation d'un modèle de rendu physiquement réaliste en OpenGL 3.3

ANDRE Jérémy[1], CLERC Billy[1] et COZZOLINO Christine[1]

[1]Aix-Marseille Université

Résumé
À travers ce rapport, nous détaillons le processus complet nécessaire à la création d'un modèle de rendu physiquement réaliste en OpenGL .3.
Pour ce faire, nous commencerons par analyser et l'expliquer les concepts liés à un moteur de rendu basé sur la physique. Avec ces informations en tête, nous définirons de manière théorique les différentes notions physique que l'on souhaite répliquer. Ensuite, il s'agira de présenter notre travail, à la fois à travers l'étude de la structure de notre code et des différents éléments qui le compose, mais aussi via des schémas, des diagrammes et des extraits de programmes informatiques. Enfin, nous terminerons par une étude des performances de notre moteur de rendu ainsi qu'une introspection sur le travail accompli, les difficultés et perspectives futures.

Mots clé : Modèle de rendu, Physically Based Rendering, OpenGL 3.3.

1. Introduction

Il est ici question d'implanter, en C++ et à l'aide de shaders écrit en GLSL 3.3, un modèle de rendu physiquement réaliste – que l'on nommera de par son acronyme anglais PBR, pour *Physically Based Rendering*.

Pour qu'un modèle de rendu puisse être considéré comme PBR, c'est-à-dire une approximation de la réalité qui se base sur les principes de la physique, il doit :

- reposer sur un système de surface à base de microfacette,
- conserver l'énergie,
- utiliser une BRDF (*Bidirectional Reflective Distribution Function* – Fonction de Distrubtion de Réflectance Bidirectionnelle), basée sur un modèle physique.

Il est important de noter qu'une bonne partie de la théorie et du code provient de **Learn OpenGL**, une introduction à l'OpenGL moderne de très grande qualité, disponible à la fois en ligne [dV] ou en livre [dV20].

Microfacettes. Lorsqu'on parle d'un système de surface à base de microfacette, on décrit le fait que n'importe quelle surface, à un niveau microscopique, est composée de minuscules miroirs, parfaitement réflectifs, que l'on appelle des microfacettes.
Lorsqu'une surface est rugueuse, les rayons réfléchis sont éparpillés dans toutes les directions (figure 1), ce qui entraîne une zone de luminosité plus grande, mais peu intense (figure 2). En revanche, lorsqu'une surface est lisse, les rayons de lumière sont majoritairement réfléchis dans la même direction, ce qui établi une petite zone de luminosité, mais bien plus intense.

Figure 1: *Direction de réflexion des rayons de lumière en fonction de la nature de la surface.*

Conservation de l'énergie. Ce que l'on identifie par conservation de l'énergie, c'est le principe que toute l'énergie lumineuse qui sort d'un objet ne dois jamais être supérieure à l'énergie lumineuse qui entre dans ce même objet (à l'exception des surfaces émissives).
En effet, à partir du moment où un rayon de lumière touche une surface, il est divisé en deux parties : une partie **réfléchie** et une partie **réfractée**.
La partie réfléchie n'entre pas au sein de la surface et constitue ce que l'on appelle la **lumière spéculaire**. En revanche, la partie réfractée, elle, pénètre la surface et se disperse dans des directions à peu près aléatoires, entrant constamment en collision avec d'autres particules, jusqu'à ce que l'énergie devienne nulle ou que le rayon ressorte de la surface ; c'est ce que l'on appelle la **lumière diffuse**.
Ce principe est présenté à travers la figure 3.

BRDF. Une BRDF est une équation qui estime la proportion

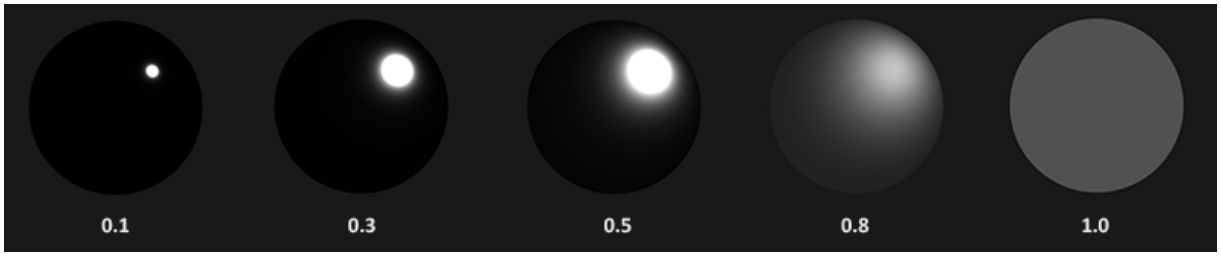

Figure 2: *Luminosité réfléchie en fonction d'un paramètre de rugosité de la surface variant entre 0 et 1.*

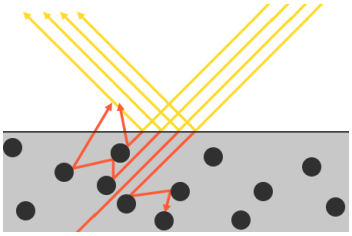

Figure 3: *Réflexion et réfraction des rayons de lumière à partir du moment où ils entrent en contact avec une surface.*

de chaque rayon de lumière dans la réflexion globale d'une surface opaque, en fonction de ses propriétés matérielles. Par exemple, si une surface est parfaitement lisse, comme un miroir, la fonction BRDF retournerai une valeur de 0.0 pour tous les rayons de lumières entrant, sauf pour le rayon de lumière qui possède exactement le même angle de réflexion que le rayon de lumière sortant, qui pour lequel la fonction retournerai 1.0.

2. Travaux relatifs

2.1. L'équation de réflectance

On définit l'équation de réflectance – c'est-à-dire le modèle qui permet de simuler les propriétés de la lumière – comme suit :

$$L_o(p,\omega_o) = \int_\Omega f_r(p,\omega_i,\omega_o) L_i(p,\omega_i) n \cdot \omega_i d\omega_i \qquad (1)$$

On définit la **radiance** comme la quantification de la magnitude d'une intensité lumineuse provenant d'une seule direction.
Cette équation additionne la radiance L_i de tous les rayons de lumières entrant ω_i à travers son aire Ω vers un point p, échelonné par des principes que nous allons détailler ci-dessous et nommés pour le moment f_r, puis retourne la somme de la lumière réfléchie L_o à ce même point p dans la direction du spectateur ω_o.
Il est important de noter que la radiance entrante peut provenir de sources de lumière directes, comme elle peut provenir de cartes d'environnements, que nous aborderons par la suite.

La seule inconnue restante de l'équation précédente (1) est le facteur f_r, c'est la BRDF. L'une des BRDF les plus utilisée dans le domaine des PBR, celle que l'on utilisera, est la **BRDF de Cook-Torrance** [CT81], définie comme suit :

$$f_r = k_d f_{lambert} + k_s f_{cook-torrance} \qquad (2)$$

Dans cette équation, $f_{lambert}$ représente la partie de la lumière réfractée, c'est-à-dire la partie correspondante à la lumière diffuse. Nous utiliserons la loi de Lambert qui est – pour c étant l'**albédo** (le pouvoir réfléchissant d'une surface) :

$$f_{lambert} = \frac{c}{\pi} \qquad (3)$$

La partie spéculaire de l'équation, $f_{cook-torrance}$, est décrite comme étant :

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \qquad (4)$$

Cette équation utilise trois fonctions importantes, qui permettent d'estimer des parties spécifiques des propriétés réflectives des surfaces, afin de respecter les principes nécessaires à un PBR. Ces fonctions sont : la fonction de **D**istribution des normales, l'équation de **F**resnel et la fonction de **G**éométrie.

La fonction de distribution des normales. Cette fonction estime de manière statistique l'aire relative de la surface des microfacettes qui sont alignés de manières exactes avec un vecteur que l'on nomme h (h pour *halfway*). La fonction que nous utiliserons et celle connue sous le nom de **Trowbridge-Reitz GGX** [TR75] :

$$NDF_{GGXTR}(n,h,\alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2-1)+1)^2} \qquad (5)$$

Ici, α est une mesure de la rugosité de la surface. Quand la rugosité est faible, c'est-à-dire lorsque la surface est lisse, une grosse concentration du nombre de microfacettes sont alignés avec le vecteur h. Le résultat de cette grosse concentration à travers un petit rayon peut être observé sur la figure 2, où la NDF affiche une petite zone très lumineuse.

L'équation de Fresnel. L'équation de Fresnel permet de décrire le ratio de lumière réfléchie par rapport à celle réfractée, qui varie en fonction de l'angle duquel on observe la

surface. L'équation de Fresnel étant très complexe, on utilisera **l'équation Fresnel-Schlick** [Sch94], qui se charge d'en faire une approximation :

$$F_{Schlick}(h,v,F_0) = F_0 + (1-F_0)(1-(h\cdot v))^5 \quad (6)$$

F_0 représente la réflectivité de base de la surface, c'est-à-dire la réflectivité d'un matériau quand on le regarde en face. Cette dernière est définit pour chaque matériau grâce à d'immenses base de données, dont les valeurs les plus communes, tirées du cours de **Naty Hoffman** [Hof10], sont présentés dans le tableau 2.1.

On définit ainsi une réflectivité de base qui est estimée pour la majorité des surfaces **diélectriques** (non-métalliques). Ensuite, en fonction de quotient métallique de la surface, on prend soit la valeur de la réflectivité de base, soit la couleur de la surface. Puisque les surfaces métalliques absorbent toute la lumière réfractée, elle n'émettent aucune réflexion diffuse, ce qui nous permet de prendre directement la couleur de la texture de la surface comme réflectivité de base.

Matériau	F_0 (Linéaire)	F_0 (sRGB)
Eau	(0.02, 0.02, 0.02)	(0.15, 0.15, 0.15)
Plastique / Verre (Low)	(0.03, 0.03, 0.03)	(0.21, 0.21, 0.21)
Plastique High	(0.05, 0.05, 0.05)	(0.24, 0.24, 0.24)
Glass (High) / Rubis	(0.08, 0.08, 0.08)	(0.31, 0.31, 0.31)
Diamant	(0.17, 0.17, 0.17)	(0.45, 0.45, 0.45)
Fer	(0.56, 0.57, 0.58)	(0.77, 0.78, 0.78)
Cuivre	(0.95, 0.64, 0.54)	(0.98, 0.82, 0.76)
Or	(1.00, 0.71, 0.29)	(1.00, 0.86, 0.57)
Aluminium	(0.91, 0.92, 0.92)	(0.96, 0.96, 0.97)
Argent	(0.95, 0.93, 0.88)	(0.98, 0.97, 0.95)

Table 1: *Valeurs de réflectivité de base les plus communes pour une surface, tirées du cours de Naty Hoffman.*

La fonction de géométrie. La fonction de géométrie estime de manière approximative l'aire de la surface où les détails microscopiques de cette dernière s'éclipsent entre eux, occultant ainsi les rayons de lumière, telle qu'on peut l'observer sur la figure 4.

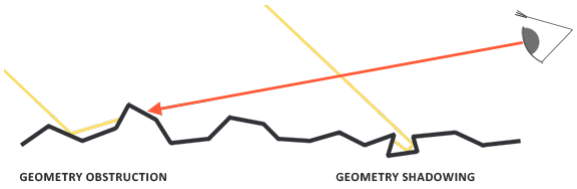

Figure 4: *Schéma de l'aire estimée par la fonction de géométrie.*

Comme pour la fonction de distribution des normales, la fonction de géométrie prend en entrée un paramètre de rugosité de la surface, puisque les surfaces les plus rugueuses sont celles qui possèdent le plus de microfacettes qui s'éclipsent. La fonction de géométrie que nous utiliseront est un mélange de GGX et Schlick-Beckmann, connue sous le nom de **Schlick-GGX** :

$$G_{SchlickGGX}(n,v,k) = \frac{n\cdot v}{(n\cdot v)(1-k)+k} \quad (7)$$

Ici, k varie en fonction de si on utilise un éclairage direct ou IBL – dont on parlera plus en détail dans la prochaine sous partie.

$$k_{direct} = \frac{(\alpha+1)^2}{8} \quad (8)$$

$$k_{IBL} = \frac{\alpha^2}{2} \quad (9)$$

Afin d'estimer correctement la géometrie, il est nécessaire de prendre en compte à la fois l'obstruction géométrique et l'ombrage géométrique. Ceci est possible en utilisant la méthode de **Smith** [Smi67], qui est la suivante :

$$G(n,v,l,k) = G_{SchlickGGX}(n,v,k)G_{SchlickGGX}(n,l,k) \quad (10)$$

Le résultat de cette fonction est démontrée par la figure 5, pour un paramètre de rugosité variant entre 0 et 1.

L'équation de réflectance Cook-Torrance.

Lorsqu'on combine l'équation de réflectance avec le BRDF de Cook-Torrance, on obtient l'équation de réflectance Cook-Torrance, qui décrit de manière complète un modèle de rendu basé sur la physique, à savoir :

$$L_o(p,\omega_o) = \int_\Omega (k_d\frac{c}{\pi} + \frac{DFG}{4(\omega_o\cdot n)(\omega_i\cdot n)})L_i(p,\omega_i)n\cdot\omega_i d\omega_i \quad (11)$$

2.2. L'éclairage basé sur l'image

Comme expliqué à travers l'équation de réflectance, la radiance entrante peut également provenir de cartes d'environnements, c'est ce qu'on appelle l'***Image-Based Lighting***, ou IBL. Contrairement à un éclairage direct, l'IBL consiste à traiter l'entièreté de l'environnement comme une grosse source de lumière. Afin de créer ces environnements, on utilise le ***cube mapping***, qui permet de projeter une image représentant l'environnement autour de la caméra via un cube à six faces.

Comme expliqué plus tôt à travers l'équation de réflectance, il est nécessaire de résoudre cette intégrale pour chaque rayon de lumière entrant dans une surface, ce qui peut vite devenir complexe. Il nous est donc nécessaire d'avoir un moyen de récupérer la radiance d'une scène pour n'importe quel vecteur de direction w_i, et le calcul de l'intégrale doit être rapide et en temps réel.

Afin de pouvoir résoudre la première contrainte, il nous suffit de traiter la *cube map* représentant l'environnement, afin de pouvoir définir chaque **texel** (de l'anglais *texture element*) comme la représentation d'une source de lumière.
De plus, afin de pouvoir déterminer les intégrales rapidement, il nous est nécessaire de pré-calculer la majorité des

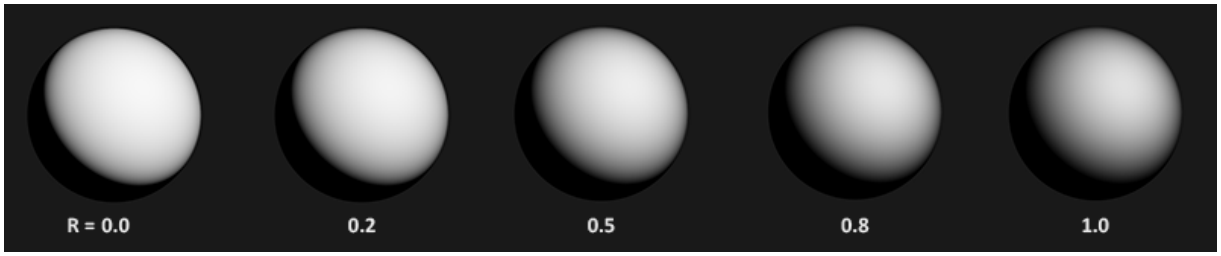

Figure 5: *Démonstration de la fonction de géométrie en fonction d'un paramètre de rugosité de la surface variant entre 0 et 1.*

calculs.
Pour ce faire, on commence par séparer l'équation de réflectance en deux parties, une partie diffuse et une partie spéculaire, afin de pouvoir les traiter séparément, puisqu'elles sont indépendantes. Ainsi, on obtient :

$$L_o(p,\omega_o) = \int_\Omega (k_d \frac{c}{\pi}) L_i(p,\omega_i) n\omega_i d\omega_i + \int_\Omega (k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}) L_i(p,\omega_i) n \cdot \omega_i d\omega_i \quad (12)$$

La lumière diffuse. Il s'agit ici de se concentrer sur la partie diffuse de l'équation de réflectance. Avant tout, on remarque que la couleur c, l'indice de réfraction k_d et π sont des constantes et il nous est donc possible de les sortir de l'intégrale, ce qui nous donne :

$$L_o(p,\omega_o) = k_d \frac{c}{\pi} \int_\Omega L_i(p,\omega_i) n \cdot \omega_i d\omega_i \quad (13)$$

Si on considère le point p comme le centre de la carte d'environnement, l'intégrale ne dépend maintenant plus que w_i. Ainsi, nous pouvons pré-calculer une nouvelle *cube map* qui sauvegarde dans chaque texel w_0 le résultat de l'intégrale par **produit de convolution** (un opérateur bilinéaire en mathématiques). Autrement dit, pour chaque direction de rayon dans la *cube map*, nous allons prendre en considération toutes les autres directions, à travers l'hémisphère Ω.
Donc, pour chaque w_0, nous allons calculer la moyenne des radiances d'un grand nombre de directions w_i, tel que décrit à travers la figure 6.

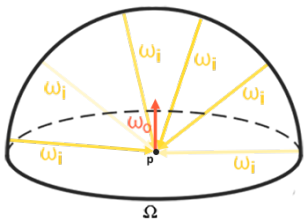

Figure 6: *Illustration du calcul des intégrales par convolution de la partie diffuse de l'équation de réflectance.*

Concrètement, cela signifie que l'on se retrouve avec une nouvelle *cube map*, que l'on nomme une **carte d'irradiance** (ou *irradiance map* en anglais). Un exemple d'une carte d'environnement et de la carte d'irradiance associée est présenté en figure 7.

Figure 7: *Exemple d'une carte d'environnement et de la carte d'irradiance associée.*

Il est important de noter que pour un moteur PBR, il est nécessaire que les cartes d'environnements soient en **HDR** (de l'anglais *High Dynamic Range*), afin de pouvoir spécifier correctement l'intensité de chaque source lumineuse.
Ainsi, dans un fichier .hdr (que l'on nomme ***radiance file format***), les six faces de la *cube map* sont sauvegardés via des flottants, ce qui nous permet d'être libéré de la contrainte de devoir spécifier toutes les couleurs entre 0.0 et 1.0, et donc d'avoir des plus grands écarts d'intensité lumineuse.
De plus, l'image est enregistrée via une **projection cylindrique équidistante** (*equirectangular map* en anglais) – c'est-à-dire via une projection sphérique aplatie. De ce fait, étant donné que certains calculs peuvent se révéler très coûteux sur une *equirectangular map*, il nous sera plus utile de la transformer en *cube map*.

La lumière spéculaire. En ce qui concerne la partie spéculaire de l'équation de réflectance, on remarque que l'indice de réflexion k_d n'est pas constant sur l'intégrale et il nous est donc impossible de le déplacer. **Epic Games** propose une solution qui, pour quelques compromis, est efficace en temps réel. Cette solution s'appelle la ***split sum approximation*** [Kar13].
Tout d'abord, revisitons la partie spéculaire de l'équation de réflectance :

$$L_o(p,\omega_o) = \int_\Omega (k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} L_i(p,\omega_i) n \cdot \omega_i d\omega_i \\ = \int_\Omega f_r(p,\omega_i,\omega_o) L_i(p,\omega_i) n \cdot \omega_i d\omega_i \quad (14)$$

Comme pour la partie diffuse, il nous est impossible de calculer efficacement ces équations d'intégrales en temps réel. On souhaite donc avoir une sorte de carte IBL spéculaire pré-calculée, afin de sauvegarder ces informations à l'avance.
L'approximation de somme partagée d'Epic Games résout ce problème en partageant la partie spéculaire de l'équation en deux parties à travers lesquelles on peut individuellement utiliser le produit de convolution avant de les combiner via le shader PBR, de la manière suivante :

$$L_o(p,\omega_o) = \int_\Omega L_i(p,\omega_i) d\omega_i * \int_\Omega f_r(p,\omega_i,\omega_o) n \cdot \omega_i d\omega_i \quad (15)$$

Après avoir appliqué le produit de convolution, la première partie est connu sous le nom de ***pre-filtered environment map***, c'est-à-dire une carte d'environnement qui présente la rugosité des surfaces grâce au flou.
On calcule ainsi plusieurs niveau de rugosité via le **MIP Mapping**; une technique d'application des textures qui fonctionne avec plusieurs niveaux de textures, chaque niveau suivant ayant une hauteur et une largeur inférieure (d'une puissance de deux) au niveau précédent. Ainsi, ce procédé créera des cartes qui seront de plus en plus floues et de plus en plus petites.
Par exemple, une *pre-filtered environment map* qui sauvegarde cinq niveaux de rugosité à travers ces cinq mipmaps ressemblera à celle présente sur la figure 8.

Figure 8: *Exemple d'une pre-filtered environment map qui sauvegarde cinq niveaux de rugosité à travers ces cinq mipmaps.*

La seconde partie de l'équation concerne la partie BRDF de l'intégrale. Si on considère que la radiance entrante est uniformément blanche dans toutes les directions, c'est-à-dire que $L(p,x) = 1.0$, on peut pré-calculer la partie BRDF en fonction d'une rugosité et de l'angle entre la normale n et la direction de la lumière w_i.
Epic Games sauvegarde le résultat du pré-calcul de la partie BRDF pour chaque combinaison possible entre n et w_i à travers une texture 2D, connue sous le nom de ***BRDF integration map***. Cette texture, présentée via la figure 9, contient une valeur de *scale* en rouge et une valeur de *bias* en vert, représentant la réponse sur la surface par rapport à l'équation de Fresnel.

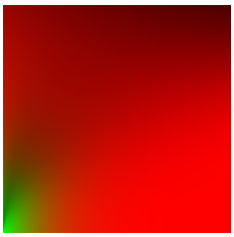

Figure 9: *La texture 2D connue sous le nom de BRDF integration map.*

3. Exposé de la méthode

3.1. Structure

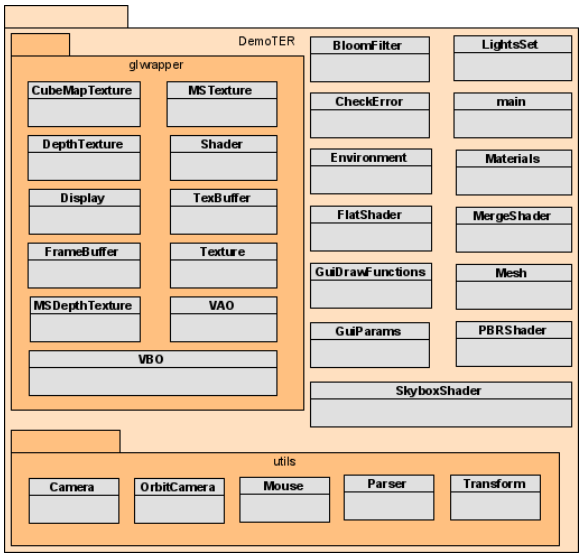

Figure 10: *Diagramme de classe représentant la structure de notre projet.*

Comme vous pouvez le voir sur la figure 10, notre code est structuré de manière à séparer les classes en fonction de leur utilité.

Dans un premier temps, on retrouve l'ensemble de classe **utils**. Il regroupe les fonctions que l'on catégorise comme utilitaire, tel que la gestion de la caméra, de la souris, ou encore l'analyseur de fichiers *.obj*. Bien qu'indispensables pour le bon fonctionnement de notre projet, dans un soucis d'espace, nous n'entreront pas en détail sur le contenu de ces classes, puisque leur comportement correspond à celui que l'on attendrait d'elles; nous préférons nous concentrer sur les parties du code concernant le PBR et le fonctionnement global.

Ensuite, nous trouvons l'ensemble de classe **glwrapper**.

Cet ensemble contient les classes qui englobent les fonctionnalités d'Open GL, dans le but de nous permettre une utilisation plus simple et plus intuitive de *glad*, *glm* et *glfw* à travers le reste du code. Ces classes correspondent aux VBOs, VAOs, à l'affichage ou encore à la gestion des textures. Encore une fois, toujours pour les mêmes raisons, nous préférons ne pas détailler le code, les méthodes étant toujours très intuitives et documentés.

Toutes les autres classes correspondent au cœur du projet. Elles représentent tous les éléments nécessaire pour l'affichage graphique, que ce soit la gestion des différents shaders, des *skybox*, des lumières, des maillages, des matériaux ou encore de l'interface graphique. Ce sont ces classes qui apparaissent à travers la pipeline – c'est-à-dire à travers les différentes étapes de la création et de l'affichage des éléments graphique – et ce sont celles sur lesquelles notre explication va se concentrer.

3.2. Pipeline

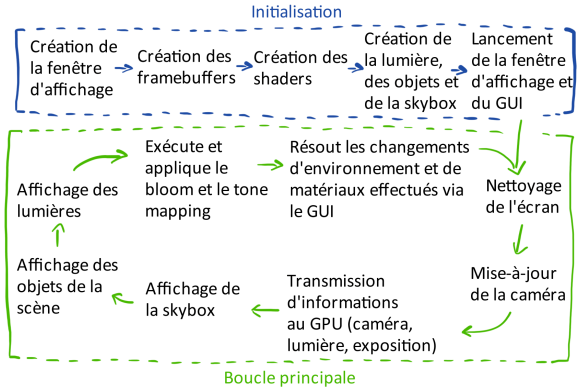

Figure 11: *Schéma de la pipeline d'exécution de notre projet.*

Comme vous pouvez le voir via la figure 11, la pipeline d'exécution du projet est plutôt simple. Il s'agit tout d'abord de créer tous les éléments dont nous allons avoir besoin pour l'affichage d'une image, puis de rentrer dans une boucle qui est exécutée à l'affichage de chaque image.
Parcourons maintenant chaque étape de la pipeline, en nous concentrant notamment sur les parties les plus importantes.

Création de la fenêtre d'affichage. La création de la fenêtre se fait de manière simple, en utilisant la classe **Display** (qui se trouve dans l'ensemble *glwrapper*). Cette classe adopte le patron de conception singleton et utilise une fenêtre de type **GLFWwindow**.

Création des framebuffers. Les framebuffers sont des objets auxquels on peut assigner tout type de textures (couleur, profondeur, masque). Leur but est de nous permettre de dessiner directement dans ces textures à la place de l'écran.
Dans notre programme, **'msFrameBuffer'** est le framebuffer à travers lequel sont rendus tous les objets de la scène, ainsi que la *skybox*. Celui-ci est composé de deux textures de couleur, une avec toutes les couleurs présentes, tandis que la seconde contient seulement les zones très lumineuses (ainsi qu'une texture de profondeur, pour le z-order).

La particularité des textures de 'msFrameBuffer' relève du fait que ces dernières soient multisamplées, c'est-à-dire que la carte graphique exécutera un algorithme **d'anti-crénelage (MSAA)** [Khr] au moment de dessiner à l'intérieur de ces textures.
Pour avoir une image finale, les échantillons de textures doivent être fusionnées ; c'est là qu'intervient les framebuffers **'resolvedFBTex0'** et **'resolvedFBTex1'**. Les deux textures de couleur précédemment mentionnés sont ainsi transvasées dans 'resolvedFBTex0' et 'resolvedFBTex1' (qui eux ne sont pas multisamplés), la carte graphique fusionne alors automatiquement les échantillons pour obtenir des textures classiques facilement exploitables.

De plus, la texture de 'resolvedFBTex1' – qui ne contient que les pixels de forte intensité – est donné en entrée du filtre de *bloom* (dont nous détaillerons le fonctionnement par la suite), qui rend en sortie une version floue de l'image.
De son côté, la texture de 'resolvedFBTex0' est simplement additionnée à l'image précédente.
Enfin, le résultat est corrigé par la formule de correction gamma avant d'être affiché à l'écran. Ce processus est présenté à travers un schéma en figure 12.

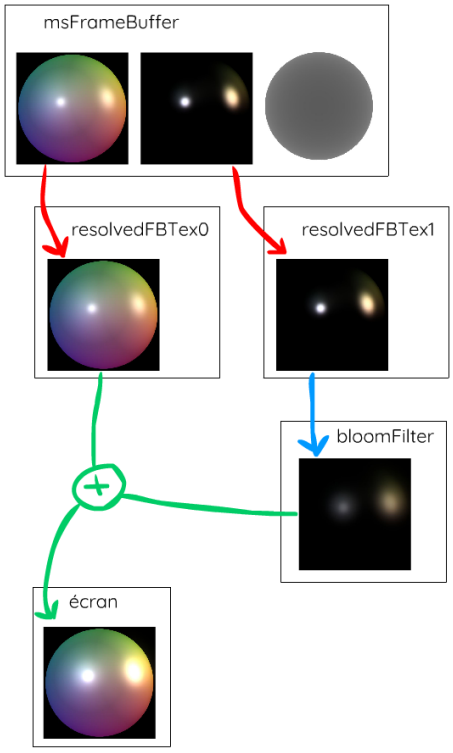

Figure 12: *Schéma explicatif du processus d'application de l'effet d'éclat et de traitement de l'image, en utilisant des framebuffers.*

Création des shaders. Les shaders sont les programmes informatiques qui permettent de paramétrer une partie du processus de rendu afin qu'il soit réalisé par une carte graphique. Ainsi, dans le cas de notre projet, les shaders sont

d'une importance cruciale, puisque ce sont eux qui, par exemple, contiendront tous les calculs liés au PBR.
Un **program shader** est constitué de deux fichiers. Tout d'abord, il est constitué d'un **vertex shader** qui, pour faire simple, est un programme se substituant à la phase de transformation des sommets et au calcul d'éclairement/coordonnées de texture aux sommets. Ensuite, il est accompagné d'un **fragment shader** qui, tout aussi simplement, correspond à un programme qui se substitue à la phase d'attribution de la couleur et des coordonnées de texture aux fragments (lors de la rasterization).

À travers notre programme, nous utilisons douze shaders qui se trouvent dans 'resources/shaders'. Ces derniers sont divisés en trois catégories : quatre pour l'effet d'éclat, quatre pour l'IBL et quatre pour les reste des objets. Nous reviendrons sur ces shaders dans la partie suivante, après l'explication complète de la pipeline.

Création de la lumière, des objets et de la *skybox*. En bref, les lumières font parties d'un objet de type **LightsSet**, elles possèdent une position, une couleur et une intensité.
Les objets, quant à eux, sont représentés via une structure, chacun possède un nom (qui permet notamment de l'identifier dans l'interface graphique), un booléen (qui permet de l'afficher ou non à l'écran), un maillage (classe **Mesh**) et un matériau (classe **Materials**). Il est important de noter qu'un matériau est un ensemble de cinq textures : l'abledo, la carte des normales, la rugosité, l'aspect métallique et l'occlusion ambiante [Max20]. Ces textures peuvent donc soit être de véritables textures artistiques (classe **Texture**), soit des flottants.
Enfin, la *skybox* correspond à la carte d'environnement, soit un objet de type **Environment**.

Lancement de la fenêtre d'affichage et du GUI. La fenêtre d'affichage et donc le lancement de la boucle principale d'exécution se fait par le simple appel d'une méthode.
De plus, le GUI, c'est-à-dire l'interface graphique utilisateur, est elle gérée via **ImGui** [Oco] à travers une méthode qui définit tous les éléments présent à travers l'interface. Au sein de cette dernière, on retrouve la possibilité de :

- changer le paramètre d'exposition et de gamma de la scène,
- modifier les lumières (intensité, couleur, allumé/éteinte),
- afficher ou non uniquement la partie *bloom* du rendu graphique,
- afficher ou non d'une carte d'environnement,
- choisir parmi les preset de carte d'environnement disponibles (récupérés depuis **sIBL Archive** [sIB]),
- afficher ou cacher les objets présents sur la scène,
- sélectionner un objet et lui appliquer un préset de matériau ou un matériau personnalisé (les textures artistiques ayant étés récupérées depuis **Poliigon** [Pol]),
- choisir entre la visualisation de la carte d'environnement finale, la *pre-filtered environment map* ou *l'irradiance map*.

Un certains nombre de paramètres, correspondant soit au paramètres globaux de la scène − tel que l'exposition − ou soit à des drapeaux (afin de savoir quels éléments on étés modifiés), sont sauvegardés via un objet de type **GuiParams**.

Nettoyage de l'écran. Le nettoyage de l'écran, afin de pouvoir dessiner une nouvelle image, correspond à la première action effectuée à chaque répétition de la boucle principale.

Mise-à-jour de la caméra. En fonction du déplacement de la souris, les nouvelles coordonnées de la caméra sont mise-à-jour − puisque l'utilisateur peut se déplacer à travers l'espace et zoomer grâce à la caméra orbitale de type **OrbitCamera**.

Transmission d'informations au GPU (caméra, lumière, exposition). C'est ici que les informations liés à la nouvelle position de la caméra, aux propriétés de la scène et des lumières − qui peuvent avoir étés modifiés par l'utilisateur − sont envoyés à la carte graphique, via le shader PBR que nous détaillerons à travers la section suivante.

Affichage de la *skybox*. On affiche ensuite la carte d'environnement grâce à son propre shader, avec la *pre-filtered environment map*, *l'irradiance map* et la *BRDF integration map* associés afin d'obtenir un résultat IBL.
Dans le cas où ne souhaite pas voir d'environnement, on associe ces cartes à celle d'une carte d'environnement totalement noire créée préalablement.

Affichage des objets de la scène. Il s'agit ici d'afficher tous les objets présents au sein la scène, grâce au shader PBR (afin d'obtenir un rendu physiquement réaliste sur chacun des objets affichés à l'écran).
Dans le cadre de notre projet, afin de mettre en valeurs les différentes propriétés de notre moteur de rendu basé sur la physique, nous affichons 49 petites sphères, dont la rugosité et l'aspect métallique varie, une sphère avec des propriétés UV (pour pouvoir lui appliquer des matériaux plus complexes) et enfin trois objets avec plus complexes − à savoir un Cerberus [Max14], une épée [3D17] et une statuette [Han18].

Affichage des lumières. On affiche les lumières grâce à un shader particulier, qui nous permet d'afficher ces dernières sans avoir à passer par toute la complexité lié au shader PBR. En effet, le but ici est de pouvoir visualiser la position des lumières dans la scène, bien que ce ne soit pas des objets concrets, d'où la simplicité du shader utilisé.

Exécute et applique le *bloom* et le *tone mapping*. Il existe plusieurs manière de rendre une image floue. Celle que nous avons choisi d'implémenter est constituée de trois étapes et provient de la documentation de l'**Unreal Engine** [Epi]. Chacune d'entre elle consiste à appliquer une matrice de convolution à l'image. Cette matrice se caractérise par sa taille en pixels et ses poids, qui sont répartis par une fonction gaussienne ; plus les poids sont loin du centre de la matrice, plus leurs valeurs sont faibles.

Chaque étape du flou est effectuée sur une résolution d'image et une taille de matrice différente. La texture correspondante à 1/16 de la taille de la texture originelle est rendue floue avec une matrice de taille 11, puis celle-ci est additionnée à une texture de taille supérieure (1/8) et floutée avec une matrice de taille 9, et enfin, le résultat est additionné à une texture de taille encore supérieur (1/4), pour être à nouveau floutée avec une matrice de taille 5, ce qui nous donne le résultat final.

Pour avoir de meilleures performances, chaque étape de flou est divisée en une passe horizontale et une passe verticale (au lieu d'applique la matrice de convolution entière), cela réduit le nombre de lectures des textures. Ce processus est présenté à travers un schéma en figure 13.

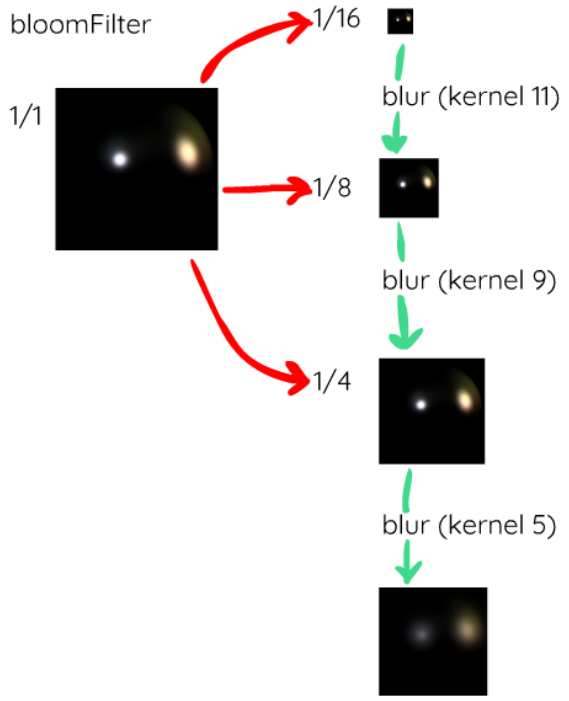

Figure 13: *Schéma explicatif du fonctionnement de l'effet d'éclat.*

Résout les changements d'environnements et de matériaux effectués via le GUI. Enfin, grâce aux différents booléens présents dans GuiParams, il nous est possible de charger un nouvel environnement ou un nouveau matériau en fonction des drapeaux qui ont étés levés. Ainsi, on évite de réaliser des instructions potentiellement coûteuses si elles ne sont pas désirés.

3.3. Shaders

Comme expliqué plus tôt, les shaders sont les éléments les plus intéressants de notre projet puisque ce sont eux qui sont responsables du rendu graphique. Par exemple, lorsqu'on souhaite afficher un objet, c'est-à-dire un maillage accompagné de son matériau, c'est le shader – en l'occurrence notre **pbrShader** – qui est responsable du résultat que l'on souhaite obtenir, à savoir un rendu graphique basé sur la physique.

Le shader PBR. Ce shader est composé de 'pbr.vert' et de 'pbr.frag'. C'est donc dans le *fragment shader* associé que l'on retrouvera le code correspondant à toute les formules détailles dans la partie 2.1.
Il est nécessaire pour ce shader de connaître tous les éléments qui peuvent influer sur la rendu graphique. Ainsi, on retrouve en valeur suivantes définie comme **uniform**, c'est-à-dire les valeurs connues pour tous les threads et accessible en lecture seule :

- les paramètres d'exposition et de gamma de la scène,
- les lumières,
- la position de la caméra,
- les cartes pré-calculés pour l'IBL liés à la carte d'environnement choisie.

Dans un premier temps, comme expliqué à travers la partie théorique, c'est dans ce shader que se situe le code lié à l'équation de réfléctance, que vous trouverez en listing 1.

```
1    // reflectance equation
2    vec3 Lo = vec3(0.0);
3    for(int i = 0; i < 4; ++i) {
4        vec3 L = normalize(lightPositions[i]
      - worldPos);
5        vec3 H = normalize(V + L);
6
7        float HdotV = max(dot(H, V), 0.0);
8        float NdotV = max(dot(N, V), 0.0);
9        float NdotL = max(dot(N, L), 0.0);
10
11       // calculate per-light radiance
12       float distance    = length(
     lightPositions[i] - worldPos);
13       float attenuation = 1.0 / (distance
     * distance);
14       vec3 radiance     = lightColors[i] *
      attenuation;
15
16       // cook-torrance brdf
17       float NDF = DistributionGGX(N, H,
     roughness);
18       float G   = GeometrySmith(NdotV,
     NdotL, roughness);
19       vec3 F    = fresnelSchlick(HdotV, F0
     );
20
21       vec3 kS = F;
22       vec3 kD = vec3(1.0) - kS;
23       kD *= 1.0 - metallic;
24
25       vec3 numerator    = NDF * G * F;
26       float denominator = 4.0 * NdotV *
     NdotL;
27       vec3 specular     = numerator / max(
     denominator, 0.001);
28
29       // add to outgoing radiance Lo
30       Lo += (kD * albedo / PI + specular)
     * radiance * NdotL;
31   }
```

Listing 1: *Extrait des lignes 88 à 118 du fichier pbr.frag concernant l'équation de réflectance.*

Ainsi, on retrouve la méthode correspondante à la fonction Trowbridge-Reitz GGX en listing 2 (équation 5), celle de Schlick-GGX en listing 3 (équation 7), la méthode de Smith en listing 4 (équation 10) et enfin l'équation de Fresnel-Schlick en listing 5 (équation 6).

```
1  float DistributionGGX(vec3 N, vec3 H, float
       roughness) {
2      float a      = roughness * roughness;
```

```
    float a2     = a * a;
    float NdotH  = max(dot(N, H), 0.0);
    float NdotH2 = NdotH * NdotH;

    float num   = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;

    return num / denom;
}
```

Listing 2: *Extrait des lignes 34 à 45 du fichier pbr.frag concernant la fonction de Trowbridge-Reitz GGX.*

```
float GeometrySchlickGGX(float NdotV, float roughness) {
    float r = (roughness + 1.0);
    float k = (r * r) / 8.0;

    float num   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return num / denom;
}
```

Listing 3: *Extrait des lignes 47 à 55 du fichier pbr.frag concernant la fonction de Schlick-GGX.*

```
float GeometrySmith(float NdotV, float NdotL, float roughness) {
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}
```

Listing 4: *Extrait des lignes 57 à 62 du fichier pbr.frag concernant la méthode de Smith.*

```
vec3 fresnelSchlick(float cosTheta, vec3 F0) {
    return F0 + (1.0 - F0) * pow(max(1.0 - cosTheta, 0.0), 5.0);
}

vec3 fresnelSchlickRoughness(float cosTheta, vec3 F0, float roughness) {
    return F0 + (max(vec3(1.0 - roughness), F0) - F0) * pow(max(1.0 - cosTheta, 0.0), 5.0);
}
```

Listing 5: *Extrait des lignes 64 à 70 du fichier pbr.frag concernant l'équation de Fresnel-Schlick.*

Dans un second temps, c'est toujours dans ce shader, cette fois-ci via la listing 6, que ce situe les calculs liés à l'IBL, tel que décrit dans la partie 2.2.

```
    // IBL
    // ambient lighting (we now use IBL as the ambient term)
    vec3 F = fresnelSchlickRoughness(max(dot(N, V), 0.0), F0, roughness);

    vec3 kS = F;
    vec3 kD = 1.0 - kS;
    kD *= 1.0 - metallic;

    vec3 irradiance = texture(irradianceMap, N).rgb;
    vec3 diffuse    = irradiance * albedo;

    // sample both the pre-filter map and the BRDF lut and combine them together as per the Split-Sum approximation to get the IBL specular part.
    const float MAX_REFLECTION_LOD = 4.0;
    vec3 R = reflect(-V, N);
    vec3 prefilteredColor = textureLod(prefilterMap, R,  roughness * MAX_REFLECTION_LOD).rgb;
    vec2 brdf  = texture(brdfLUT, vec2(max(dot(N, V), 0.0), roughness)).rg;
    vec3 specular = prefilteredColor * (F * brdf.x + brdf.y);

    vec3 ambient = (kD * diffuse + specular) * ao;

    vec3 color = ambient + Lo;
```

Listing 6: *Extrait des lignes 123 à 143 du fichier pbr.frag concernant l'éclairage basé sur l'environnement.*

Le shaders environnement. Tous les shaders qui sont dans le sous-dossier 'env' correspondent aux shaders IBL, qui pré-calculent les différentes cartes d'environnements — tels que décrits dans la section à propos de l'éclairage basé sur l'image — afin que le shader PBR puisse les utiliser à travers le rendu graphique des objets de la scène. On retrouve donc quatre shaders dans cet ensemble.

Le premier shader, **'rectToCube'**, permet de transformer une image HDR en projection cylindrique équidistante vers une image sous la forme d'une *cube map*.

Le deuxième shader, **'brdf'**, pré-calcul une seule fois au lancement du programme la *BRDF integration map*, correspondant à la figure 9.

Le troisième shader, **'prefilter'**, pré-calcul la *pre-filtered environment map*, ce à chaque changement d'environnement. Le fichier 'prefilter.frag' est présenté en listing 7.

```
// ----------------
vec2 Hammersley(uint i, uint N)
{
  return vec2(float(i)/float(N), RadicalInverse_VdC(i));
}
// ----------------
vec3 ImportanceSampleGGX(vec2 Xi, vec3 N, float roughness)
{
  float a = roughness*roughness;

  float phi = 2.0 * PI * Xi.x;
```

```
  float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (a*a - 1.0) * Xi.y));
  float sinTheta = sqrt(1.0 - cosTheta*cosTheta);

  // from spherical coordinates to cartesian coordinates - halfway vector
  vec3 H;
  H.x = cos(phi) * sinTheta;
  H.y = sin(phi) * sinTheta;
  H.z = cosTheta;

  // from tangent-space H vector to world-space sample vector
  vec3 up          = abs(N.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
  vec3 tangent   = normalize(cross(up, N));
  vec3 bitangent = cross(N, tangent);

  vec3 sampleVec = tangent * H.x + bitangent * H.y + N * H.z;
  return normalize(sampleVec);
}
// ----------------
void main()
{
    vec3 N = normalize(WorldPos);

    // make the simplyfying assumption that V equals R equals the normal
    vec3 R = N;
    vec3 V = R;

    const uint SAMPLE_COUNT = 1024u;
    vec3 prefilteredColor = vec3(0.0);
    float totalWeight = 0.0;

    for(uint i = 0u; i < SAMPLE_COUNT; ++i)
    {
        // generates a sample vector that's biased towards the preferred alignment direction (importance sampling).
        vec2 Xi = Hammersley(i, SAMPLE_COUNT);
        vec3 H = ImportanceSampleGGX(Xi, N, roughness);
        vec3 L  = normalize(2.0 * dot(V, H) * H - V);

        float NdotL = max(dot(N, L), 0.0);
        if(NdotL > 0.0)
        {
            // sample from the environment's mip level based on roughness/pdf
            float D   = DistributionGGX(N, H, roughness);
            float NdotH = max(dot(N, H), 0.0);
            float HdotV = max(dot(H, V), 0.0);
            float pdf = D * NdotH / (4.0 * HdotV) + 0.0001;

            float saTexel  = 4.0 * PI / (6.0 * resolution * resolution);
            float saSample = 1.0 / (float(SAMPLE_COUNT) * pdf + 0.0001);

            float mipLevel = roughness == 0.0 ? 0.0 : 0.5 * log2(saSample / saTexel);

            prefilteredColor += textureLod(environmentMap, L, mipLevel).rgb * NdotL;
            totalWeight      += NdotL;
        }
    }

    prefilteredColor = prefilteredColor / totalWeight;

    FragColor = vec4(prefilteredColor, 1.0);
}
```

Listing 7: *Extrait des lignes 36 à 106 du fichier prefilter.frag concernant la création de la pre-filtered environment map associée à une carte d'environnement.*

Le quatrième shader, **'irradiance'**, pré-calcul *l'irradiance map*, ce à chaque changement d'environnement. Le fichier 'irradiance.frag' est présenté en listing 8.

```
void main()
{
    vec3 N = normalize(WorldPos);

    vec3 irradiance = vec3(0.0);

    // tangent space calculation from origin point
    vec3 up    = vec3(0.0, 1.0, 0.0);
    vec3 right = normalize(cross(up, N));
    up         = normalize(cross(N, right));

    float sampleDelta = 0.025;
    float nrSamples = 0.0;
    for(float phi = 0.0; phi < 2.0 * PI; phi += sampleDelta)
    {
        for(float theta = 0.0; theta < 0.5 * PI; theta += sampleDelta)
        {
            // spherical to cartesian (in tangent space)
            vec3 tangentSample = vec3(sin(theta) * cos(phi),  sin(theta) * sin(phi), cos(theta));
            // tangent space to world
            vec3 sampleVec = tangentSample.x * right + tangentSample.y * up + tangentSample.z * N;

            irradiance += texture(environmentMap, sampleVec).rgb * cos(theta) * sin(theta);
            nrSamples++;
        }
    }
    irradiance = PI * irradiance * (1.0 / float(nrSamples));

    FragColor = vec4(irradiance, 1.0);
```

```
30 }
```

Listing 8: *Extrait des lignes 9 à 43 du fichier irradiance.frag concernant la création de la l'irradiance map associée à une carte d'environnement.*

Les autres shaders. Bien que tout aussi utiles, les autres shaders sont bien moins complexes et sont ainsi moins intéressants à étudier. Ils nous permettent notamment d'afficher les lumières dans la scène sans les modéliser de manière complexe via le shader PBR (c'est le rôle du shader **'flat'**) ou encore d'afficher la *skybox* en tant qu'environnement de la scène (c'est le rôle du shader **'skybox'**).

4. Résultats et validation

4.1. Rendu final

Une capture d'écran du rendu final pour un épée est présenté en figure 14.
L'épée elle-même est composé d'un maillage et des cinq textures que nous avons introduites plus tôt dans le rapport : l'albedo, la carte des normales, l'aspect métallique, la rugosité et l'occlusion ambiante.
Cet épée est ensuite éclairée directement par deux lumières : une verte et une rouge. On voit que ces éclairages sont placés du côté droit de la scène (elles sont visibles sur la capture d'écran) et que les deux luminosités qu'elles génèrent sont également capables de se mélanger, ce qui permet d'afficher une couleur jaune à l'intersection de leurs deux zones d'éclairages respectives.
De plus, l'épée est illuminé par l'environnement, qui est ici représenté par un lac sombre et bleuté. Ainsi, les trois cartes pré-calculées par notre programme − c'est-à-dire la carte d'intégration BRDF, la carte d'irradiance et la carte d'environnement pré-filtré − permettent un éclairage dynamique par l'environnement ; il suffit de comparer le résultat final à la miniature "Direct lighting" pour voir la différence.
Enfin, l'effet d'éclat est appliqué sur l'épée au niveau des zones avec une grande intensité lumineuse. L'impact de cette effet seul est visible sur la miniature "Bloom effect".

4.2. Perfomances

Nous avons mesuré les performances du moteur de rendu en temps réel depuis trois machines au caractéristiques différentes. Sur chaque machine, nous avons mesuré le nombre d'image par seconde ainsi que l'usage mémoire du processus sous deux conditions, lorsqu'aucun objet était affiché (**cas 1**) et lorsque les cinquante-trois objets décrits dans la partie 3.2 (paragraphe "Affichage des objets de la scène") était modélisés (**cas 2**).
La **machine 1** est un ordinateur portable sous Windows 10, avec un processeur AMD Ryzen 7 4800H 2,90 GHz, une carte graphique NVIDIA GeForce RTX 2060 et 16 Go de RAM.
La **machine 2** est un ordinateur fixe sous Windows 10, avec un processeur Intel Core i5 3,40 GHz, une carte graphique NVIDIA GeForce GTX 1060 et 16 Go de RAM.
La **machine 3** est un ordinateur fixe sous Windows 7, avec un processeur Intel Core i7-2600 3,40 GHz, une carte graphique NVIDIA GeForce GTX 550 Ti et 8 Go de RAM.

Machine, cas	Images par secondes	Usage mémoire
Machine 1, cas 1	450 fps	105 MB
Machine 2, cas 1	1000 fps	64 MB
Machine 3, cas 1	240 fps	127 MB
Machine 1, cas 2	395 fps	105 MB
Machine 2, cas 2	600 fps	64 MB
Machine 3, cas 2	105 fps	127 MB

Table 2: *Nombre d'images par secondes et usage mémoire mesuré à travers trois machine et selon deux cas d'étude, tels que présentés dans la partie 4.2.*

On constate une baisse du nombre d'images par secondes en fonction du nombre d'objets rendus, ce qui n'est pas anormal étant donné que la quantité de travail attribué à la carte graphique augmente en fonction du nombre d'objets affichés dans la scène. En revanche, l'usage mémoire est constant à travers le temps.

4.3. Difficultés rencontrés

L'avantage d'avoir suivi les cours et tutoriels proposés par Learn OpenGL, c'est la clarté de l'introduction qui est faite au moteur de rendu PBR ainsi qu'aux diverses notions associés. Ainsi, en effectuant un certain nombre de recherches en parallèle, les concepts et idées ne nous posait pas trop de difficulté.
Malgré tout, nous avons rencontré quelques obstacles, principalement lorsqu'il s'agissait de faire fonctionner les *normal maps* des objets ou lors de la création des cartes précalculées pour l'IBL (avec une plus grande difficulté au niveau du rendu dans les MIPS maps).
De plus, au niveau du *bloom*, nous avons eu du mal à obtenir dès la première itération un effet qui soit lisse. Au début, lors de nos premiers essais, nous pouvions observer un effet "d'escalier" au niveau de la zone d'éclat.

5. Conclusion

En conclusion, nous pouvons dire que nous sommes satisfait du travail que nous avons accompli. Avec le peu de temps que nous avions à notre disposition, nous avons réussi à créer de toute pièce − avec l'aide de différents tutoriels, divers papiers de recherches et l'accompagnement de notre encadrant de TER bien sûr − un moteur de rendu basé sur la physique qui fournit un résultat plutôt réaliste tout en ayant de bonnes performances, le tout sous une structure de code assez claire. Néanmoins, nous sommes conscient que notre moteur graphique est loin d'être parfait et qu'il reste une quantité considérable de travail si nous souhaitions arriver au niveau de qualité d'un réel moteur de rendu PBR, utilisable dans les jeux vidéo par exemple.
En effet, notre moteur fonctionne très bien pour les matières que nous présentons, tel que le métal ou le plastique, mais ne permettrait pas d'afficher des visuels bluffant pour d'autres éléments, tel que la peau par exemple (qui est soumit à un principe de réfraction plus complexe) ou encore les surfaces transparentes.

Figure 14: *Capture d'écran du résultat final de notre moteur de rendu PBR, avec un aperçu des différents éléments qui permettent un tel rendu graphique.*

Par conséquent, cela nous a permis d'apprendre énormément de choses à propos de la création d'un moteur de rendu graphique. Que ce soit la compréhension et la considération de divers concepts physiques, les différentes manières de résoudre un problème, ou encore l'aspect technique, nous sommes maintenant bien plus informés sur toutes les subtilités qui composent l'informatique graphique.

Enfin, en ce qui concerne les perspectives futures, tout ce que nous souhaiterions faire, c'est améliorer la qualité et les possibilités offertes par notre moteur de rendu. Nous aurions aimer ajouter par exemple la projection d'ombres (***shadow mapping***) ou d'avantages d'effet de post-traitement – tel que la diffusion des ondes lumineuses (***light scattering***).
Nous aurions également pu passer en ***deferred shading***, technique dans laquelle le calcul de l'algorithme d'ombrage est divisé en tâches plus réduites qui écrivent dans des tampons intermédiaires dans le but d'être combinées a posteriori, plutôt que d'écrire immédiatement le résultat du shader dans la mémoire vidéo, comme nous le faisons actuellement.

Références

[3D17] 3D K. : Turbosquid - sting sword. `https://www.turbosquid.com/fr/3d-models/sting-sword-fbx-free/1125944`, 2017. Last accessed : 2021-05-22.

[CT81] COOK R. L., TORRANCE K. E. : A reflectance model for computer graphics. *Computer Graphics. Vol. 15*, Num. 3 (1981).

[dV] DE VRIES J. : Learn opengl : Learn modern opengl graphics programming in a step-by-step fashion. `https://learnopengl.com/`. Last accessed : 2021-05-22.

[dV20] DE VRIES J. : *Learn OpenGL : Learn modern OpenGL graphics programming in a step-by-step fashion*. Kendall Welling, 2020.

[Epi] EPIC G. : Bloom. `https://docs.unrealengine.com/en-US/RenderingAndGraphics/PostProcessEffects/Bloom/index.html`. Last accessed : 2021-05-22.

[Han18] HANE3D : Turbosquid - thai sandstone female model. `https://www.turbosquid.com/fr/3d-models/thai-sandstone-female-model-1275160`, 2018. Last accessed : 2021-05-22.

[Hof10] HOFFMAN N. : Physically-based shading models infilm and game production. *SIGGRAPH* (2010).

[Kar13] KARIS B. : Real shading in unreal engine 4. *SIGGRAPH* (2013).

[Khr] KHRONOS G. : Multisampling. `https://www.`

khronos.org/opengl/wiki/Multisampling. Last accessed : 2021-05-22.

[Max14] MAXIMOV A. : Physically based texturing for artists. http://artisaverb.info/PBT.html, 2014. Last accessed : 2021-05-22.

[Max20] MAXWELL W. : What is pbr and what a 3d artist should know. https://cgobsession.com/what-is-pbr-and-what-a-3d-artist-should-know/, 2020. Last accessed : 2021-05-22.

[Oco] OCORNUT : Imgui. https://github.com/ocornut/imgui. Last accessed : 2021-05-22.

[Pol] Poliigon. https://www.poliigon.com/. Last accessed : 2021-05-22.

[Sch94] SCHLICK C. : An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum. Vol. 13*, Num. 3 (1994).

[sIB] sibl archive. http://www.hdrlabs.com/sibl/archive.html. Last accessed : 2021-05-22.

[Smi67] SMITH B. : Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation. Vol. 15*, Num. 5 (1967).

[TR75] TROWBRIDGE T. S., REITZ K. P. : Average irregularity representation of a rough ray reflection. *Journal of the Optical Society of America. Vol. 65*, Num. 5 (1975).