UNIVERSITÉ DE MONTPELLIER

Rapport de Projet

RECA

Réalisation d'un moteur de rendu 3D temps réel réaliste en WebGL

Réalisé par
Ange CLEMENT
Erwan REINDERS

Sous la direction de
Noura FARAJ

Dépot Github
https://github.com/erwan-reinders/TER_M1_moteur3D

Année universitaire 2021-2022

Résumé

Le but de ce projet est de présenter et de mettre en place une série de techniques de représentation d'éléments en trois dimensions d'une scène sous forme d'images numériques afin d'obtenir un rendu réaliste en temps réel.
Cette contrainte de temps réel va nous amener à devoir approximer les phénomènes physiques du monde qui nous entourent, mais c'est là tout l'intérêt d'un tel projet. En effet, ces techniques et leur mise en application permettent la création d'interfaces en trois dimensions interactives. Nous réaliserons cette solution en JavaScript principalement, pour une implémentation graphique en WebGL. Nous détaillerons les différentes techniques que nous avons pu rencontrer pour y parvenir et les implémentations que nous avons pu en faire.

Mots clés : rendu 3D - réaliste - temps réel - WebGL

The aim of this project is to implement and show some 3D rendering techniques from a scene as digital images in order to obtain a real time and realistic rendering.
The real time constraint implies that we will have to approximate real physical phenomenons, but this is the interesting part. Indeed, these techniques and their applications allow 3D interactive interfaces. We will realize the project majorly in JavaScript, with a graphical implementation with WebGL. We will give detail about the different techniques that we encountered to reach the goal and what we have implemented.

Key words : 3D rendering - realistic - real time- WebGL

Remerciements

Nous tenons à remercier Noura Faraj pour avoir accepté de nous encadrer dans un tel projet, et pour son aide apportée tout au long de celui-ci.
Nous tenons également à remercier Sébastien Beugnon pour les présentations qu'il nous a données sur le rendu réaliste en temps réel, et qui a motivé la réalisation de ce projet.

Nous tenons aussi à adresser un grand merci à Mountaz Hascoët pour ses enseignements de bonnes pratiques de programmation en WebGL.

Table des matières

Table des figures

Glossaire

API (Application Programming Interface) : Interface logicielle permettant à deux logiciels ou services de pouvoir se connecter entre eux.

WebGl (Web Graphic library) : API JavaScript, maintenue par le groupe Khronos, permettant la production de rendu 2D (historiquement) et 3D

3D : Trois dimensions, souvent représentées par les trois axes principaux X,Y et Z.

Pipeline graphique : Suite de rendus effectués les un à la suite des autres.

Fps (frame per second) : taux d'images affichées en une seule seconde à l'écran (aussi appelé taux de rafraîchissement).

Langages du WEB : Langages de programmation historiquement dédiés à de la programmation orientée sur le développement de solutions informatiques WEB, comme HTML ou JS.

DOM (Document Object Model) : Interface de programmation pour des documents HTML ou XML/SVG, fournissant la manière de structurer et manipuler et styliser du contenu, dans notre cas des pages internets.

Texture : Une image formée d'éléments appelées texels. Ces texels possèdent une couleur rouge, verte et bleue, avec des valeurs comprises entre 0 et 1.

Shaders : Un code qui doit s'exécuter sur la carte graphique.

Program : Un assemblage de deux shaders : l'un pour les vertex des maillages, l'autre pour les pixels de la texture de sortie.

Uniform : Une valeur passée dans un program qui, pour un rendu, ne change pas.

Varying : Une valeur passée depuis le vertex shader jusqu'au fragment shader.

Stencil buffer : Type particulier de buffer de données, permettant de faire passer des informations binaires du CPU vers le GPU pour un traitement graphique ensuite (peut s'employer dans des opérations de masquage).

Chapitre 1

Introduction

La 3D est au cœur de nombreuses applications de nos jours, et tend à se démocratiser de plus en plus dans des secteurs comme la publicité, le divertissement ou encore l'architecture. Dans ce flot de nouveaux domaines possibles d'application, deux courants majeurs de rendu 3D émergent : le rendu en temps réel (exploitant le pipeline graphique GPU) et les rendus en temps plus ou moins maîtrisés (comme le raytracing par exemple), demandant le déploiement de ressources de calculs jusqu'à parfois plusieurs jours pour le rendu d'une simple image numérique à partir d'une scène en 3D.

Pour certains domaines, il en va de soi que le calcul d'une image se faisant sur plusieurs jours n'est pas envisageable, comme pour le jeu vidéo par exemple (30 fps). C'est donc vers le rendu temps réel que ce genre d'applications vont se tourner, et c'est ce que nous allons aborder dans notre TER.

Plus précisément, nous allons aborder les différentes techniques permettant d'obtenir un bon compromis entre temps de calcul d'une seule image et qualité de rendu numérique, le tout à partir d'une scène 3D sur ordinateur. Cela passera par la prise en main des différentes possibilités de rendu en 3D temps réel existantes (Blinn-Phong, BRDF ...), de ce qu'elles permettent ou pas de faire, puis implémenter certaines de leurs caractéristiques, les comparer (sur leur aspect qualitatif, effet visé…). Tout cela aura pour but d'en entrevoir leurs limites, mais également l'ensemble des possibles qu'elles allouent (type de matériaux et effets implémentables, réaction à la lumière, type d'éléments nécessaires à l'élaboration de ce type de rendu ...).

Chapitre 2

Détails techniques

Pour passer d'une scène en trois dimensions d'un ordinateur vers une image affichée à l'écran de celui-ci, on doit passer par un pipeline graphique transformant ces données 3D en une matrice de pixels 2D, image finale de notre rendu. Lorsque l'on parle de rendus informatiques, il existe deux grandes familles de techniques à proprement parler, le lancer de rayons et la rastérisation.

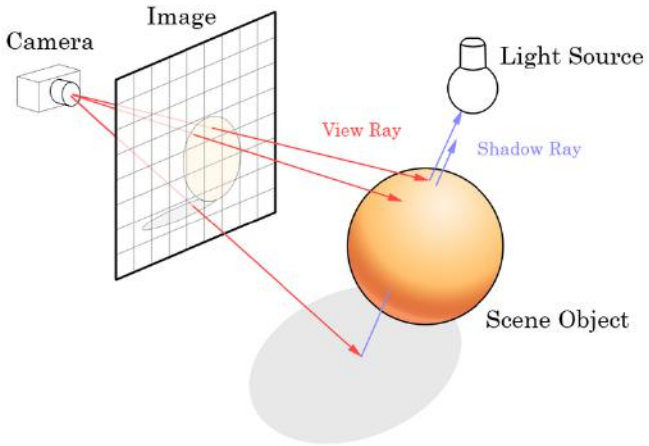

Figure 1 : schéma du lancer de rayons.

Le lancer de rayons peut trouver des implémentations en temps réel pour des rendus réalistes dans les solutions modernes de rendu informatique, mais a historiquement connu une utilisation plutôt basée sur des rendus précalculés, n'offrant pas d'interactions avec un utilisateur. Pour la définir simplement, c'est une technique pour laquelle on va se voir lancer, pour chacun des pixels de l'image que l'on cherche à rendre, une série de rayons (origine, direction et pas de déplacement) puis venir calculer les interactions entre ces rayons et les éléments de la scène en 3D. Ces rayons peuvent ensuite rebondir dans la scène, et de manière récursive, venir entrer en collision avec d'autres éléments.

Durant ce projet, nous n'allons pas manipuler de techniques basées sur le lancer de rayons, car trop complexes à mettre en place pour obtenir des gains significatifs en termes de rendu, tout en conservant un aspect temps réel.

Une autre technique de rendu existante se base cette fois-ci sur le principe de rastérisation. On va, au lieu de partir de l'image que l'on cherche à rendre, partir des éléments 3D de la géométrie des objets de la scène (les fragments), et venir projeter chacun de ces éléments dans l'espace de notre image finale. Pour ce faire, on peut se baser sur un pipeline graphique existant, proposant déjà toute une série d'étapes pour arriver au rendu final, ou partir d'un pipeline graphique artisanal, où on viendra définir chacune des étapes de génération d'une image numérique.

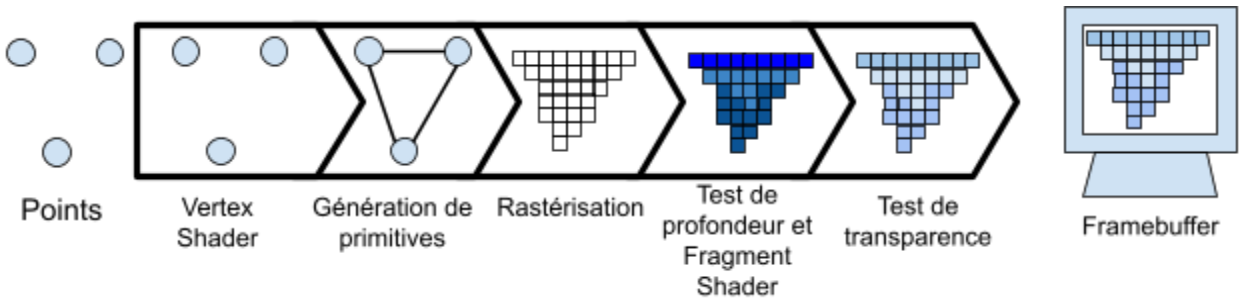

Figure 2 : schéma d'un pipeline de rendu.

Nous allons, dans ce projet, partir sur une solution de rendu basée sur la rastérisation en partant d'un pipeline graphique existant, WebGL. Même si nous nous basons sur un pipeline existant, celui-ci ne nous fournit que les fonctionnalités basiques de rendus sur la carte graphique. Il nous conviendra de devoir définir nous-même le traitement que l'on veut appliquer à nos objets 3D de notre scène afin de les rendre selon des critères précis.

2.1 Spécificités du langage

Comme énoncé précédemment, nous allons vous présenter les travaux de recherches et implémentations que nous avons fait pour mettre en place une solution de rendu en WebGL. WebGL étant une solution de rendu graphique dérivée d'OpenGL, mais pour le WEB, nous allons implémenter notre solution avec des langages issus du domaine du WEB.

Pour les pages informatiques, nous développons en HTML, CSS pour style et JavaScript pour tout le côté logique graphique et mise en place du rendu de notre solution. Nous allons également manipuler des fichiers GLSL servant à la génération de shaders. Ces fichiers sont des petits scripts que l'on va envoyer à notre carte graphique afin qu'elle opère le traitement de rendu que l'on souhaite dans le pipeline WebGL. Ils possèdent leur propre syntaxe, proche du C, et sont également présents en OpenGL. Les deux principaux shaders que nous manipulons sont les shaders de sommets et de fragments. On peut combiner ces deux programmes en un

seul afin de générer un unique programme que l'on utilisera au moment de rendre des éléments (détaillé en partie 3.2).

Le choix d'opérer une solution de rendu à partir de WebGL nous permet de pouvoir mettre en place et partager efficacement des solutions informatiques, de par la nature même du WEB. Il existe tout un ensemble de fonctionnalités permettant de gérer les différents événements utilisateurs depuis le DOM, rendant la mise en place de solutions interactives plus structurées et intuitives (gestion des événements souris, claviers). Le déploiement de notre solution est simple, car il suffit simplement de lancer le fichier "index.html".

De plus, nous avons surtout eu l'occasion de développer des solutions informatiques en OpenGL et C++ durant notre cursus, et basculer sur une solution en WebGL représente un défis supplémentaire, stimulant à la réalisation de ce projet.

La principale composante HTML de notre projet est la balise "canvas". Elle diffère des autres balises présentes en HTML par le fait qu'elle intègre à des pages internet la possibilité de dessiner du contenu graphique en deux et trois dimensions. Dans notre cas, elle nous permet d'afficher directement le résultat des rendus que l'on opère via WebGL :

```
<div id="canvas-holder">
   <canvas id="webglcanvas"></canvas>
</div>

[...]

let gl; // Le WebGL2RenderingContext attaché au canvas.
let canvas; // Le HTMLCanvasElement de la page.

canvas = document.getElementById("webglcanvas");
gl = canvas.getContext("webgl2") || canvas.getContext("experimental-webgl");
```

Pour le débogage, il est possible d'afficher des informations directement depuis une console intégrée au navigateur, et accessible en inspectant une page informatique.

JavaScript est, comme son nom l'indique, un langage de type script, ce qui veut dire qu'il ne nécessite pas de phase de compilation pour pouvoir être exécuté. En effet, un programme dans ce langage est simplement interprété par l'élément qui va l'exécuter, en l'occurrence le navigateur. Il n'y a donc pas besoin de devoir passer par des possibles longues phases de compilation pour tester une fonctionnalité implémentée.

2.2 Architecture des fichiers et classes

Afin de générer des rendus en temps réel, nous avons structuré les différents éléments qui composent notre solution sous forme de classes importantes. La notion de programmation objet diffère en JavaScript par rapport à un langage de programmation classique comme Java (dont il est la version “script”). En effet, les objets sont considérés en JavaScript sous forme de chaînes de prototypages. On peut donc augmenter à volonté le comportement et données d'une instance afin de lui attribuer un traitement particulier et de lui faire faire des actions différentes d’une autre instance de la même classe. Cela permet aussi de redéfinir des comportements existants dans le prototype de la classe en elle-même, et lui ajouter dynamiquement des méthodes et attributs supplémentaires.

Figure 3 : Exemple de chaîne de prototypage.

On remarque ici que les deux instances “humanClassic” et “humanWithAge” de la même classe “Human”, n’ont pas les mêmes valeurs, car on a rajouté à l’instance “humanWithAge”, et non la classe “Human”, un attribut supplémentaire.

Nous avons donc séquencé notre code en plusieurs classes importantes pour notre rendu. Dans un premier temps, nous avons des classes nous permettant de générer des formes utiles pour les afficher ensuite à l’écran. Nous avons également des classes permettant de structurer les différents shaders de rendus. En effet, nous avons jugé important de dissocier les programmes de rendu du reste des informations de la scène, car il s’agit de programmes qui vont indiquer un traitement à suivre côté rendu graphique, indépendamment du contenu de l’information qu’ils peuvent recevoir (mais pas du type d'informations).

Nous avons aussi mis en place une série de méthodes utiles pour l'affichage d'informations (création de sliders) et de récupération/traitement de données (chargement d'images, compilation de GLSL).
Étant donné la forte modularité au niveau des instances que le concept de chaîne de prototypage induit, il n'est pas nécessairement obligatoire de générer une nouvelle classe pour traduire un type unique d'instance. On peut simplement indiquer à l'instance que l'on cherche à spécifier, le comportement et données en plus par rapport à l'instance de base, dont on sait qu'elle en est propriétaire pour le traitement futur.

2.3 Techniques de rendus

Nous avons déjà abordé le fait qu'il existe deux grands moyens de faire du rendu depuis une scène en trois dimensions. Cependant, la manière de les implémenter peut varier d'une technique à l'autre. En effet, en ce qui concerne la rastérisation, il est possible d'opérer notre rendu des primitives graphiques en une seule passe de rendu. Cela consiste à prendre toutes les informations importantes de la scène (lumières, caméra, maillages) et de les traiter par le biais d'un unique programme. On va par exemple calculer l'apport d'une lumière de la scène pour chaque point de la scène. Cela peut demander beaucoup de calculs, dont des calculs inutiles. En effet, on peut se retrouver dans notre exemple à devoir calculer l'apport des lumières en un point qui ne sera de toutes façons pas visible à l'écran (car ils sont positionnés derrière un objet du point de vue de notre caméra).

Pour pallier ce problème et gagner en performance (Les calculs les moins coûteux sont ceux que l'on ne fait pas.), on peut implémenter des techniques de rendu en plusieurs passes différentes. Ce type de rendu s'appelle le rendu différé. Au lieu d'afficher le résultat d'une passe de rendu directement sur l'écran, on va venir la récupérer et la placer dans une texture. Une texture peut être vue comme une image en deux dimensions, contenant des informations par pixels. Il est possible en rendu 3D de venir récupérer ces informations pour ensuite appliquer un traitement par pixel sur cette image, ou par exemple venir la plaquer sur une forme via des coordonnées de textures.

Dans notre cas, nous allons récupérer des informations liées à chaque élément de notre scène pour ensuite venir n'appliquer qu'une seule fois le calcul de la lumière pour chaque fragment visible. Cela va considérablement réduire le temps de calculs dès lors que l'on va se retrouver avec des scènes denses en géométries.

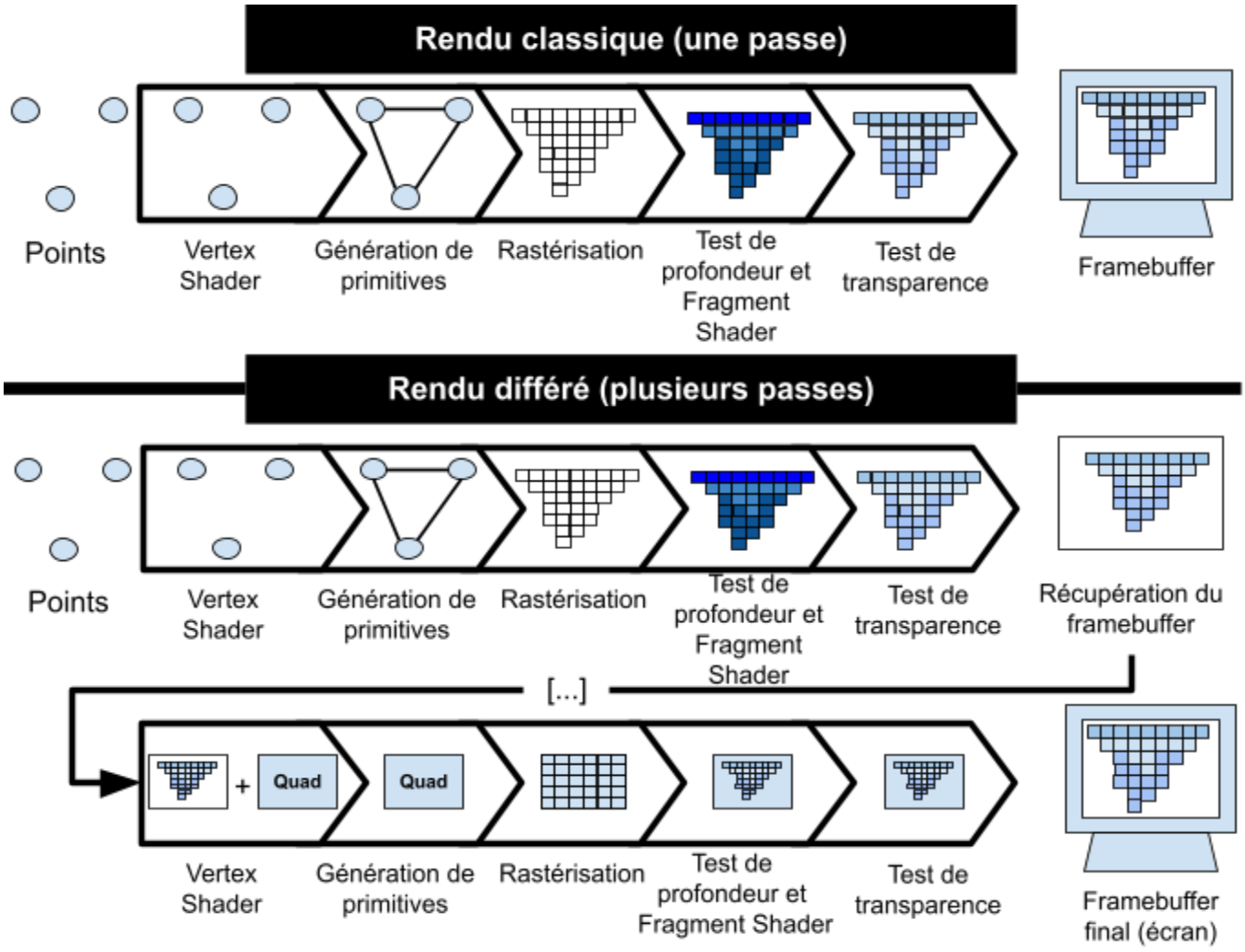

Figure 4 : Rendu classique contre rendu différé.

Par la suite, on verra également que pour rendre certains traitements 3D, il devient nécessaire de passer par ce type de rendu, comme pour le calcul des ombres portées par exemple.

Chapitre 3

Création d'un moteur de rendus

Maintenant que nous avons décrit la manière dont, partant d'une scène définie numériquement et en 3D, il est possible de générer une image affichable ensuite sur un écran, nous allons aborder la solution que nous avons mise en place, en commençant par ses différents composants.

3.1 Composants principaux d'un moteur de rendu

3.1.1 Monde en 3D

Nous cherchons à rendre un monde en trois dimensions. La structure de données classique pour représenter des données issues d'un tel environnement sont par le biais de triplets de valeurs, ou des vecteurs 3D. On se sert de ces éléments pour représenter des coordonnées dans l'espace, ou des vecteurs de déplacement ou d'orientation par exemple.

On utilise aussi des vecteurs en quatre dimensions (xyz et w), qui sont des vecteurs de coordonnées homogènes de vecteurs 3D. Les coordonnées homogènes se reposent sur une notation dans laquelle les vecteurs en N dimensions sont représentés par un vecteur en N+1 dimensions. Cela permet de pouvoir caractériser plus simplement des transformations dans l'espace.

Pour les matrices de transformations, on utilise principalement des matrices 3x3, ou 4x4 dans leur version homogène.

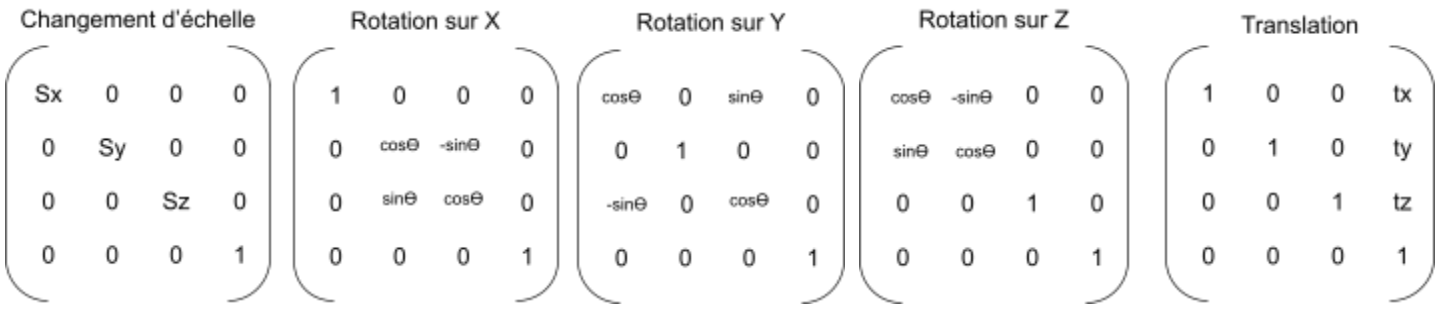

Figure 5 : Matrices de transformation en coordonnées homogènes

Afin de nous permettre de manipuler ces différentes structures de données, nous utilisons une bibliothèque dédiée, "glmatrix". La particularité principale dont il faut tenir compte avec cette librairie, est que les matrices sont représentées en column-major, soit colonne par colonne.

3.1.2 Lumières et caméra

Sans lumières, les objets sont continuellement dans l'ombre, et nous ne les voyons pas. De plus, sans un point de captation dans la scène, nous ne pouvons l'observer et donc la rendre. Nous représentons ces deux éléments par le biais de lumières et de caméra disposées dans la scène.

Pour les lumières, nous les représentons par un vecteur 3D symbolisant ses coordonnées dans la scène, d'une couleur et de facteurs d'atténuations lumineuses (linéaire et quadratique). La manière dont la lumière va éclairer les objets de la scène due à l'équation que l'on va mettre en place pour en calculer son apport quant à la couleur finale de l'objet, peut se décomposer simplement dans le calcul de deux grandes réflexions :

- Spéculaire : le rayonnement lumineux réfléchi par la surface de notre objet l'est dans une seule et même direction (Descartes).
- Diffuse : la surface de notre objet n'est pas complètement lisse, aussi la réflexion de la lumière n'est pas complètement nette, induisant une réflexion diffuse.

Concernant l'objet caméra, il va être placé dans la scène, au même titre que les autres maillages à rendre. Il est défini par trois vecteurs 3D, que sont sa position dans l'espace, son vecteur de direction vers le haut, et l'endroit vers lequel la caméra regarde.

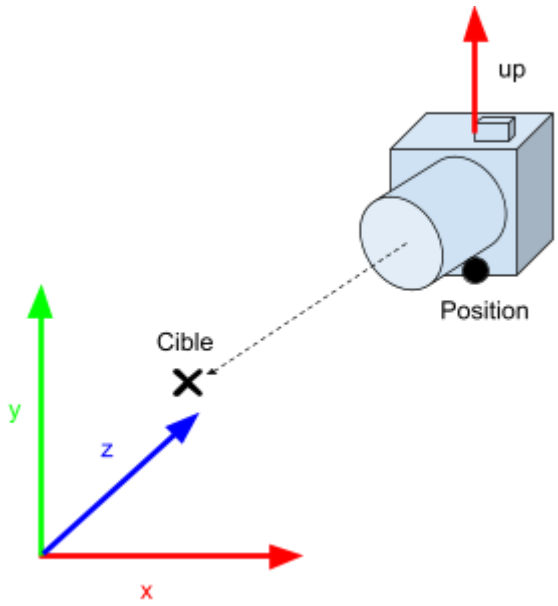

Figure 6 : Représentation de la caméra.

Ces informations nous permettent de composer la matrice de vue, composante essentielle du rendu. Par la suite, nous allons également devoir choisir une perspective de représentation pour composer la matrice de projection. Ces deux matrices vont nous permettre de passer d'objets définis dans notre scène, avec des coordonnées globales (ou monde), à des

objets dans un premier temps définis dans le repère de la caméra, puis projetés sur l'image à rendre par choix de perspective.

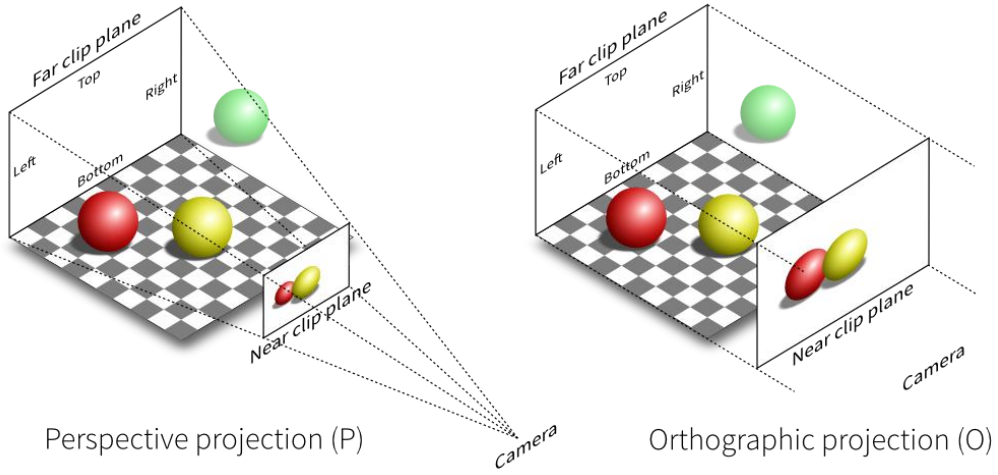

Figure 7 : Projection perspective et projection orthographique.

Pour chercher à calculer l'endroit où va se situer notre objet 3D de notre scène dans notre image finale, on peut donc opérer une série de projections par multiplications matricielles entre la position de celui-ci dans la scène et les matrices de vue et de projection.

Ces deux dernières sont calculées directement depuis le point d'observation et le type de projection souhaitée.

```
layout (location=0) in vec3 aVertexPosition;

uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;


void main(void) {
    gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix * vec4(aVertexPosition, 1.0);
}
```

Voici un exemple de calcul de ces différentes projections dans un programme GLSL. Vu qu'il est possible de calculer cela par des multiplications matricielles, il est possible de faire le calcul inverse par l'inverse des matrices utilisées, afin de partir de la coordonnée écran vers la coordonnée dans la scène. Cela est très pratique quand on cherche à sélectionner des objets depuis le curseur de la souris (picking). Nous avons implémenté ceci afin de rendre notre scène plus interactive et pouvoir visualiser plus facilement les effets de rendu à la lumière.

3.1.3 Modèles en 3D

Un maillage est une forme géométrique dans l'espace. Il est représenté par des notions de géométrie (sommets) et d'associations entre ces sommets (arêtes et faces). Un sommet d'un maillage possède donc des coordonnées dans l'espace, mais également des données liées à son orientation (normale) ou à la manière dont nous allons lui appliquer des textures (coordonnées de texture). Il est également possible d'avoir des normales pour chacune des faces de l'objet. Ce maillage va venir représenter notre objet numériquement.

Pour procéder à l'orientation des sommets, nous passons par la connectivité du maillage. En effet, un maillage est souvent composé de face triangulaires. Il est donc facile, connaissant les arêtes qui les composent, de retrouver leurs normales par le produit vectoriel. La normale d'un sommet devient donc la réunion de toutes les normales des faces auxquelles il appartient.

Ces informations sont primordiales pour calculer ensuite le bon rendu pour une forme 3D. Lors du rendu d'un maillage, nous allons passer au pipeline de rendu les sommets, avec leurs informations de normales, afin de calculer ensuite dans le fragment shader l'apport de la lumière sur la forme par exemple, car plus une face est directement orientée vers la lumière, et plus elle sera éclairée par celle-ci.

Les coordonnées des différents points que compose la donnée d'un modèle sont définies localement par rapport à l'objet. En effet, pour pouvoir avoir leurs coordonnées dans la scène (coordonnées modes), on passe par la matrice "modèle" de l'objet.

3.1.4 Scène

Afin de structurer les modèles et les lumières, une scène est mise en place. Elle contient une liste de modèles, une liste de lumières, et la caméra principale. Pour faire un rendu sur la scène, il est nécessaire de parcourir la liste des modèles et des lumières en utilisant la caméra. La scène peut également se mettre à jour, c'est-à-dire qu'elle va mettre à jour les données de la caméra, des modèles et des lumières.

3.2 Instructions GPU : Les shaders

Les shaders sont des programmes qui sont exécutés sur carte graphique. Ils possèdent donc des optimisations différentes que pour un programme sur processeur. Dans les termes WebGL, on manipule un "program**"**. Un "program" est capable, à partir d'un maillage et de différentes valeurs appelées "uniform**"**, d'obtenir une image. Pour cela, il s'aide de "varying", des valeurs passées du "vertex shader" au "fragment shader".

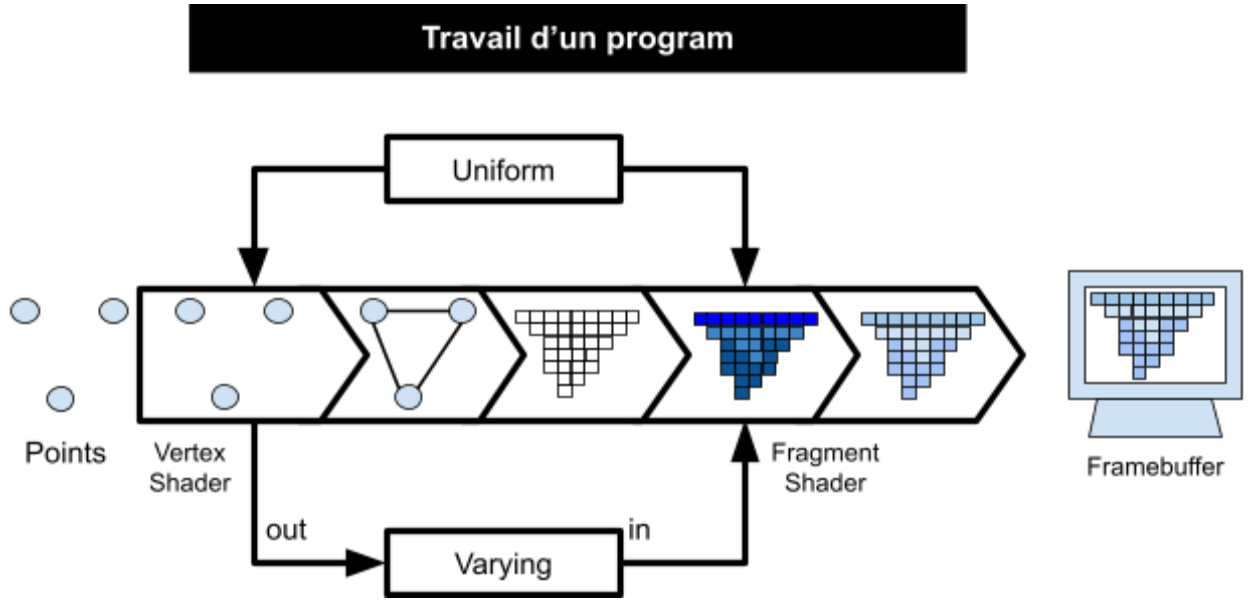

Figure 8 : Représentation d'un "program" WebGL.

Exemple :

Vertex shader : on prend un maillage ainsi que les informations de la caméra principale.

```
#version 300 es
precision highp float;

layout (location=0) in vec3 aVertexPosition;
layout (location=1) in vec3 aVertexNormal;
layout (location=2) in vec2 aVertexUV;

uniform mat4 uModelMatrix;
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;
uniform mat4 uNormalMatrix;

out vec3 vNormal;
out vec3 vFragPos;
out vec2 vFragUV;

void main(void) {
    vFragPos = vec3(uModelMatrix * vec4(aVertexPosition, 1.0));
    vNormal = vec3(uNormalMatrix * vec4(aVertexNormal, 0.0));
    vFragUV = aVertexUV;

    gl_Position = uProjectionMatrix * uViewMatrix * vec4(vFragPos, 1.0);
}
```

Fragment shader : on récupère ces informations, mais on n'affiche que la position (en coordonnée monde).

```
#version 300 es
precision highp float;
// Fragment-Interpolated data
in vec3 vNormal;
in vec3 vFragPos;
in vec2 vFragUV;

out vec4 FragColor;

void main(void) {
    FragColor = vec4(vFragPos, 1.0);
}
```

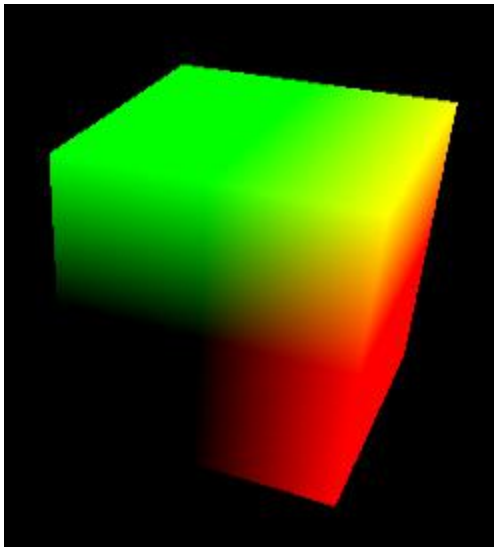

Figure 9 : Exemple de rendu des coordonnées mondes d'un maillage.

3.2.1 Encapsulations

Utiliser un shader requiert divers pré-traitements qui peuvent être répétitifs. On a donc créé une classe englobant ces traitements. Cela nous permet de créer et de mettre à jour leurs données. Pour créer un shader il suffit de donner les liens vers les fichiers du shader. La classe englobante va alors chercher et récupérer le contenu du fichier, compiler les fichiers et enfin créer l'objet webgl "program" et y attacher les fichiers. Ensuite, il faut déclarer les "uniform" que l'on va utiliser. Cela permet de préparer leurs positions pour que l'on puisse y accéder. Enfin, lors du rendu des objets, il faut mettre les valeurs que l'on souhaite dans ces "uniform".

Prenons l'exemple du "geometry shader", ou shader de géométrie. Il a pour objectif de générer des textures correspondant aux informations géométriques et spécifiques aux objets. Il va donc rendre tous les modèles de la scène et produire en sortie quatre textures : les positions en coordonnées monde, les normales (orientation des sommets), les couleurs et l'intensité spéculaire des modèles visibles par la caméra.

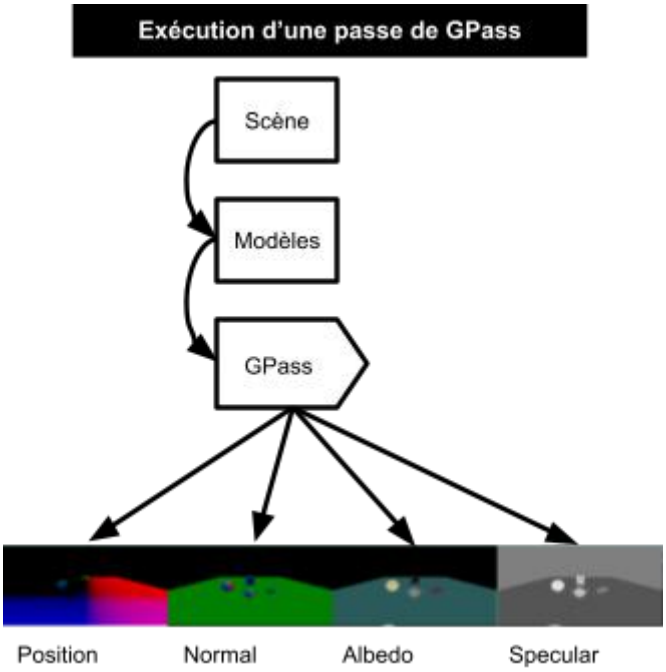

Figure 10 : Visualisation de la passe géométrique.

Le rendu se fait alors sous cette forme :

```
scene.models.forEach(model => {
            if (this.shouldRenderOnModel(model)) {
                this.setModelData(model);
                model.render();
            }
        });
```

La fonction `shouldRenderOnModel(model)` permet d'effectuer un filtrage sur les modèles de la scène. On peut par exemple décider de ne pas rendre les modèles qui ont l'attribut "invisible". Ensuite, on a `setModelData(model)` qui met à jour les "uniform" du shader, spécifiques à l'objet (matrice de modèle, matrice de normales, texture de couleur et d'intensité spéculaire). Enfin, on fait le rendu de l'objet avec `model.render()`.

3.2.2 Un pipeline de rendu dédié

Afin d'avoir un traitement générique nous permettant de rendre les shaders en plusieurs passe et sur différentes scènes, nous avons mis en place un système de pipeline de traitement.

Un "Pipeline" tel qu'on le conçoit, contient une liste de "ShaderRenderer**"**, ainsi qu'une liste de "ShaderRendererResult". Un "ShaderRenderer" représente une passe, et va donc générer une ou plusieurs textures. Afin de faire le rendu du "Pipeline", on fait un rendu de tous les "ShaderRenderer**"** dans l'ordre, on fait donc plusieurs passes. Chacune de ces passes va lire des textures générées dans des passes précédentes et va ajouter de nouvelles textures.

C'est dans un souci d'organisation de ces résultats que le "Pipeline" possède la liste de "ShaderRendererResult". Cette classe représente donc le résultat d'une passe. Ses instances possèdent donc la texture générée par le "ShaderRenderer".

Sur un plan plus technique, il ne possède pas que la texture, mais aussi d'autres données sur la façon dont la texture a été générée. Par exemple, il possède aussi la caméra qui a été utilisée pour le rendu (point de vue). Cela permet à une future passe de récupérer les matrices qui ont servi à effectuer ce rendu. Ce mécanisme est utilisé pour effectuer le calcul des ombres par exemple.

Pour un simple rendu différé, le pipeline est composé de trois "ShaderRenderer**"**. Premièrement, on fait une passe géométrique nous permettant d'extraire de la scène les coordonnées mondes, les normales, la couleur et l'intensité spéculaire de chaque pixel de l'écran. Ensuite, on fait un calcul de lumière en utilisant les textures calculées précédemment et les lumières de la scène. Enfin, on applique le résultat sur l'écran.

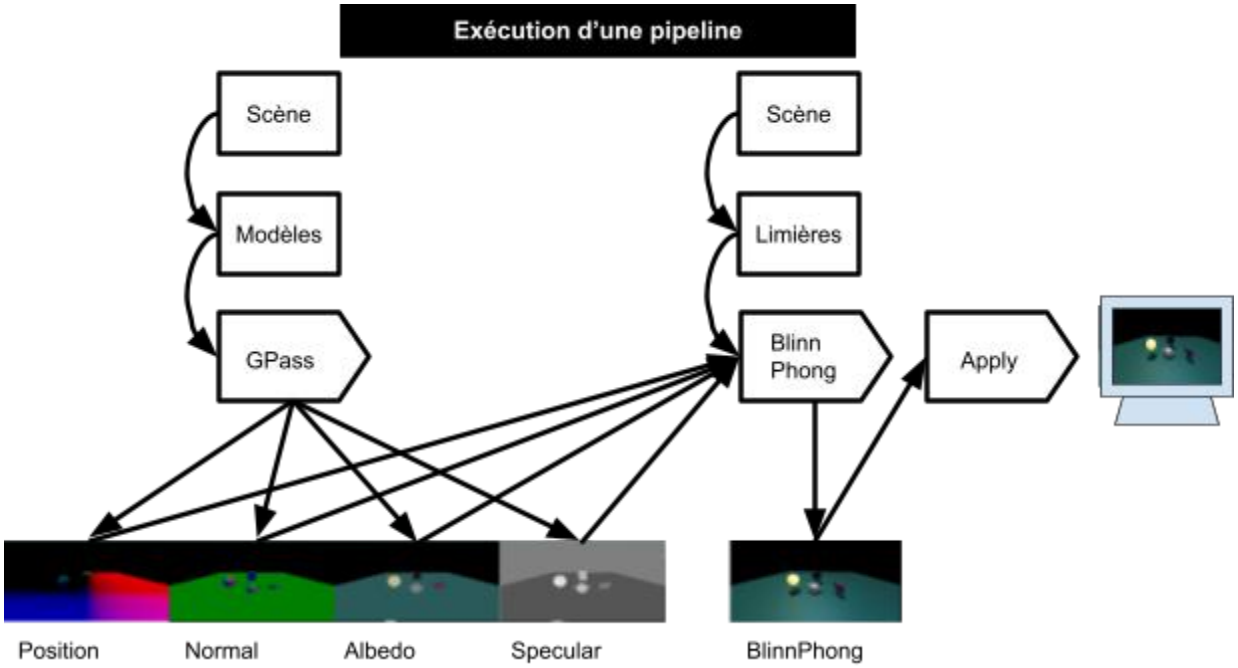

Figure 11 : Visualisation d'un pipeline de rendu différé.

3.3 Interface utilisateur

Mettre en place un moteur graphique est utile, mais on souhaite aussi permettre à l'utilisateur de modifier les paramètres des différentes passes dynamiquement. Pour cela, un menu étirable a été conçu.

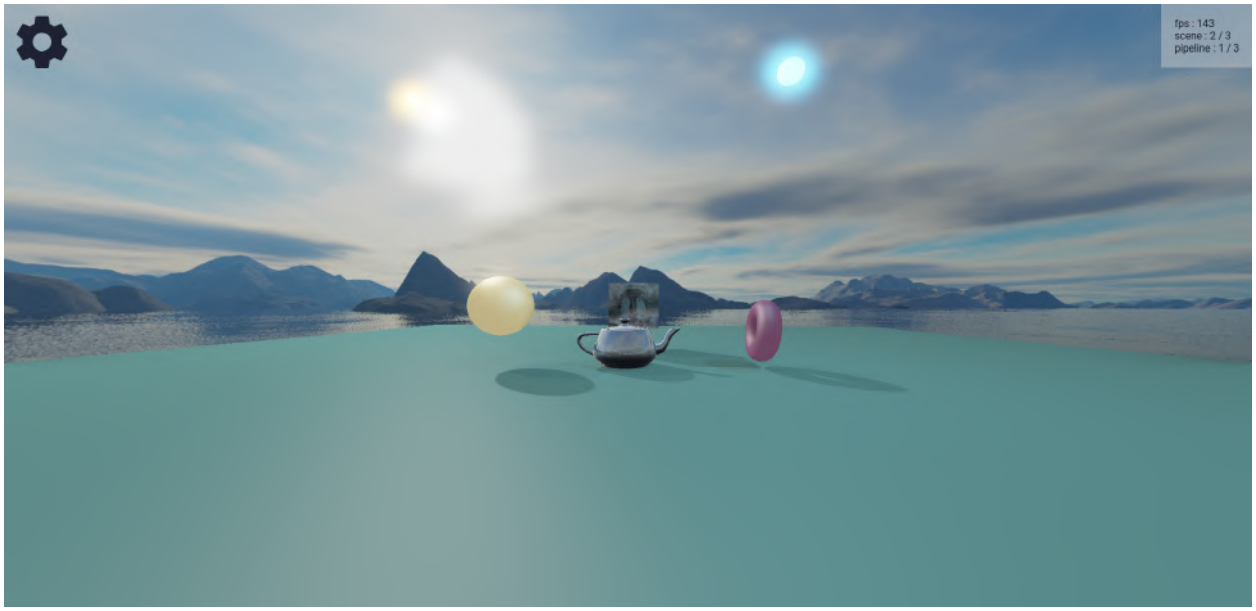

Figure 12 : Visualisation d'une scène présentant les éléments d'interface.

L'utilisateur peut se déplacer dans la scène. Il lui suffit pour cela de bouger la souris avec le clic droit enfoncé. La caméra va alors se déplacer en orbitant autour du centre de la scène rendue. On peut également se rapprocher ou s'éloigner du centre en utilisant la molette de défilement. Enfin, l'utilisateur peut déplacer les objets de la scène grâce au clic gauche.

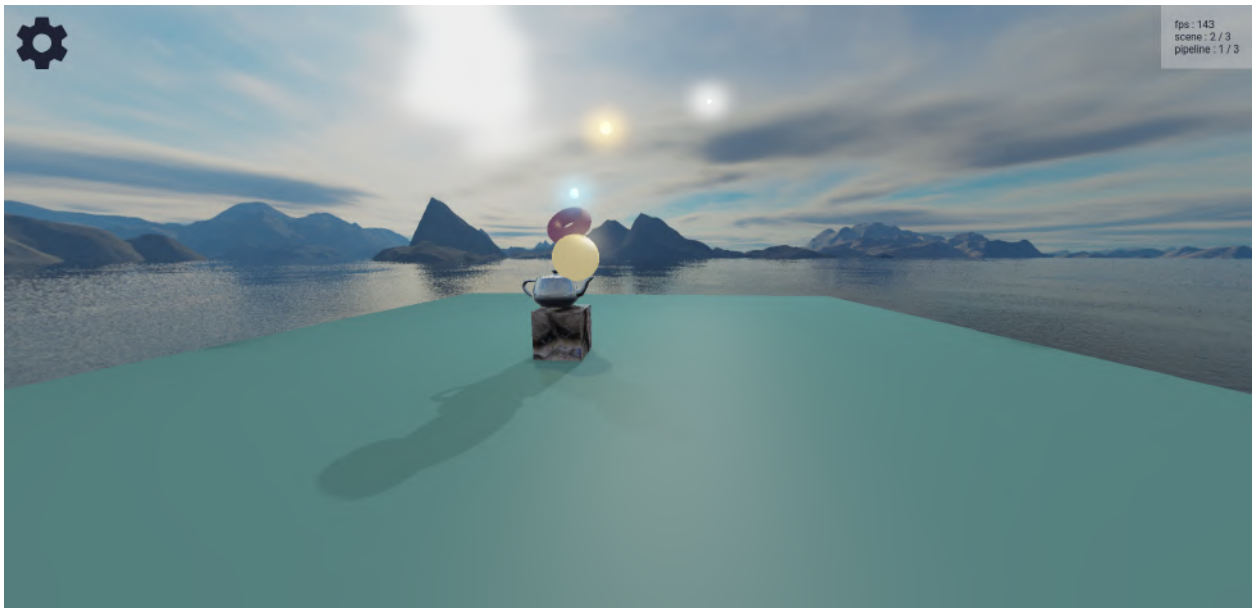

Figure 13 : Résultat d'une manipulation de la scène.

L'action de déplacer un objet en le sélectionnant avec un clic est rendue possible par la mise en place de boîtes de collision et de lancer de rayon. Les boîtes de collisions sont une représentation simplifiée de maillages souvent complexes, et donc difficilement analysables géométriquement.

Figure 14 : Exemple de représentations simplifiées (boîte et sphère).

Pour déterminer notre rayon à lancer (origine et direction), on se base sur les coordonnées écran normalisées du point où l'utilisateur clique (penser à inverser l'axe Y), que l'on va projeter sur la scène en 3D par la multiplication inverse des matrices de vue et de projection. Si passer de coordonnées 3D à des coordonnées 2D est relativement simple, l'inverse est moins trivial. En effet, il nous faut reconstruire une coordonnée manquante, la profondeur. Pour cela, on se donne arbitrairement des profondeurs, correspondant aux plans de vue de la caméra (plan proche et plan éloigné).

Concernant l'interface, un menu à gauche est également déroulable (par clic sur l'engrenage). On y trouve les différentes options permettant de changer des éléments de la scène, ou de changer dynamiquement les paramètres des passes.

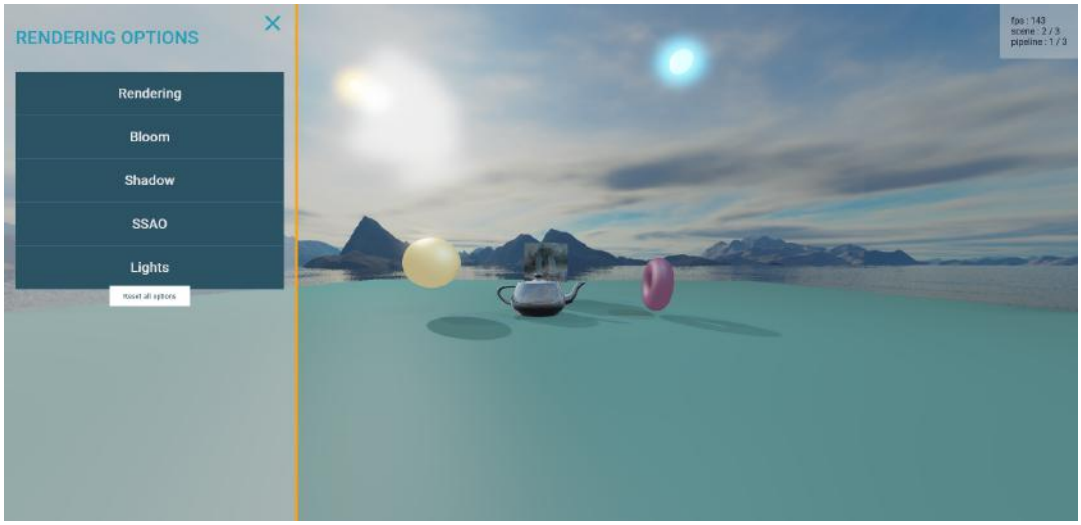

Figure 15 : Visualisation du menu d'édition des paramètres.

À l'aide de ce menu, on peut par exemple modifier la couleur des lumières (voir image en dessous).

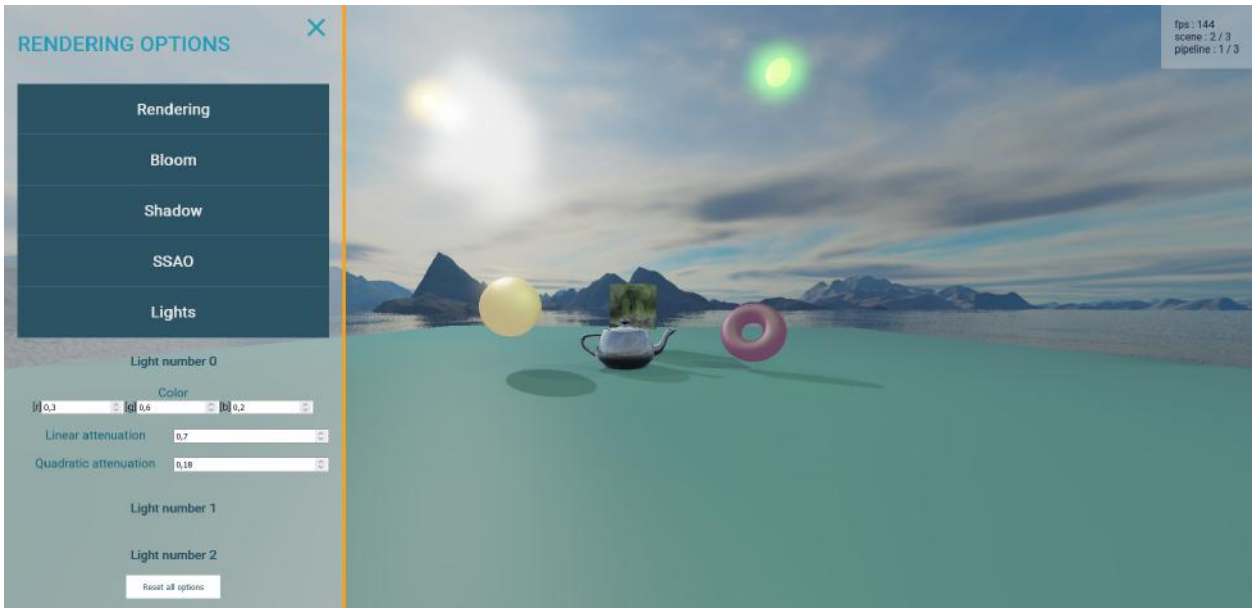

Figure 16 : Changement de la couleur de la lumière.

Ou alors, on peut modifier un attribut des passes, comme par exemple l'exposition, qui simule le réglage de temps d'exposition des caméras, et donc modifie la luminosité perçue.

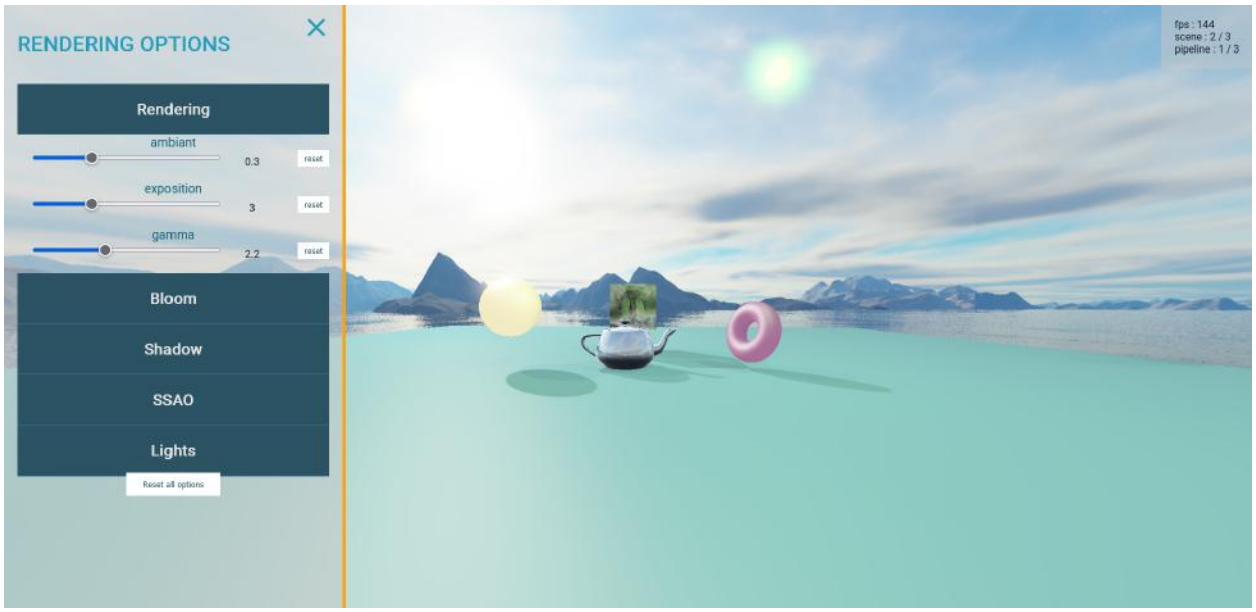

Figure 17 : Changement du paramètre d'exposition.

Enfin, on peut changer de scène ou de pipeline avec les touches “Entrée” et “Retour” respectivement. En effet, on a la possibilité de créer différentes scènes et pipelines. On peut par exemple concevoir un pipeline composé des mêmes shaders que ceux utilisés pour faire le rendu, mais qui, à la fin du pipeline, affiche les différentes textures qui ont été générées, passes après passe. Cela possède un intérêt pédagogique puisqu’on montre les étapes nécessaires pour avoir le rendu final.

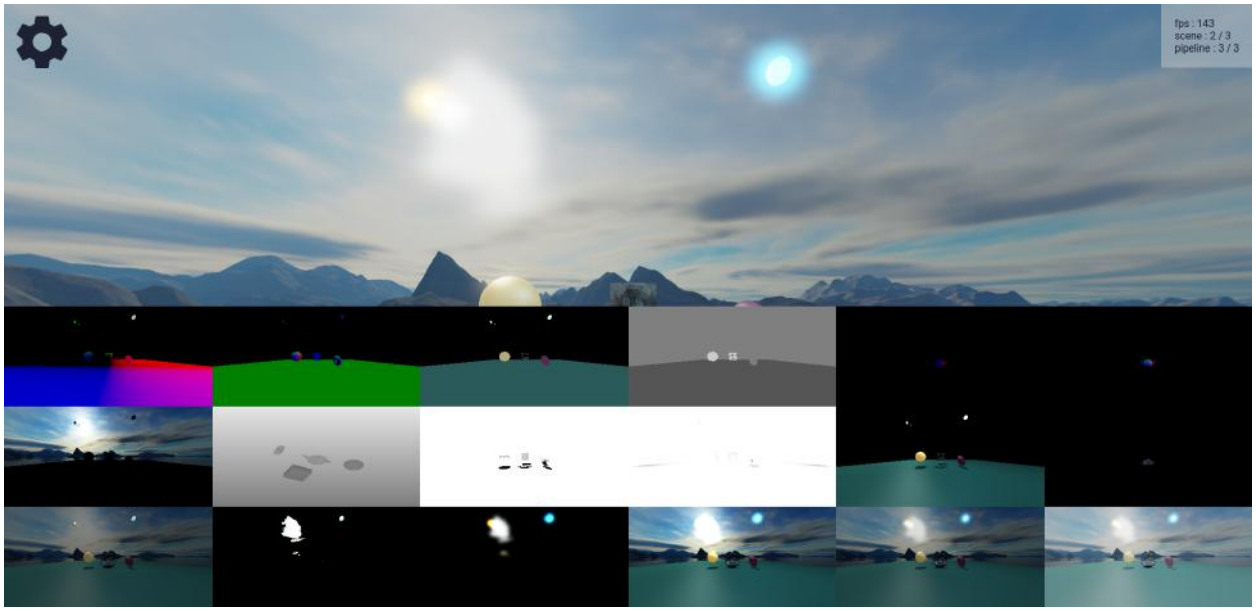

Figure 18 : Rendu de la pipeline d’affichage

La plupart de ces rendus sont expliqués dans la partie suivante. On retrouve, sur la première ligne et de gauche à droite, la position (coordonnée monde), les normales, les couleurs, l'intensité spéculaire, mais aussi la position et la normale des objets réfléchissants. Sur la deuxième ligne, il y a la carte d’environnement (“skybox”), la carte de profondeur pour le calcul des ombres, la carte d’ombrage, l’occlusion ambiante, le calcul de lumière “Blinn Phong**”,** et les couleurs des objets réfléchissants. Sur la dernière ligne, on observe la fusion entre la carte d’environnement, le calcul de lumière et des objets réfléchissants, l’extraction des valeurs de forte intensités, un flou gaussien sur ces valeurs (ce qui constitue le “bloom”), la fusion des couleurs et le “bloom”, le calcul d’exposition et la “correction gamma”.

Chapitre 4

Exemple de shaders et rendus

Maintenant que nous avons détaillé les différents composants d'un rendu 3D appliqués à un moteur de rendu en temps réel, nous allons présenter dans cette partie les différentes implémentations que nous avons réalisées, en expliquant dans un premier temps la théorie derrière l'effet de rendu souhaité, puis les résultats que nous avons obtenus.

4.1 Gamma correction

L'un des premiers effets visuels que nous avons voulu mettre en place est la "correction gamma". Il permet d'augmenter grandement le réalisme d'une scène, pour une complexité de mise en place abordable.

4.1.1 Explication

La "correction gamma" est une technique consistant à adapter l'intensité lumineuse à l'œil humain. En effet, on ne perçoit pas l'intensité lumineuse de manière linéaire :

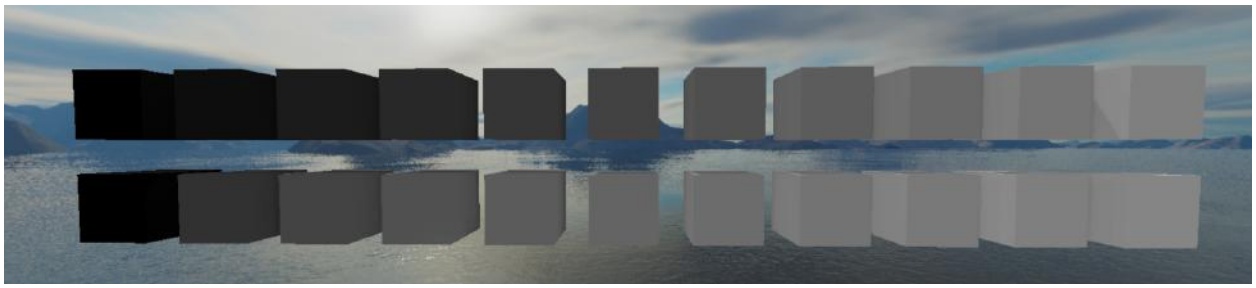

Figure 19 : Rendu de présentation de la correction gamma sur des intensities entre 0.0 et 1.0.

Sur l'image en dessus, la ligne du dessus représente ce que l'œil perçoit comme linéaire, alors que la ligne du bas traduit l'intensité physiquement linéaire. On remarque que l'œil humain est plus sensible aux changements de couleurs sombres qu'aux changements de couleurs claires.

Pour pallier cela, les écrans numériques ont leurs propres corrections appelé "CRT gamma correction 2.2". Historiquement, la valeur 2.2 provient des écrans cathodiques auxquels doubler le voltage ne doublait pas l'intensité. Il y avait une relation exponentielle avec pour valeur constante 2.2 (approximativement).

Ainsi, lors d'un calcul de lumière, on n'affiche pas les couleurs dans un espace linéaire, mais dans l'espace de l'écran ; c'est-à-dire que l'on affiche des couleurs dont l'intensité va être nécessairement diminuée par la correction.
Les rendus obtenus par les calculs ne sont donc pas physiquement corrects. Pour y remédier, il faut donc appliquer une correction avant d'afficher les pixels sur l'écran. Cette correction est l'inverse du traitement effectué par l'écran :

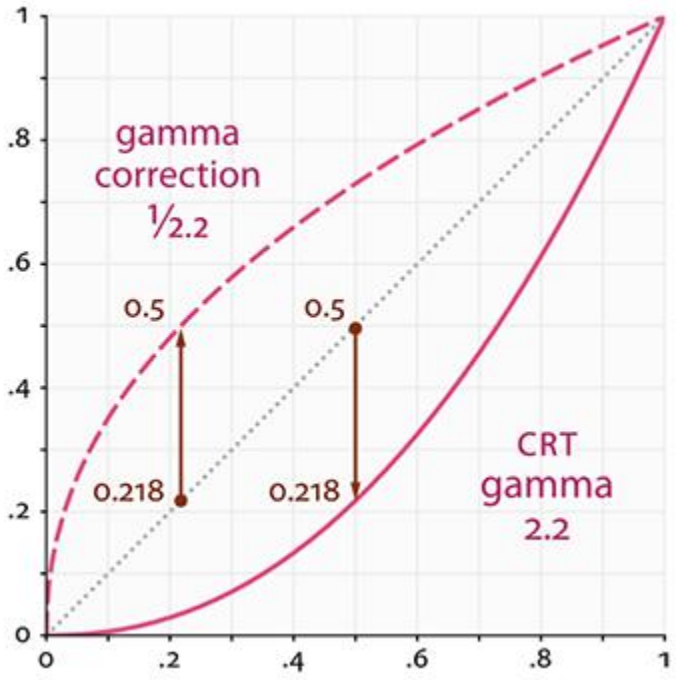

Figure 20 : Schéma explicatif de la correction gamma .

Légende
Ligne en tirets : correction inverse à implémenter
Ligne en points : intensité linéaire
Ligne pleine : correction effectuée par l'écran

Cette correction inverse va éclaircir la plupart des pixels, mais la correction de l'écran va les assombrir pour retrouver une intensité linéaire. On applique en quelque sorte une compensation de la correction initiale faite historiquement par nos écrans.

On peut par ailleurs caractériser la formule de "correction gamma" de la sorte :
Soit I l'intensité lumineuse et g le gamma (2.2), la correction effectuée par l'écran est I^g et la correction inverse est donc $I^{1/g}$.

L'espace des couleurs obtenues par un gamma de 2.2 est appelé espace sRGB. Cependant, cette valeur change de moniteurs en moniteurs (mais reste proche de 2.2), on peut donc demander à l'utilisateur de la modifier dynamiquement.

Cette étape de correction doit être faite à la fin du traitement de rendu, juste avant de rendre sur l'écran. En effet, on ne doit pas faire de calculs de lumières sur les valeurs corrigées, sinon, ces calculs ne seront plus en espace linéaire, mais en espace sRGB.

Enfin, il faut faire attention à la manière dont on charge les textures. En effet, la plupart des textures issues de traitements artistiques ont déjà le traitement gamma. Si on ignore ce fait, on applique finalement deux fois l'inversion gamma. On peut par exemple appliquer le "CRT gamma" lors du chargement des textures.

4.1.2 Implémentation

La correction gamma est un shader qui se rend sur l'écran. Il prend en entrée une texture générée lors d'une phase précédente, et rend une nouvelle texture possédant les couleurs de la précédente avec une correction gamma appliquée.

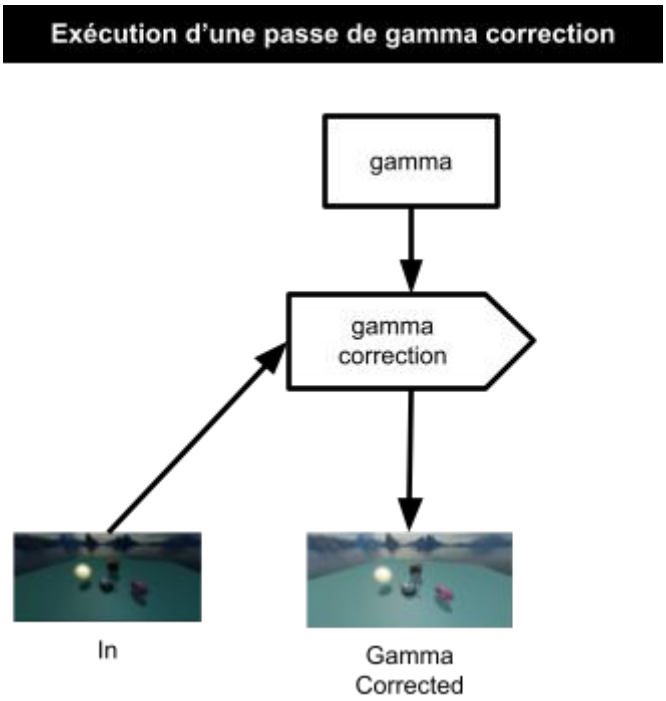

Figure 21 : Visualisation de la passe de correction gamma.

Pour simuler le fait que l'on part d'une image précalculée vers une autre image corrigée, on fait donc le rendu d'un quad sur l'espace écran :

```
#version 300 es
```

```
precision highp float;

layout (location=0) in vec3 aVertexPosition;
layout (location=1) in vec3 aVertexNormal;
layout (location=2) in vec2 aVertexUV;

out vec2 TexCoords;

void main()
{
    TexCoords = aVertexUV;
    gl_Position = vec4(aVertexPosition, 1.0);
}
```

Pour la correction, on l'applique en fonction d'une texture symbolisant l'image à corriger.

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D inputColor;

uniform float gamma;

void main()
{
    //On sépare l'alpha car on ne lui applique pas le traitement

    vec3 color = texture(inputColor, TexCoords).rgb;
    float alpha = texture(inputColor, TexCoords).a;

    FragColor = vec4(pow(color, vec3(1.0/gamma)), alpha);
}
```

4.1.3 Résultats

Figure 22 : Rendu sans la correction gamma.

Figure 23 : Rendu avec la correction gamma.

On remarque que la correction apporte un plus grand réalisme à la scène, et permet de rendre des couleurs plus physiquement proches de couleurs réelles.

4.2 Bloom

Les couleurs affichées sur l'écran ont heureusement une limite. Elles sont comprises entre une valeur minimale et maximale. Cependant, pendant les calculs de lumière, on peut calculer une valeur supérieure à l'intensité maximale de l'écran. Si on essaye d'afficher ces valeurs telles qu'elles, elles vont être ajustées à l'intensité maximale de l'écran, et non à leurs valeurs calculées. Le "bloom" est une technique nous permettant d'effectuer un traitement de ces valeurs.

4.2.1 Explication

Le “bloom” cherche à reproduire un défaut des caméras lorsque le capteur reçoit une luminosité trop grande. En effet, dans la réalité, les lentilles présentent dans bon nombre de dispositifs d’acquisition d’images ne convergent pas la lumière en un seul point, mais sur une zone plus grande appelée tache d’Airy :

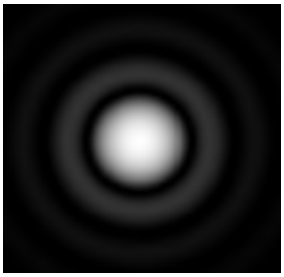

Figure 24 : Exemple de tache d'Airy simulée par ordinateur.

Cela n’a pas de grandes conséquences pour des faibles intensités de lumière, mais devient visible lorsque ces intensités deviennent plus grandes.
Pour réaliser un bloom, on passe par trois étapes. Premièrement, on extrait les valeurs au-dessus d’un seuil dans une texture. Ensuite, on applique l’aberration de la lentille grâce à une convolution avec la tache d’Airy. Cependant, cette tache est coûteuse à calculer. Heureusement, elle est assez proche d’un flou gaussien, on fait donc une approximation de la tache par un flou gaussien :

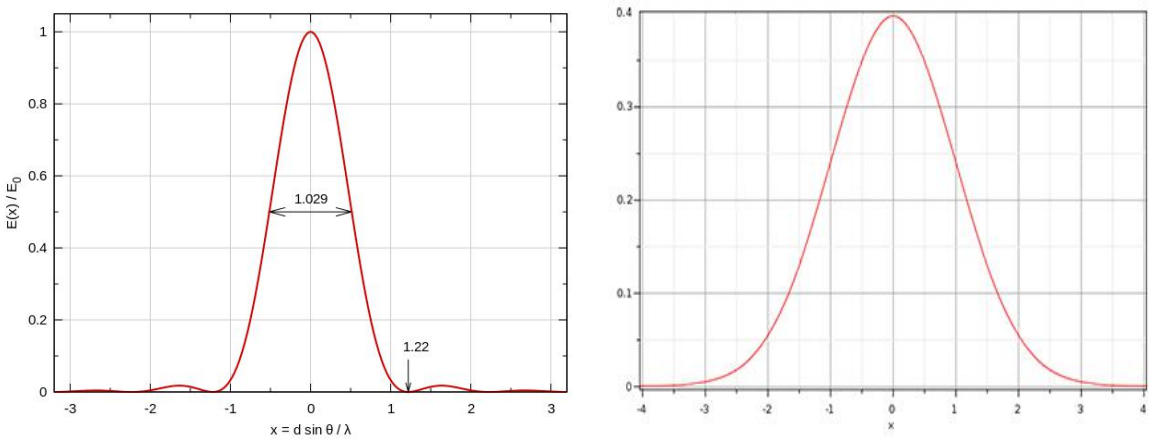

Figure 25 : Comparaison entre la tache d’Airy et une gaussienne.

Enfin, en dernière étape, on additionne les couleurs de base avec l’image floutée.

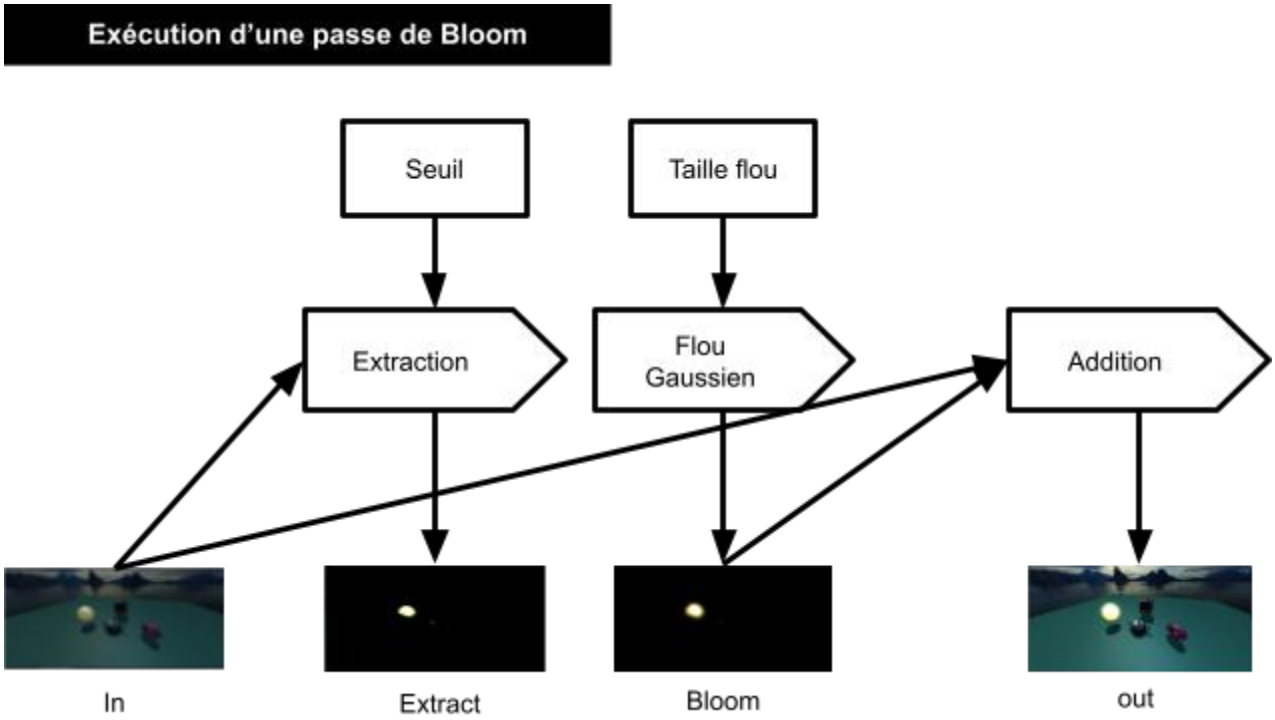

Figure 26 : Visualisation de la passe de bloom.

4.2.2 Implémentation

Tous ces shaders sont dans l'espace écran, on rend donc un quad avec le même vertex shader.

Première phase : Extraction

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D inputColor;

uniform float seuil;
uniform vec4 defaultColor;

void main()
{
```

```
    //On sépare l'alpha car on ne lui applique pas le traitement

    vec3 color = texture(inputColor, TexCoords).rgb;
    float alpha = texture(inputColor, TexCoords).a;

    float brightness = dot(color.rgb, vec3(0.2126, 0.7152, 0.0722));

    FragColor = brightness > seuil ? vec4(color, alpha) : vec4(defaultColor);
}
```

Pour le flou gaussien, on utilise la technique proposée par LearnOpentGL (voir lien en annexe). On fait donc plusieurs passes en enchaînant les passes horizontales et verticales.

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D inputColor;

uniform bool horizontal;
uniform float weight[5];

void main()
{
    //On sépare l'alpha car on ne lui applique pas le traitement

    vec3 color = texture(inputColor, TexCoords).rgb;
    float alpha = texture(inputColor, TexCoords).a;

    vec2 tex_offset = 1.0 / vec2(textureSize(inputColor, 0)); // gets size of
single texel
    vec3 result = texture(inputColor, TexCoords).rgb * weight[0]; // current
fragment's contribution
    if(horizontal) {
        for(int i = 1; i < 5; ++i) {
            result += texture(inputColor, TexCoords + vec2(tex_offset.x *
float(i), 0.0)).rgb * weight[i];
            result += texture(inputColor, TexCoords - vec2(tex_offset.x *
float(i), 0.0)).rgb * weight[i];
        }
    }
    else {
```

```
        for(int i = 1; i < 5; ++i) {
            result += texture(inputColor, TexCoords + vec2(0.0, tex_offset.y *
float(i))).rgb * weight[i];
            result += texture(inputColor, TexCoords - vec2(0.0, tex_offset.y *
float(i))).rgb * weight[i];
        }
    }

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

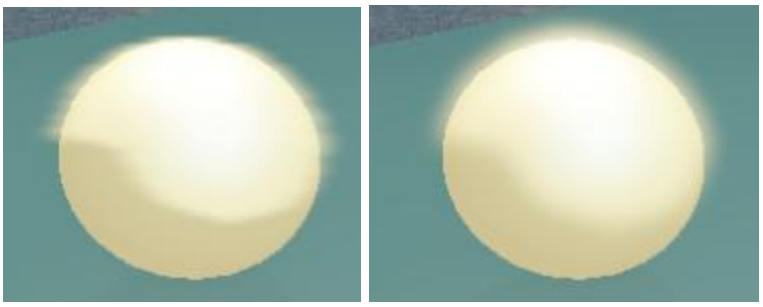

Figure 27 : Gauche : 1ère passe horizontale, Droite : 2eme passe verticale

De plus, comme on passe par un framebuffer, on peut diminuer la résolution des textures. Cela permet de diminuer le temps de calcul du flou gaussien, et nous fait donc gagner en performances.

4.2.3 Résultats

Figure 28 : Rendu sans le bloom.

*Figure 29 : Rendu avec le bloom (20 passes, résolution du canvas * 0.3).*

On peut également comparer les performances avec un “bloom” sur tout le canvas :

*Figure 30 : Rendu avec le bloom (100 passes, résolution du canvas * 1.0).*

Pour le rendu avec le “bloom” ayant une texture de la même taille que le canvas, on obtient une moyenne de 42 fps. Cependant, pour le “bloom” ayant une résolution plus faible, on atteint la limite de fps de l’écran (60 ou 144), alors qu’il n’y a visuellement pas de différence.

4.3 Carte d'ombrages

Si les lumières dans une scène éclairent les objets, elles peuvent aussi indirectement les assombrir. Ainsi, la carte d'ombrage, ou "shadow mapping", est une technique permettant, pour un fragment donné, de savoir s'il est à l'ombre ou non. En d'autres termes, on cherche à calculer s'il est visible de la lumière ou non. Les ombres dans une scène permettent une meilleure compréhension de sa composition, de la position de ses objets et de leurs reliefs, et ajoutent également du réalisme.

4.3.1 Explication

La carte d'ombrage est en vérité une carte de profondeur. On se positionne du point de vue de la lumière et on détermine la distance entre les points des objets et la lumière. Si le point observé est le plus proche de la lumière, alors il est éclairé. Mais s'il ne possède pas la distance la plus courte, c'est qu'il y a un autre point plus proche entre lui et la lumière. Il est donc à l'ombre.

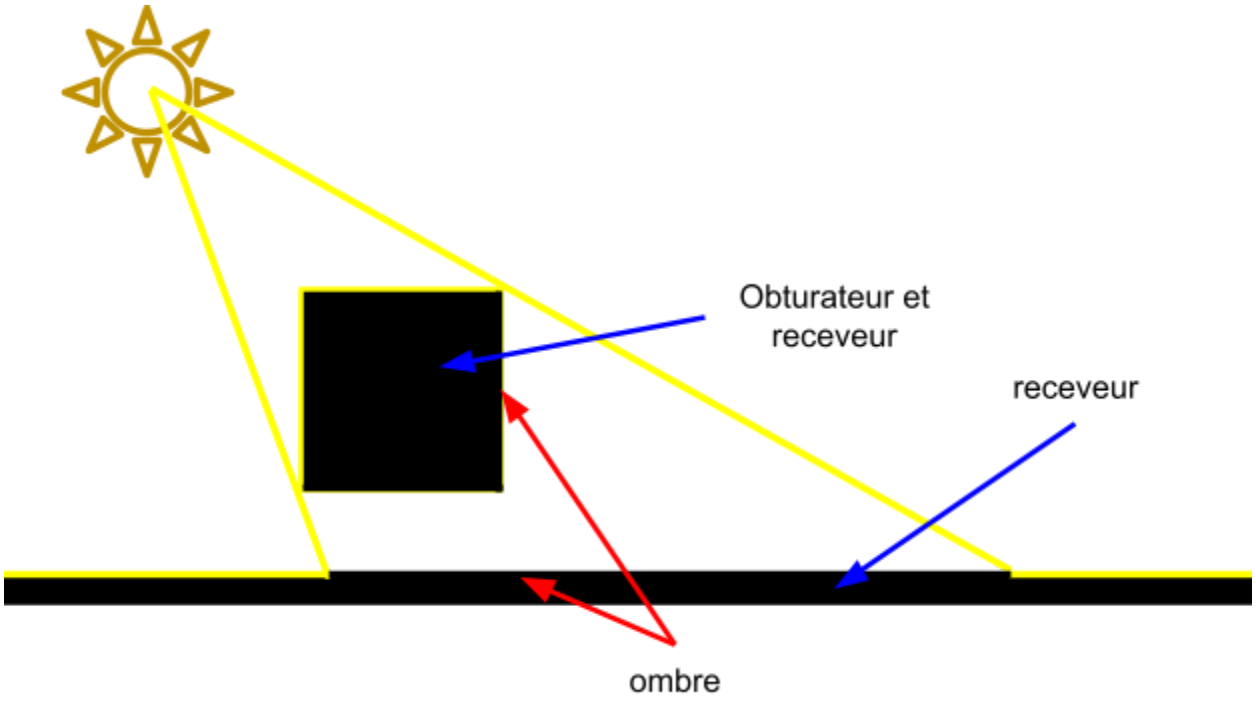

Figure 31 : Schéma explicatif des zones d'ombres.

On peut directement chercher les zones étant éclairées en faisant un rendu de la scène du point de vue de la lumière. En effet, le fait de faire un rendu avec les tests de profondeurs activés nous permet de récupérer les fragments les plus proches du point de rendu. Ensuite, il faut calculer, à partir des positions dans l'espace monde et de la carte de profondeur, un facteur d'ombrage. Pour cela, on calcule la distance du fragment à la lumière, et on la compare à la plus courte distance, donc à la valeur donnée par la carte de profondeur.

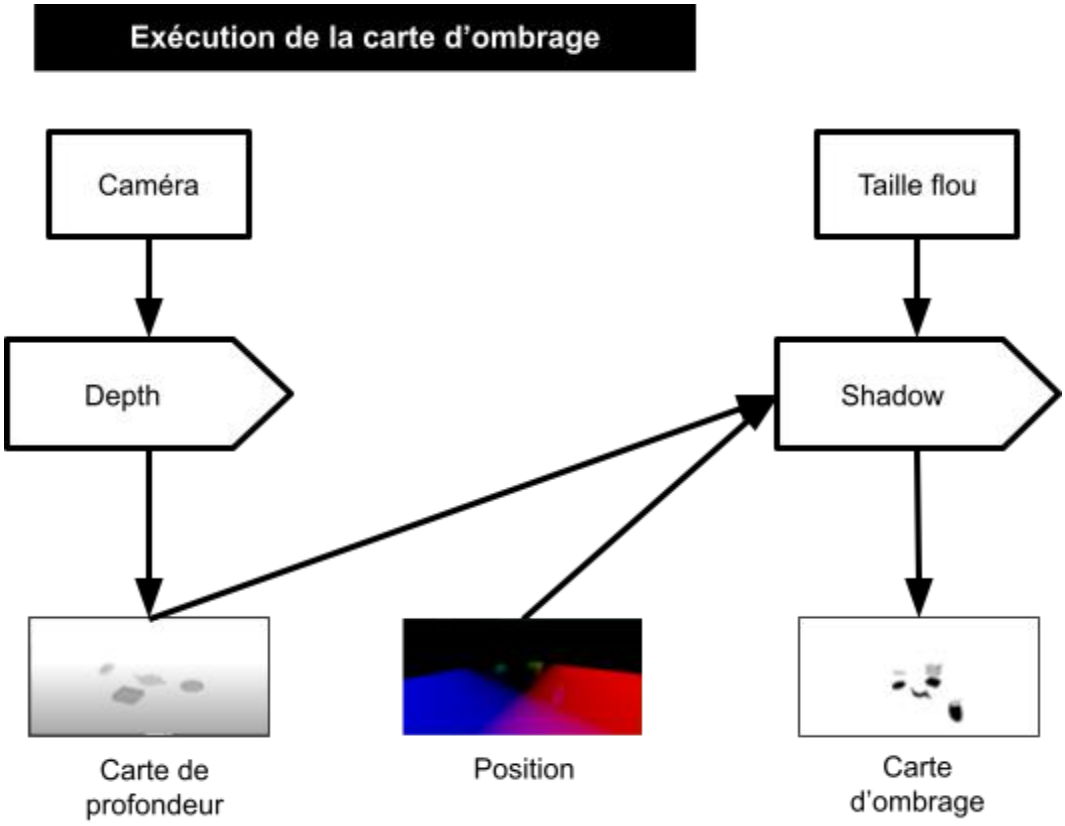

Figure 32 : Visualisation de la passe de carte d'ombrage.

On peut maintenant savoir si un fragment de l'écran est oui ou non à l'ombre de la lumière. Il peut par exemple être utilisé par le shader "Blinn Phong**"** : lorsque l'objet est dans l'ombre, il ne reçoit que la composante ambiante (couleur de base de l'objet).

4.3.2 Implémentation

On utilise une implémentation proche de celle de LearnOpenGL. Premièrement, on crée la carte de profondeur. Pour cela, il faut rendre les modèles de la scène.

Vertex shader :

```
#version 300 es
precision highp float;

layout (location=0) in vec3 aVertexPosition;
layout (location=1) in vec3 aVertexNormal;
layout (location=2) in vec2 aVertexUV;

uniform mat4 uModelMatrix;
```

```
uniform mat4 uViewMatrix;
uniform mat4 uProjectionMatrix;

void main(void) {
    gl_Position = uProjectionMatrix * uViewMatrix * uModelMatrix *
vec4(aVertexPosition, 1.0);
}
```

Quand on va faire le rendu d'une image via le pipeline de rendu WebGL, il va nous générer jusqu'à deux informations importantes : la projection des fragments composants un objet de la scène dans l'image, et la profondeur de ces derniers (notion de troisième dimension).
Pour les fragments, on doit mettre cette valeur de profondeur pour notre traitement. Cependant, comme on passe par un framebuffer, la profondeur est déjà stockée, on a donc rien de plus à faire qui n'a pas déjà été fait précédemment.

Fragment shader :

```
#version 300 es
precision highp float;

void main()
{
    // gl_FragDepth = gl_FragCoord.z;
}
```

En revanche, le calcul des ombres va lui se faire dans l'espace écran, et non dans l'espace de la scène.

Fragment shader (calcul des ombres) :

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D depthMap;

uniform mat4 uDepthViewMatrix;
uniform mat4 uDepthProjectionMatrix;

uniform float uBias;
```

```
void main()
{
    vec3 FragPos = texture(gPosition, TexCoords).rgb;

    vec4 fragPosLightSpace = uDepthProjectionMatrix * uDepthViewMatrix *
vec4(FragPos, 1.0);

    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;

    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;

    // get closest depth value from light's perspective (using [0,1] range
fragPosLight as coords)
    float closestDepth = texture(depthMap, projCoords.xy).r;

    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;

    // check whether current frag pos is in shadow
    float shadow = currentDepth - uBias > closestDepth  ? 1.0 : 0.0;
    shadow = projCoords.z >= 1.0 ? 0.0 : shadow;

    FragColor = vec4(vec3(1.0 - shadow), 1.0);
}
```

Comme la carte de profondeur est stockée dans une texture, elle est composée de pixels. Cela pose quelques problèmes lorsque la source de la carte de profondeur est presque à l'horizon d'une surface. En effet, il n'y a pas beaucoup de pixels pour échantillonner la face et on peut se retrouver avec des artefacts de rendu importants. Cela induit donc des valeurs fausses qui mettent à l'ombre des zones qui ne devraient pas l'être. On doit donc prendre en compte un biais lors de la comparaison.
De plus, les contours des ombres rendent visibles les pixels qui composent la carte de profondeur. On peut donc mettre en place une deuxième version qui choisit un biais automatiquement et fait une interpolation des valeurs d'ombrages :

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;
```

```
uniform sampler2D gPosition;
uniform sampler2D depthMap;
uniform sampler2D gNormal;

uniform mat4 uDepthViewMatrix;
uniform mat4 uDepthProjectionMatrix;

uniform vec3 uLightDir;
uniform float uBias;
void main()
{
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal  = texture(gNormal, TexCoords).rgb;

    vec4 fragPosLightSpace = uDepthProjectionMatrix * uDepthViewMatrix *
vec4(FragPos, 1.0);

    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;

    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;

    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;

    // check whether current frag pos is in shadow
    float bias = max(uBias * (1.0 - dot(Normal, uLightDir)), uBias * 0.1);

    float shadow = 0.0;
    vec2 texelSize = vec2(1.0) / vec2(textureSize(depthMap, 0));
    for(int x = -1; x <= 1; ++x) {
        for(int y = -1; y <= 1; ++y) {
            float pcfDepth = texture(depthMap, projCoords.xy + vec2(x, y) *
texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;
    shadow = projCoords.z >= 1.0 ? 0.0 : shadow;

    FragColor = vec4(vec3(1.0 - shadow), 1.0);
}
```

Enfin, pour avoir des ombres plus réalistes, on souhaiterait réaliser plusieurs cartes d'ombres, idéalement une pour chaque lumière. Cependant, dans les programmes GLSL**,** on ne peut pas réaliser de boucle sur les textures. Cependant, même si on ne peut pas faire un rendu avec plusieurs textures, on peut faire plusieurs rendus avec une même texture. Pour chaque lumière de la scène, on effectue des passes pour avoir la carte d'ombrage, on calcule ensuite la lumière, par exemple avec la méthode de "Blinn Phong", et on ajoute le résultat de ce calcul de la lumière précédente.

4.3.3 Résultat

Figure 33 : Rendu sans les ombres.

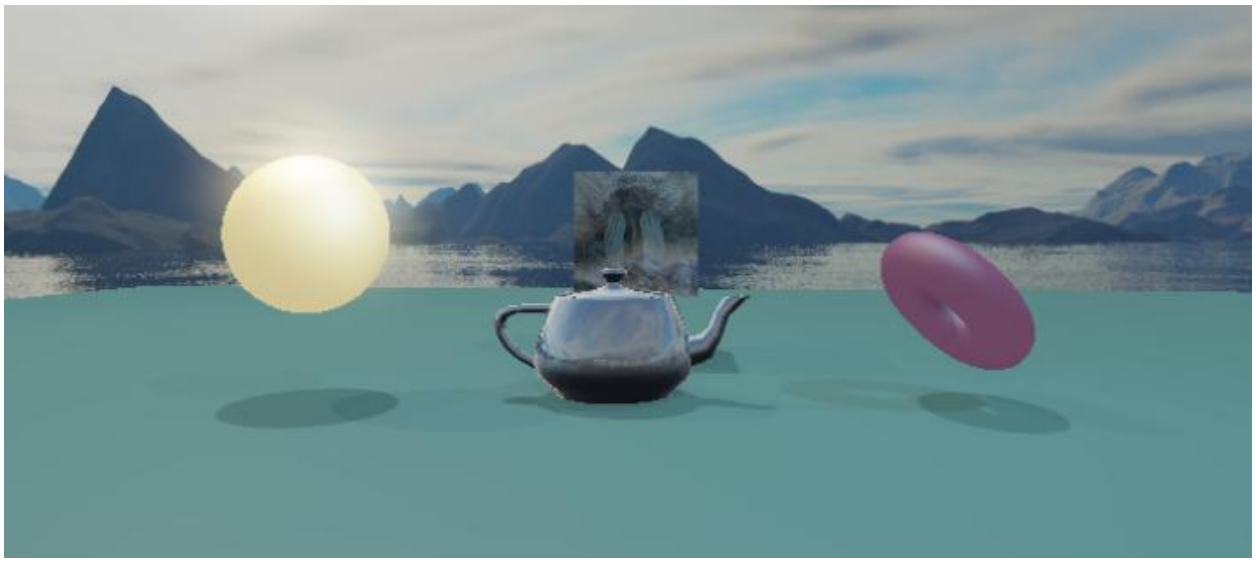

Figure 34 : Rendu avec les ombres.

4.4 Occlusion ambiante : espace écran

Pour que l'on puisse effectuer des calculs de lumières en temps réel, on ne calcule véritablement que l'éclairage direct. C'est-à-dire qu'on ne prend en compte que les rayons renvoyés par les objets sous la lumière (réflexion), mais qu'on ne prend pas en compte les rebonds entre les objets. On peut faire une approximation de ces rayons sous la composante ambiante du calcul de lumière. Cette approximation présente plusieurs défauts, comme par exemple, il n'est pas possible de modéliser l'illumination globale (un objet n'éclaire pas son environnement.). Ou alors, lorsque deux surfaces sont suffisamment proches, la lumière a des difficultés à tout éclairer, on aperçoit donc une zone plus sombre se dessiner naturellement, c'est le principe d'occlusion ambiante.

L'occlusion ambiante consiste donc à faire une meilleure approximation d'un éclairage global en assombrissant les zones où plusieurs surfaces se côtoient.

4.4.1 Explication

L'occlusion ambiante en espace écran (ou "Screen Space Ambient Occlusion : SSAO") va calculer la valeur d'occlusion dans l'espace écran, en utilisant notamment les textures de positions et de normales calculées auparavant.

Le principe de l'occlusion ambiante est d'assombrir les fragments de géométrie qui sont proches de l'intersection de plusieurs surfaces. Pour ce faire, on va placer des points d'échantillonnage autour du fragment courant (dans une demi-sphère orientée sur la surface) et on va venir déterminer si ces derniers sont à l'intérieur ou à l'extérieur de l'objet. L'occlusion ambiante est alors la proportion d'échantillons à l'intérieur de l'objet sur le nombre total d'échantillons.

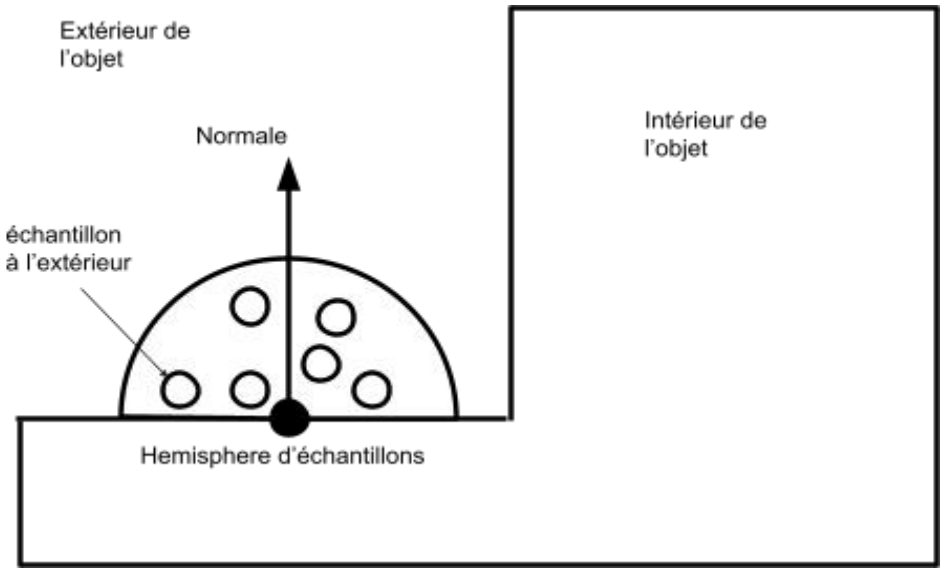

Figure 35 : Visualisation de la technique d'occlusion ambiante lorsque il n'y a pas d'occlusion.

Quand t'il n'y a pas d'occlusion, on ne diminue tout simplement pas l'intensité ambiante.

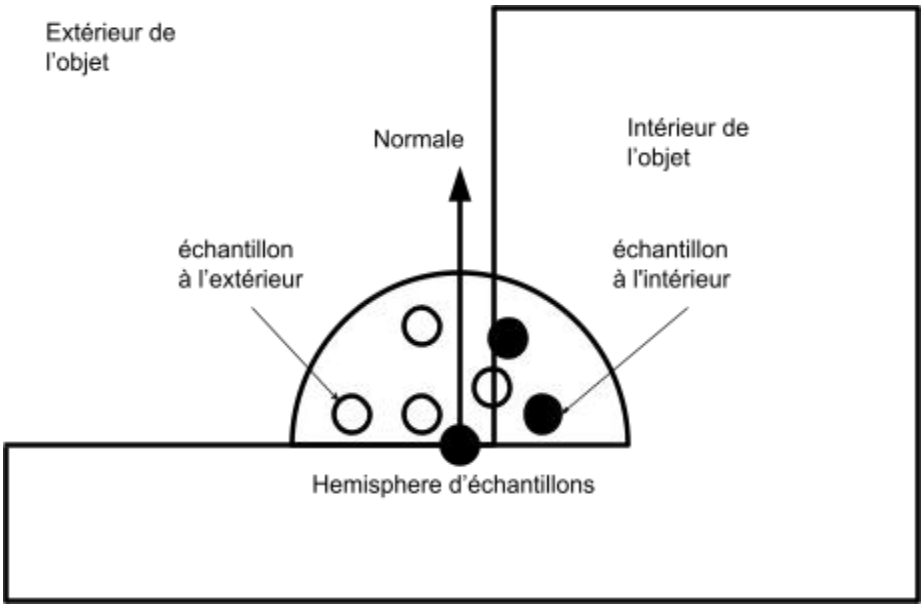

Figure 36 : Visualisation de la technique d'occlusion ambiante lorsque il y a une occlusion.

Quand il y a une occlusion, l'intensité ambiante est diminuée de 2/6 (car deux points sont à l'intérieur sur les six points de l'exemple).

Pour réaliser une technique de SSAO, on retrouve donc trois paramètres principaux. En premier lieu, il y a le nombre d'échantillons, impactant directement la qualité du résultat, mais aussi les performances. Ensuite, on trouve le rayon de l'hémisphère. Celui-ci va permettre de modifier la distance à une surface avant qu'il ne commence à s'assombrir (distance limite de détection de surface proche). Enfin, on peut ajuster la puissance d'occlusion, qui dicte au traitement à quel point l'occlusion est importante.

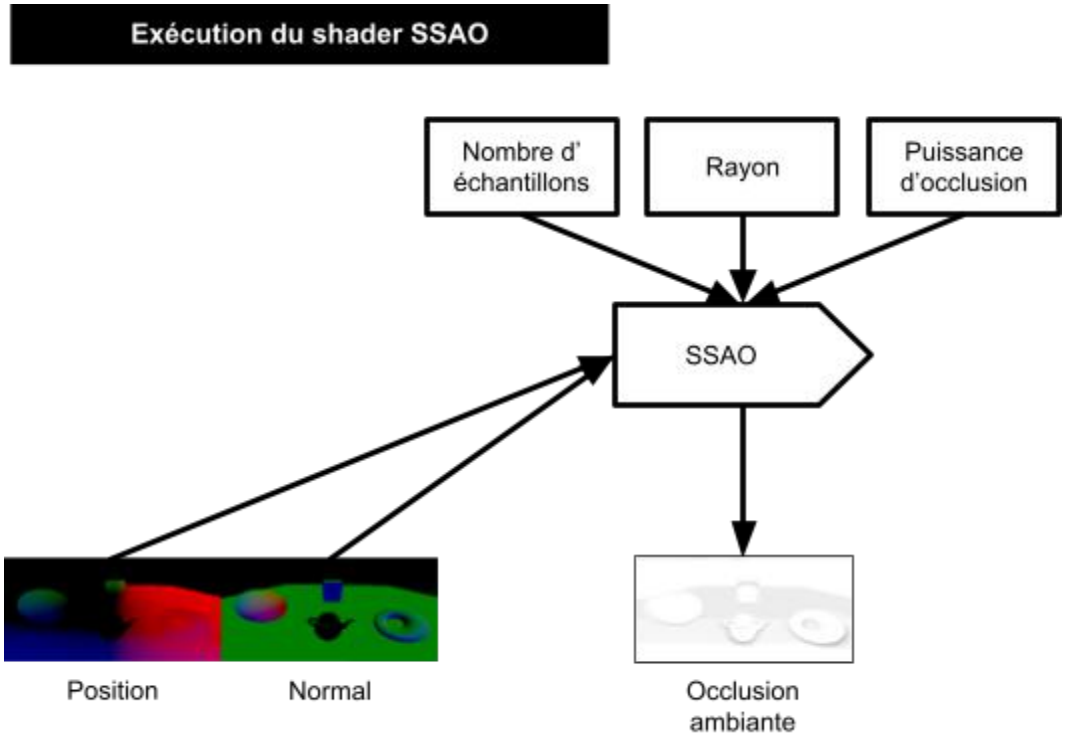

Figure 37 : Visualisation de la passe d'occlusion ambiante

4.4.2 Implémentation

Pour l'implémentation, nous sommes passés par une technique proche de celle expliquée dans l'article de LearnOpenGL traitant de ce sujet, et nous avons corrigé quelques aberrations, grâce à notamment un article de Philip Fortier (voir les liens en annexe).

Concernant le cœur de l'implémentation, celle-ci se fait en plusieurs étapes. On doit tout d'abord générer des points d'échantillonnages aléatoires. Pour ce faire, on génère également une texture aléatoire afin d'introduire une rotation sur l'hémisphère. Une fois ces deux données générées, on passe le traitement au shader correspondant. Ce shader est dans l'espace écran, on utilise donc le même vertex shader.

On fait cependant différents calculs dans le fragment shader. Premièrement, on récupère les coordonnées monde que l'on convertit en coordonnées vues. Cela est utile pour récupérer la profondeur par exemple. Ensuite, on détermine la matrice de l'espace tangent, afin de pouvoir facilement poser l'hémisphère sur la surface. Enfin, on parcourt les échantillons en comptant le nombre d'entre eux qui sont sous la surface.

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D texNoise;

uniform vec3 samples[128];

uniform int kernelSize;
uniform float radius;
uniform float depthBias;
uniform float angleBias;
uniform float uNoiseScale;
uniform float occlusionPower;

uniform mat4 uUsedViewMatrix;
uniform mat4 uUsedProjectionMatrix;

float lerp(float a, float b, float x) {
    return a + x * (b - a);
}

void main()
{
    // get input for SSAO algorithm
    vec3 WorldFragPos = texture(gPosition,   TexCoords).rgb;
    vec3 FragPos   = (uUsedViewMatrix * vec4(WorldFragPos, 1.0)).rgb;
    vec3 Normal    = texture(gNormal, TexCoords).rgb;

    vec2 noiseScale = vec2(textureSize(gPosition, 0)) * uNoiseScale;
    vec3 randomVec = normalize(texture(texNoise, TexCoords * noiseScale).xyz);

    // create TBN change-of-basis matrix: from tangent-space to view-space
    vec3 tangent = normalize(randomVec - Normal * dot(randomVec, Normal));
    vec3 bitangent = cross(Normal, tangent);
    mat3 TBN = mat3(tangent, bitangent, Normal);

    // iterate over the sample kernel and calculate occlusion factor
    float occlusion = 0.0;
```

```
    vec3 currentSample;
    float invKernelSize = 1.0 / float(kernelSize);
    for(int i = 0; i < kernelSize; ++i) {
        if (dot(normalize(samples[i]), Normal) > angleBias) {

            // scale samples so that they're more aligned to center of kernel
            currentSample = samples[i];
            currentSample *= lerp(0.1, 1.0, float(i) * invKernelSize);

            // get sample position
            vec3 samplePos = TBN * currentSample; // from tangent to view-space
            samplePos = FragPos + samplePos * radius;

            // project sample position (to sample texture) (to get position on
screen/texture)
            vec4 offset = vec4(samplePos, 1.0);
            offset = uUsedProjectionMatrix * offset; // from view to clip-space
            offset.xyz /= offset.w; // perspective divide
            offset.xyz = offset.xyz * 0.5 + 0.5; // transform to range 0.0 -
1.0

            // get sample depth
            float sampleDepth = (uUsedViewMatrix * vec4(texture(gPosition,
offset.xy).rgb, 1.0)).z; // get depth value of kernel sample

            // range check & accumulate
            float rangeCheck = smoothstep(0.0, 1.0, radius / abs(FragPos.z -
sampleDepth));
            occlusion += (sampleDepth >= samplePos.z + depthBias ? 1.0 : 0.0) *
rangeCheck;
        }
    }
    occlusion = 1.0 - (occlusion / float(kernelSize));

    FragColor = vec4(vec3(pow(occlusion, occlusionPower)), 1.0);
}
```

Pour appliquer cette occlusion, il faut passer au calcul de lumière la texture résultante. L'occlusion agit alors comme un facteur sur la luminosité ambiante.

4.4.3 Résultat

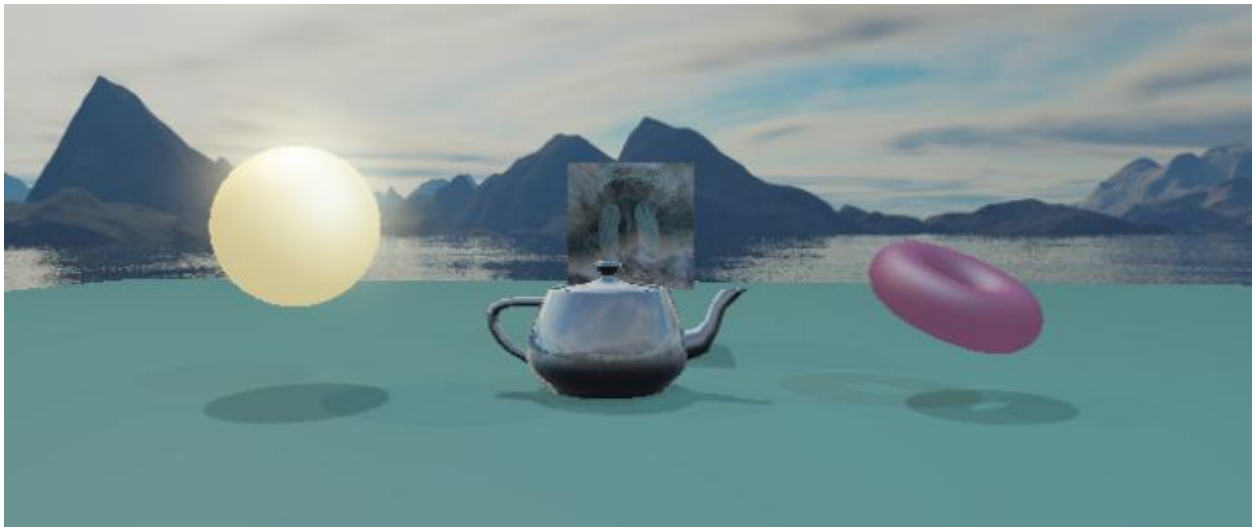

Figure 38 : Rendu sans occlusion ambiante.

Figure 39 : Rendu avec occlusion ambiante (puissance de 1.5).

Figure 40 : Rendu avec occlusion ambiante (puissance de 10).

Cet effet n'est pas très visible, mais ajoute tout de même une présence aux objets. Cela permet de mieux en déterminer leur contour notamment.

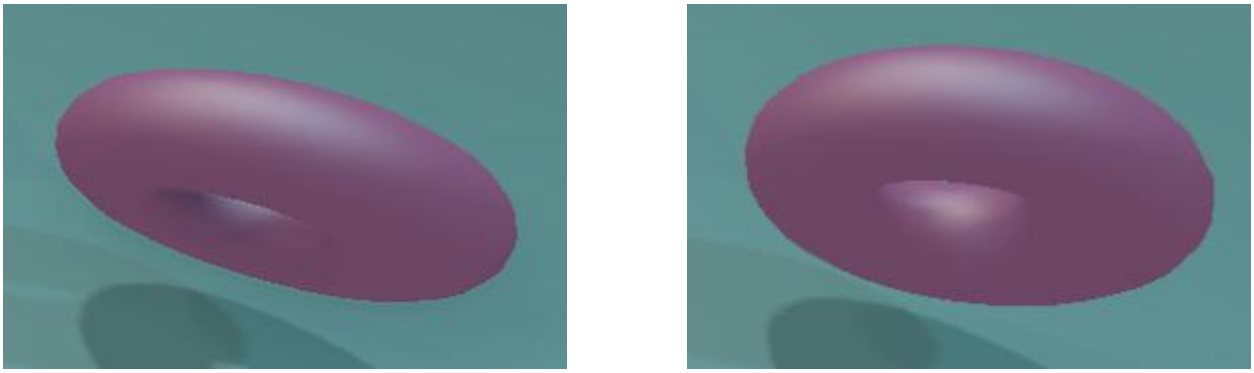

Figure 41 : Torus avec et sans occlusion ambiante.

Sur cet exemple d'images là, on remarque plus facilement le trou du torus grâce à l'occlusion ambiante.

4.5 Calcul de lumières Blinn Phong

Afin de prendre en compte les lumières dans le rendu de la scène, on fait ce que l'on appelle une passe de lumière. Cette passe va, en plus des textures calculées précédemment, utiliser les lumières de la scène pour se rendre.

Blinn-Phong est un moyen rapide et simple de faire cette passe, mais il est plus esthétique que physiquement correct.

4.5.1 Explication

Le calcul de lumière via Blinn Phong se décompose en trois parties. Premièrement, on calcule une couleur ambiante. Cela correspond à une approximation des rebonds des lumières dans la scène. Par exemple, si la scène représente un paysage au soleil, alors on a une forte composante ambiante. Cependant, si on représente une scène à l'abri du soleil, comme dans un tunnel ou pendant la nuit, alors la composante ambiante est faible. Elle se calcule en multipliant la couleur de l'objet à un facteur de luminosité ambiante. Ensuite, on calcule pour chaque lumière sa composante diffuse. Cela correspond au pourcentage de lumière réfléchie dans toutes les directions depuis un point donné. Moins formellement, il s'agit, pour le calcul de cette composante, de tenir notamment compte de l'inclinaison avec laquelle la lumière arrive sur la surface que l'on calcule. Enfin, la troisième composante entrant en compte est la composante de spécularité. Elle va renseigner la proportion de lumière réfléchie depuis la surface éclairée et dans la direction de réflexion d'inclinaison de celle-ci (direction de réflexion géométrique). Cela permet de simuler la rugosité du modèle.

Ce modèle de calcul va permettre de déterminer l'apport de la lumière à un objet de la scène localement à celui-ci, et nous fournit une règle de calcul de luminosité dépendant de paramètres tels que l'orientation de la face à prendre en compte, ou la position de la lumière.

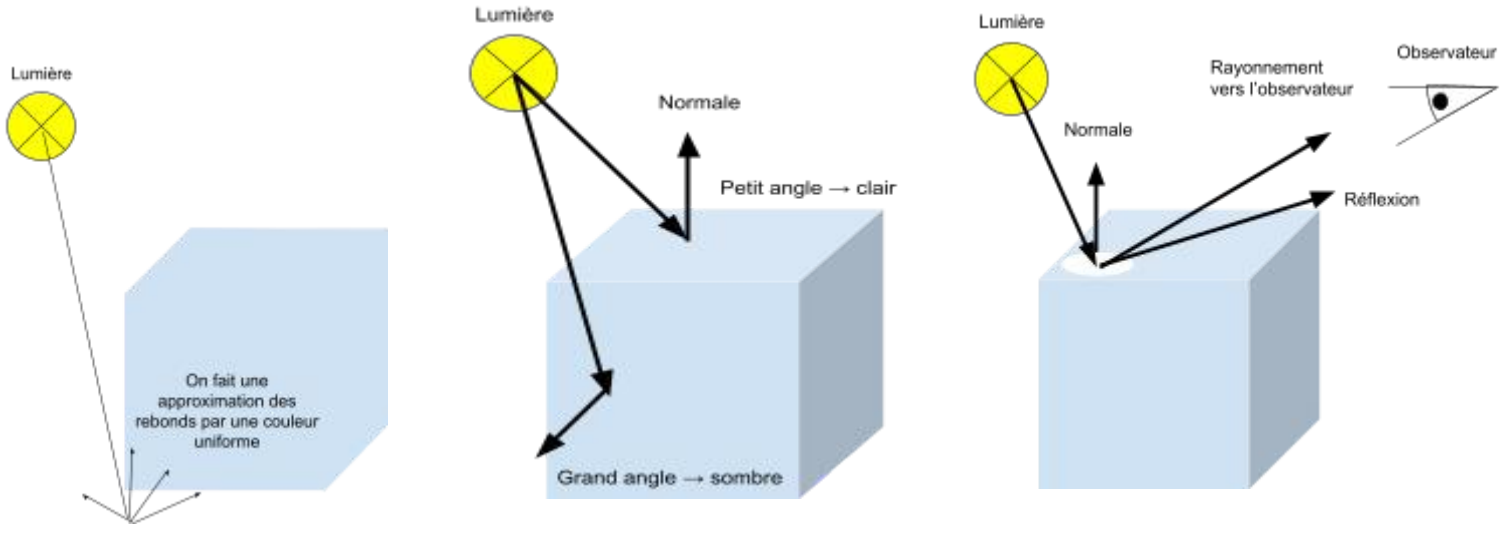

Figure 42 : Composantes ambiante(gauche), diffuse (milieu) et spéculaire (droite).

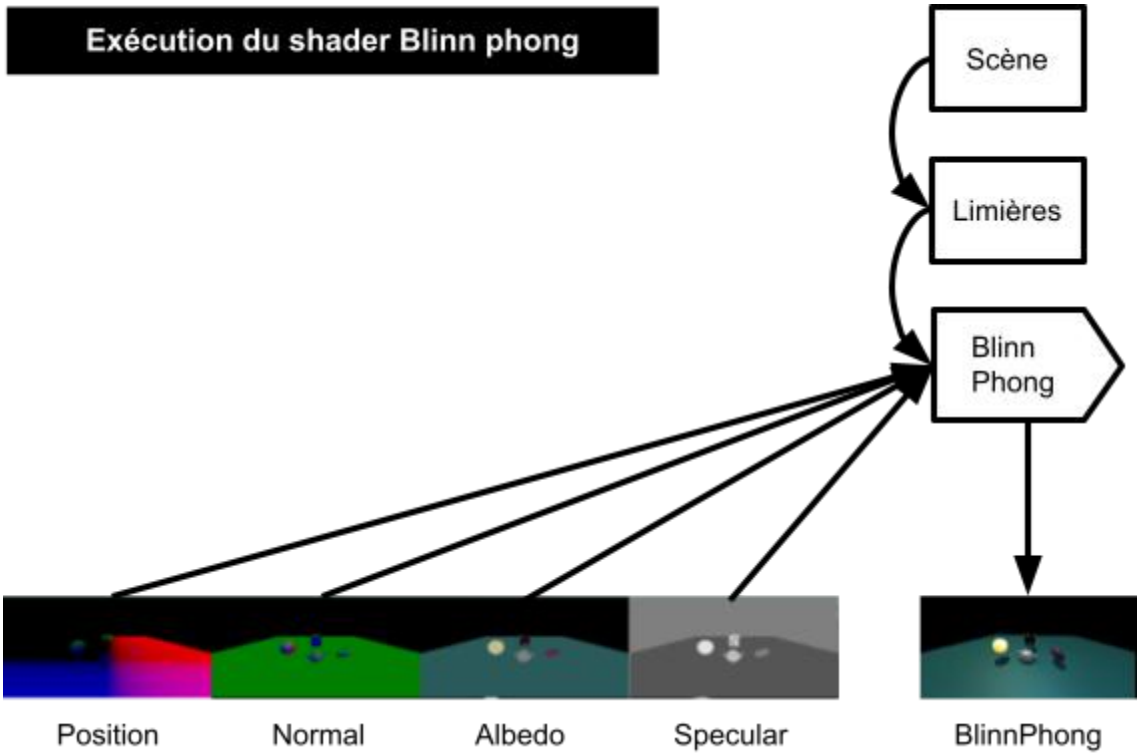

Figure 43 : Visualisation de la passe de Blinn Phong

Blinn Phong n'est pas la seule passe lumineuse possible. Il y a aussi les passes d'illumination de Gouraud ou d'ombrage plat (style low poly).

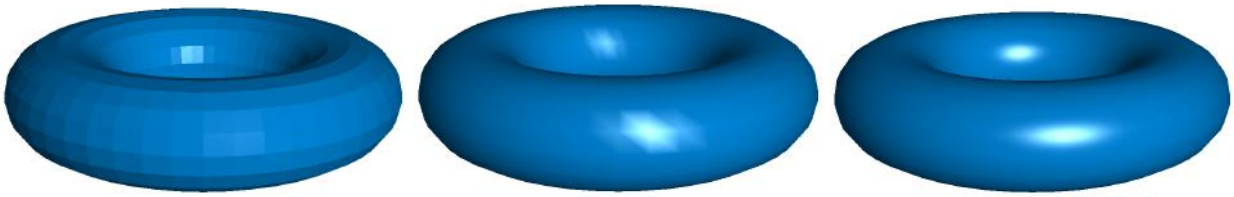

Figure 44 : Flat shading (gauche) vs Gouraud (milieu) vs Blinn Phong (droite).

4.5.2 Implémentation

Ce shader utilise les textures calculées lors de la passe géométrique en faisant un rendu sur un quad. Il utilise également la liste des lumières de la scène. Dans le fragment shader, on va donc utiliser ces informations. On calcule tout d'abord la composante ambiante, puis on ajoute pour chaque lumière les composantes diffuses et spéculaires.

Fragment shader :

```
#version 300 es
precision highp float;

layout (location = 0) out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D gPosition;
uniform sampler2D gNormal;
uniform sampler2D gAlbedoSpec;

struct Light {
    vec3 Position;
    vec3 Color;

    float Linear;
    float Quadratic;
};
const int NR_LIGHTS = 16;
uniform Light uLights[NR_LIGHTS];

uniform int uNLights;
uniform vec3 uViewPos;
uniform float uAmbiant;

void main()
{
    vec3 FragPos = texture(gPosition, TexCoords).rgb;
    vec3 Normal = texture(gNormal, TexCoords).rgb;
    vec3 Diffuse = texture(gAlbedoSpec, TexCoords).rgb;
    float Specular = texture(gAlbedoSpec, TexCoords).a;

    // ambient
    vec3 lighting = Diffuse * uAmbiant;

    vec3 viewDir  = normalize(uViewPos - FragPos);
    int loop = min(uNLights, NR_LIGHTS);
    for(int i = 0; i < loop; ++i)
    {
        // diffuse
        vec3 lightDir = normalize(uLights[i].Position - FragPos);
        vec3 diffuse = max(dot(Normal, lightDir), 0.0) * Diffuse *
uLights[i].Color;
```

```
        // specular
        vec3 halfwayDir = normalize(lightDir + viewDir);
        float spec = pow(max(dot(Normal, halfwayDir), 0.0), 16.0);
        vec3 specular = uLights[i].Color * spec * Specular;
        // attenuation
        float distance = length(uLights[i].Position - FragPos);
        float attenuation = 1.0 / (1.0 + uLights[i].Linear * distance +
uLights[i].Quadratic * distance * distance);
        diffuse *= attenuation;
        specular *= attenuation;
        lighting += diffuse + specular;
    }
    FragColor = vec4(lighting, 1.0);
}
```

4.5.3 Résultat

On modifie la composante ambiante globale, et les composantes diffuses et spéculaires du torus.

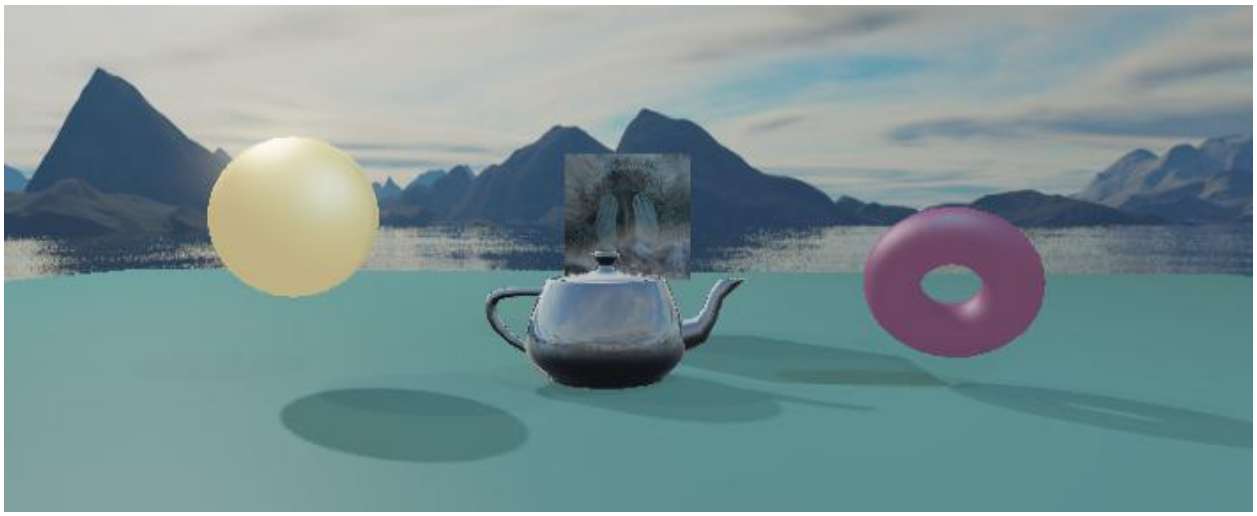

Figure 45 : Blinn Phong (ambient = 0.3, diffuse = [.7, .4, .6], spéculaire = 1)

Figure 46 : Blinn Phong (ambient = 0.8, diffuse = [.7, .4, .6], spéculaire = 1)

Figure 47 : Blinn Phong (ambient = 0.3, diffuse = [.7, .6, .4], spéculaire = 1)

Figure 48 : Blinn Phong (ambient = 0.3, diffuse = [.7, .4, .6], spéculaire = 0)

Cette technique est utile pour implémenter faire des approximations simples et rapides de l'éclairage, mais elle n'est pas physiquement correcte. Si l'on souhaite avoir un rendu plus réaliste et poussé, alors il faut implémenter d'autres techniques, dont les techniques PBR.

4.6 Rendu physique réaliste : PBR (Physically based rendering)

Pour résumer simplement ce principe, il s'agit d'une collection de techniques de rendu se basant plus ou moins sur la même théorie, tentant de reproduire la physique du monde réel. Celles-ci ont toutes pour objectif principal d'imiter les comportements de la lumière physiquement parlant, plus réaliste que Phong ou Blinn-Phong par exemple.
Cet ensemble de techniques nous permet de définir des éléments de part de vraies propriétés physiques matérielles, comme la rugosité ou encore la capacité d'une surface ou d'un matériau à conduire l'énergie (lumineuse dans notre cas).
Cependant, comme tout ce qui a été discuté jusqu'à présent, cela reste une approximation du comportement de la lumière, voilà pourquoi on parle de rendu "basé" physique, et pas de vrai rendu physique.

On peut distinguer trois grandes caractéristiques essentielles dans un modèle d'éclairage basé sur la physique :

- Doit être basé sur un modèle de microfacette.
- Doit respecter le principe de conservation d'énergie.
- Doit s'appuyer sur une BRDF physique.

4.6.1 Microfacettes

Cette première caractéristique part du principe physique qu'à l'échelle microscopique, la surface d'un objet ne sera jamais complètement lisse.

Cette rugosité de la surface ne la définit plus comme un unique miroir réflecteur de lumière, mais comme la composition de pleins de petits miroirs appelés des micro facettes. Ainsi, plus une surface est rugueuse, et plus ces miroirs seront alignés de manière aléatoire. Cela va avoir pour effet direct de faire réfléchir la lumière dans toutes les directions.

Figure 49 : Schéma explicatif du rôle des microfacettes.

On peut approximer statistiquement le calcul des microfacettes par un paramètre de répartition de celles-ci, le paramètre matériel de rugosité (entre 0 et 1).

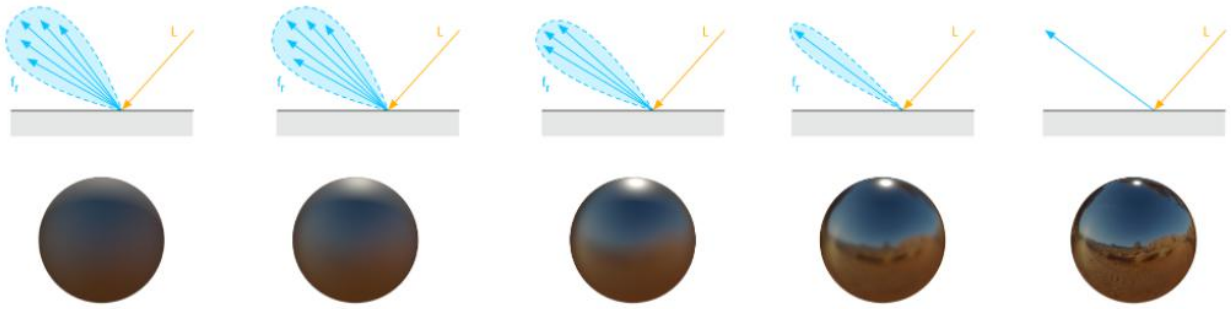

Figure 50 : Exemple de rugosité et de lobe spéculaire.

On remarque que sur l'image la plus à gauche, qui a un coefficient de rugosité maximum, la réflexion spéculaire est imperceptible.

4.6.2 Conservation d'énergie

Ce deuxième principe se base sur le fait physique que l'énergie au départ d'une transmission lumineuse ne doit jamais être moins forte que l'énergie lumineuse sortante de cette transmission, sauf cas particulier des surfaces émissives. Si on reprend l'image précédente, ce phénomène s'observe par le fait que plus la surface est rugueuse, et plus la zone de réflexion spéculaire sera étendue, et plus l'intensité de réflexion sera moindre. Si l'intensité restait constante, on aurait une surface de réflexion spéculaire constante très forte. On aurait alors plus d'énergie lumineuse transmise qu'acquise par l'objet.

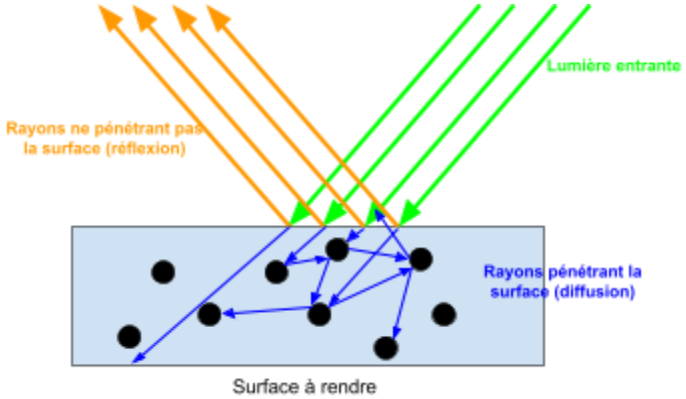

Figure 51 : Schéma du principe de conservation énergétique.

Ce schéma montre ce principe pour les composantes diffuses et spéculaires : quand un rayon lumineux va entrer en contact avec la surface d'un objet, celui-ci va se séparer en deux rayons distincts, le rayon de réfraction et le rayon de réflexion. La partie réfractée de ce rayon va venir pénétrer la surface et composer son aspect diffus, tandis que la partie réfléchie ne va pas pénétrer dans la surface, et directement être renvoyée, formant ainsi l'aspect spéculaire.

Selon ce principe de conservation énergétique, on peut calculer les coefficients diffus et spéculaires de la manière suivante :

$$Ks = specularComponent()$$
$$Kd = 1 - Ks$$

Pour les surfaces métalliques en PBR (non-conductrices de lumière), toute la lumière réfractée est directement absorbée sans être diffusée. Ces surfaces ne produisent que de la lumière réfléchie (ou spéculaire).

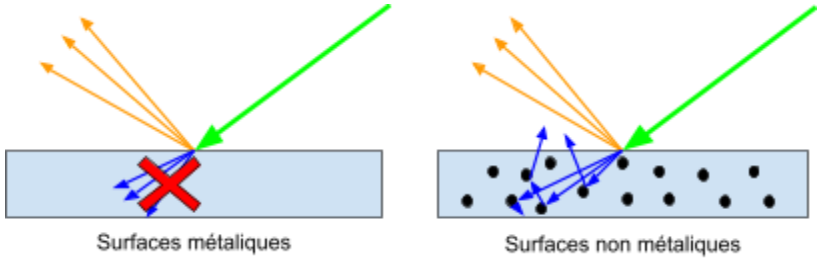

Figure 52 : Surfaces métalliques et non métalliques.

4.6.3 Illumination locale et comportement à la lumière

Équation du rendu

Tout modèle de calcul du comportement de la lumière sur des surfaces en 3D se voulant réaliste se base sur une équation du rendu. Elle peut être définie en tout point de la scène à rendre comme la radiance quittant ce point.

Moins formellement, il s'agit de la description du modèle mathématique que l'on va appliquer afin de venir calculer en tout point de la surface d'un objet, son comportement local avec les sources de lumière qui l'éclairent le long d'un hémisphère centré le plus souvent sur la normale à la surface (Ω). Plus l'approximation par des calculs mathématiques sera fine, et plus les résultats que l'on obtiendra seront des résultats proches de rendus réalistes. Il est important de noter que quand on parle de rendus 3D, il s'agit nécessairement d'approximations d'équations de rendu mathématiques, car on opère dans un monde discret informatique, et non-continu. La plupart du temps, on résout ce problème de continu à discret en échantillonnant sur une intervalle précise, créant des sources d'erreurs évidentes.

Dans les faits, une équation du rendu peut s'écrire de la manière suivante :

$$Lo(x, wo, \lambda, t) = Le(x, wo, \lambda, t) + \int_{\Omega} fr(x, wi, wo, \lambda, t) Li(x, wi, \lambda, t)(wi.n) dwi$$

Sous ses airs un peu complexes, cette équation du rendu est en réalité assez simple dans sa construction. Elle traduit le taux de lumière sortante (Lo) en un point donné d'une scène comme la combinaison de son taux de lumière émise (Le, lumière ambiante) et de l'apport de la lumière fourni en ce point par toutes les lumières de la scène qui viennent l'éclairer

(composante de spécularité et de diffusion). L'intégrale de l'équation se traduit numériquement comme une somme des apports des différents points de lumières dans la scène sur le point depuis lequel on cherche à opérer les calculs de lumière, ici x.

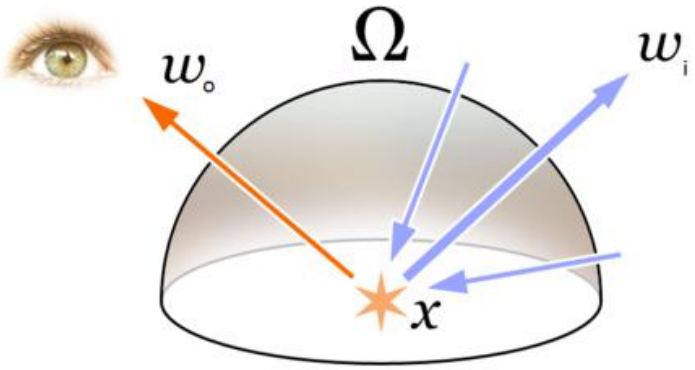

Figure 53 : Schéma de l'équation de rendu.

Cette équation de rendu est actuellement le meilleur modèle dont nous disposons pour simuler la lumière sur un point de la scène.

BRDF (Bidirectional Reflectance Distribution Function)

La fonction de distribution de la réflectance bidirectionnelle $fr(x, wi, wo, \lambda, t)$ est une fonction définissant globalement la manière dont la lumière se réfléchit sur une surface opaque. C'est son implémentation qui va permettre de définir la manière dont la lumière sera réfléchie depuis la surface sur laquelle elle rentre en contact, soit la composante spéculaire de la lumière étudiée en notre point de la scène.

Cette fonction de BRDF va venir appliquer une pondération en fonction des coefficients matériels de l'objet que l'on cherche à éclairer. En effet, le résultat final de l'éclairage d'un point d'une scène par les lumières qui le compose va s'obtenir grâce au calcul de la radiométrie de chacune des sources de lumière de la scène (ou l'apport énergétique de celle-ci), mais également de l'aspect de la surface à éclairer, qui va venir perturber les réflexions lumineuses de l'objet (apporter du chaos dans les réflexions).

4.6.4 Implémentation

Pour générer un modèle de rendu basé sur la physique, on va devoir donc approximer au mieux cette équation de rendu. Plus on va approximer fidèlement, et plus on va avoir un rendu réaliste, et plus on va perdre en performances. On peut déjà en simplifier son écriture par l'équation suivante :

$$IntensiteLumineuseSortante(x, wo, \lambda, t) = \int_{\Omega} BRDF(x, wi, wo, \lambda, t)\ radiance(x, wi, \lambda, t)\ (wi.n)\ dwi$$

Pour le choix des différentes méthodes qui composent cette équation, nous nous sommes beaucoup basés sur l'article présenté sur le site LearnOpenGL à ce sujet. Il y indique les différentes méthodes possibles, et surtout celle présentant de bons résultats de rendu.

Pour la méthode d'approximation de BRDF, on se base sur la méthode de Cook-Torrance :

$$BRDF = kd * flambert + ks * fcook - torrance$$
$$flambert = \frac{albedo}{\pi}$$
$$fcook - torrance = \frac{DFG}{4(w0.n)(wi.n)}$$

DFG fait ici référence à trois fonctions :

- D : distribution des normales (plus une surface est rugueuse, et plus elle aura ses normales de micro facettes non-alignées).
- F : équation de Fresnel décrivant le rapport de la réflexion de la surface à différents angles de la surface (plus on regardera une surface avec un angle rasant à celle-ci, et plus la lumière qui l'éclaire aura un lobe spéculaire qui nous paraîtra étendu.).
- G : fonction de géométrie décrivant la propriété d'auto-ombrage d'un objet de la scène induit par les microfacettes à la surface (plus une surface est rugueuse, et plus ses microfacettes auront tendance à être marquées et donc à s'auto-ombrager).

Pour D, on prendra l'équation GXX de Trowbridge-Reitz (avec a comme coefficient de rugosité) :

$$GXX\ Trowbridge - Reitz = \frac{a^2}{\pi((n.h)^2(a^2-1)+1)^2}$$

```
float DistributionGGX(vec3 N, vec3 H, float a)
{
    float a2     = a*a;
    float NdotH  = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

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

    return nom / denom;
}
```

D'après les explications sur le PBR fournies par Filament, il est possible d'optimiser ce calcul et de gagner en précision, car le produit scalaire n'est pas très précis pour des résultats proches d'un et des flottants.

Pour G, nous prenons en considération la GGX de Smith :

$$GGXSchlick = \frac{n.v}{(n.v)(1-k)+k}$$

```
float GeometrySchlickGGX(float NdotV, float k)
{
    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}
```

Pour F, nous prenons l'approximation de Fresnel-Schlick :

$$FSchlick = F0 + (1 - F0)(1 - (h.v))^{5}$$

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

4.6.5 Propriétés matérielles

Ainsi, pour composer un tel système de rendu physiquement réaliste, on va appliquer à nos objets une série de propriétés matérielles que nous ferons varier afin d'obtenir des effets différents, proches de vrais matériaux réels.
Une des premières propriétés matérielles définissables est la rugosité, qui va venir faire fluctuer la spécularité issue des micro facettes.

On détecte aussi l'aspect métallique ou non d'une surface. Cela peut se traduire par un simple coefficient binaire, ou une valeur flottante entre 0.0 et 1.0 pour des surfaces non parfaitement métalliques (métal poussiéreux, usé, brossé).

La valeur F0 (combien la surface reflète si on regarde directement la surface) de réflexivité à angle perpendiculaire peut être un paramètre en plus pour notre calcul du BRDF. Cela peut être un paramètre matériel supplémentaire ou être intégré à une valeur par défaut dans notre programme.

```
vec3 F0 = vec3(0.04);
F0 = mix(F0, surfaceColor.rgb, metalness);
```

Pour chacun des composants décrivant un comportement précis de notre objet à la lumière, nous allons passer par une texture que nous pondérons par un coefficient variable. Le fait de passer par des textures permet de renseigner un comportement différent pour deux points de la même surface, et ainsi d'avoir des surfaces beaucoup plus riches.

On distingue quatre grandes textures applicables :

- Albedo : détermine la couleur de la surface, ou la réflexivité de base pour les endroits métalliques. Cette texture est différente de la texture de diffuse car elle ne renseigne que la couleur de l'objet.
- Normale : carte de normale permettant de spécifier par fragments une normale unique (donner l'illusion qu'une surface est bosselée).
- Métallique : Carte renseignant si un texel est un texel métallique ou non (carte binaire ou en niveau de gris).
- Rugosité : Pareil que la carte métalique mais pour la rugosité de notre composante. Cette valeur va venir influencer les orientations statistiques des micro facettes de la surface.
- Occlusion Ambiante : Spécifie un facteur d'ombrage supplémentaire de la surface.

4.6.6 Résultats

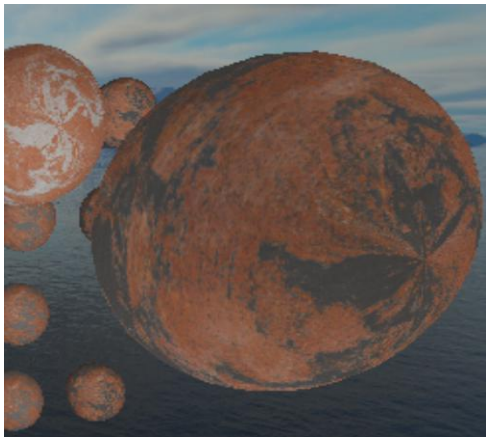

Figure 54 : Rendu PBR avec des textures pour les propriétés matérielles.

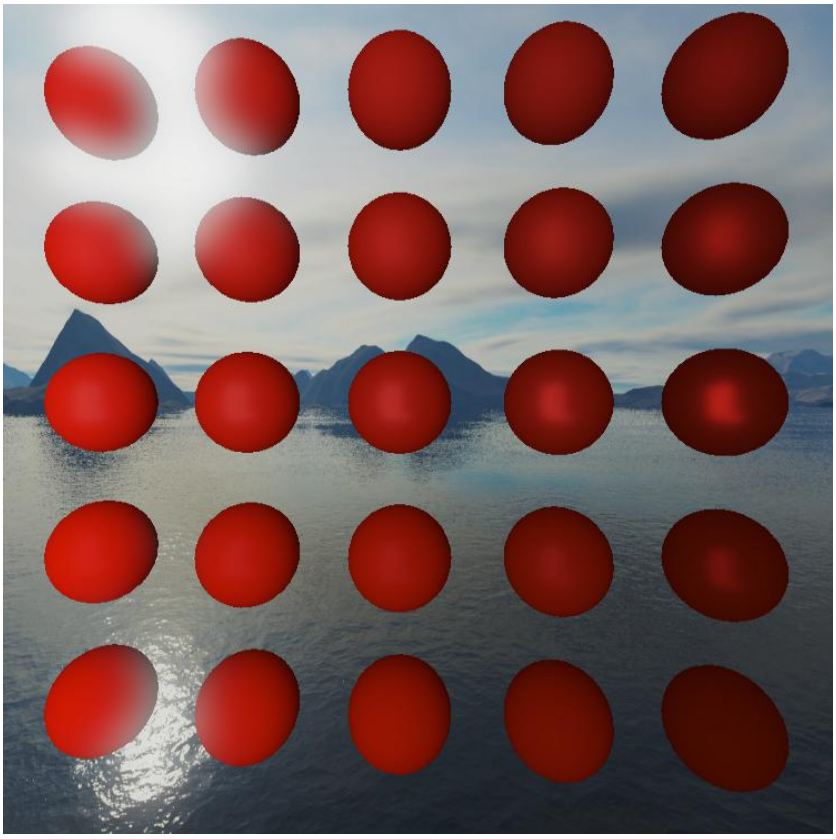

Figure 55 : Rendu PBR en grille.

Sur cette image, plus on va à droite et plus la surface devient métallique, et plus on va vers le haut, et plus la rugosité augmente.

On remarque que plus le coefficient d'aspect métallique d'un objet est grand, et plus les réflexions de la lumière seront nettes à sa surface. Plus sa surface sera lisse, et moins au contraire ces réflexions seront présentes, pour laisser place à des réflexions plus diffuses.

Quand on utilise des textures pour les coefficients de matériaux, on obtient des rendus proches de rendus de vrais objets.

.

Chapitre 5

Perspectives futures et contraintes

Il y a de nombreuses méthodes de rendu que nous aurions voulu implémenter, mais qui n'ont pas abouti par contrainte de temps. En effet, nous avons rencontré quelques problèmes qui nous ont ralentis, voire bloqués sur certains points. Certaines difficultés concernent des techniques qui en apparence paraissent simples, mais pour une raison inconnue ne fonctionnent pas sur notre système. Par exemple, si l'on met en place des "stencil buffer", les performances sont tout de suite diminuées de moitié. Cela est peut-être dû à une mauvaise implémentation de WebGl, mais nous n'avions pas eu le temps de regarder dans les détails ce qu'il pourrait en être la cause.

De plus, nous aurions voulu mettre en place des réflexions dynamiques. Ceci est possible grâce à des rendus sur une "cubemap". Cependant, même en ayant essayé de nombreuses techniques, le rendu sur la "cubemap" ne parvient pas à modifier celle-ci. Nous pouvons lire une "cubemap", mais nous ne pouvons pas y écrire. Mais avec plus de temps, nous sommes sûrs de pouvoir implémenter ces techniques-là. En revanche, il y a, au moment de faire ce travail, des limitations provenant de l'environnement WebGL, qui ne sont pas présentes en OpenGL. Comme par exemple la totale absence de "geometry shader". Ce concept n'existe pas en WebGL. Il en est de même pour les textures 3D.

Il y a également des améliorations que l'on peut apporter sur ce projet. Sur l'interface graphique, on peut concevoir un menu pour sélectionner et modifier un modèle de la scène. On peut également améliorer le rendu en apportant un calcul de lumière plus réaliste. On peut mettre en place un autre type de lumière. On ne supporte pour l'instant que les lumières sur un point. Et aussi mettre en place des ombres douces, prenant parti d'un nouveau type de lumière étendue. On peut par exemple essayer d'autres techniques de PBR, ou implémenter le "subsurface scattering".

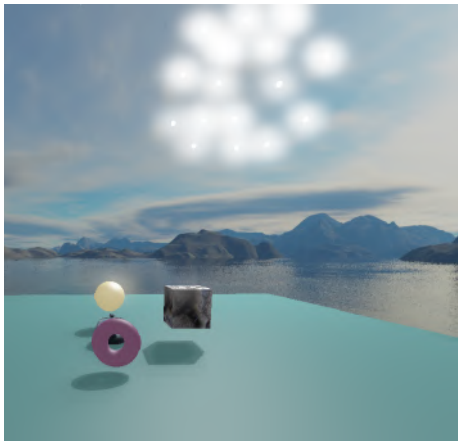

Figure 56 : Manipulation de la scène pour faire des ombres douces.

Concernant le PBR, la lumière réfractée n'est jamais complètement transformée en lumière diffuse. En effet, si on reprend la modélisation de la lumière comme un faisceau, au contact à la surface, une partie va être directement réfléchie (spéculaire), une partie va pénétrer la surface et ressortir (diffusion) et une partie va se perdre en chaleur. Il faut donc modéliser ce phénomène de perte d'énergie lumineuse en énergie thermique.

Les techniques de shaders spécifiques qui tiennent compte de ce phénomène sont connues sous le nom de techniques de diffusion sous la surface et sont employées lorsqu'il s'agit de rendre fidèlement la peau humaine, le marbre ou encore la cire de bougie.

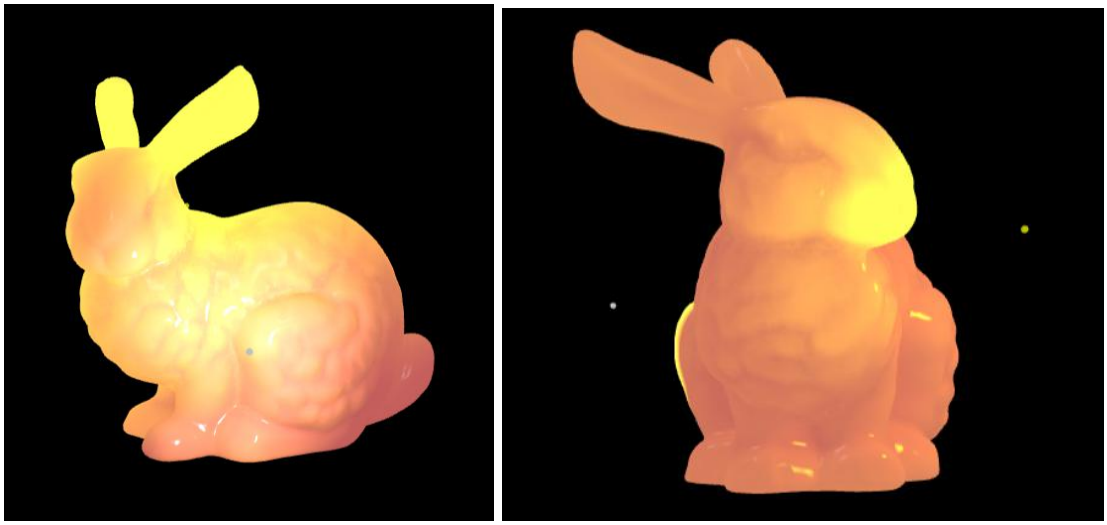

Figure 57 : Exemple de subsurface scattering.

Dans cet exemple, on remarque bien la présence d'une lumière pénétrant la surface de l'objet, et ne se contentant pas d'interagir simplement avec la surface de celui-ci. Dans la seconde image, on s'aperçoit de l'apport de la lumière jaune dans le rendu de la face avant du modèle, alors que celle-ci se trouve derrière lui pourtant. Cependant, ces techniques ne représentent que peu d'intérêts de résultats de rendu en comparaison de coups de calculs élevés.

Pour le PBR, il est également possible de prendre en compte encore plus de matériaux implémentables. Filament dans leur implémentation du PBR par exemple, ont également ajouté des propriétés matérielles supplémentaires permettant de rendre des surfaces avec différentes couches, et ainsi opérer des surfaces possédant une fine couche réflexive, la couche claire.

Figure 58 : Exemple de couche claire.

Dans leur implémentation, cette couche supérieure de l'objet se définit par un coefficient supplémentaire de rugosité appliqué à cette couche, et une puissance de cette couche, définissant de combien celle-ci impactera le rendu direct de l'objet.

On peut également prendre en compte le concept de surfaces anisotrope dans notre rendu. Jusqu'à présent, la totalité des matériaux que l'on peut rendre avec le système présenté sont des matériaux isotropes. L'anisotropie (contraire d'isotropie) est la propriété d'être dépendant de la direction. Quelque chose d'anisotrope pourra présenter différentes caractéristiques selon son orientation (ex : velour).

De nombreux matériaux du monde réel, tels que le métal brossé, ne peuvent être reproduits fidèlement qu'à l'aide d'un modèle de réflectance anisotrope.

Figure 59 : Image de velour.

On voit bien sur cette image que la couleur de l'objet est différente en fonction de son orientation à la lumière (teintes oscillant entre bleu clair et bleu foncé).

Enfin, dans notre approche du PBR, nous sommes partis de l'approximation qu'un point de lumière correspondait à un échantillon d'hémisphère possible. On peut également prendre en compte l'éclairage depuis une carte d'environnement. Cela va demander un raffinement plus important des échantillons à prendre en compte (car on ne connaît plus leur position) et précalculer des informations comme une carte d'irradiance, à l'avance, pour rester sur du temps réel. Cela va grandement augmenter la complexité du rendu PBR, mais va permettre d'obtenir des objets de réflexions adaptative à leur environnement proche (réflexions jaunâtre forte dans le désert, faible et froide de nuit). Cette technique, de part son surcoût et sa spécificité (ne marche que pour une carte d'environnement), semble très peu adaptée pour des applications temps réelles autre que de la visualisation.

Chapitre 6

Conclusion

Le but de ce projet était de présenter et de mettre en place une série de techniques de représentation d'éléments en trois dimensions d'une scène sous forme d'images numériques afin d'obtenir un rendu réaliste en temps réel. Cet objectif, nous le pensons, a été atteint.

En effet, nous avons mis en place un moteur de rendu nous permettant d'interagir avec les objets, et nous y avons ajouté des implémentations de techniques de rendu, et le tout reste exécutable en temps réel. Nous avons mis en place une solution se voulant interactive pour un utilisateur, facilement partageable et utilisable, de par l'utilisation de langages WEB ne nécessitant pas de gros logiciels de rendu (un navigateur suffit). De plus, nous avons pu dresser une architecture modulaire de rendu, permettant la mise en place de pipelines de production d'images numériques à partir de scènes en trois dimensions. Cette architecture nous permet le développement intuitif et rapide de techniques de rendu et de scène où l'utilisateur pourra interagir librement de manière ludique.

Pendant la recherche de techniques et leurs implémentations, nous avons également trouvé d'autres projets similaires au nôtre, mais visant à créer une scène plutôt que d'implémenter des passes de shaders. Il y a, par exemple, Three.js, mais aussi Verge 3D ou encore OneShader.

Cependant, nous n'avons pas pu implémenter tout ce que nous voulions mettre en place. Le monde de la 3D temps réelle évolue constamment, et les techniques de rendu sont nombreuses dans ce domaine, que cela soit pour rendre des matériaux physiquement réalistes en particuliers, implémenter des effets de lumières inédits, jouer sur les ombres d'un objet ou encore améliorer des techniques déjà existantes en termes de performances de rendu ou de confort visuel.

Ainsi, les améliorations pour un tel projet ne manquent pas, à commencer par celles que nous avons pu citer dans la partie précédente. WebGL est une API JavaScript de rendu relativement récente et peu souvent mise à jour, ce qui complique la mise en place de certaines techniques de rendu nécessitant des traitements graphiques modernes.

Finalement, ce projet nous aura permis de mettre en pratique nos connaissances actuelles en rendu interactifs 3D, et de les confronter à de nouvelles pratiques dans un langage de programmation aussi riche que JavaScript.

Chapitre 7

Annexes

Annexes de notre projet, comportant des liens vers les différentes sources de documentation que nous avons prises.

7.1 Bibliographie

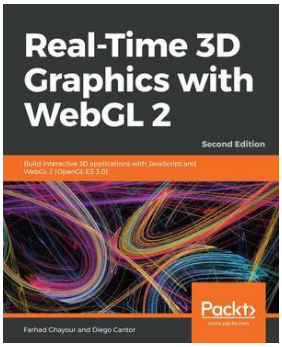

Figure 1 : https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.png
Real-Time 3D Graphics with WebGL 2 : https://github.com/PacktPublishing/Real-Time-3D-Graphics-with-WebGL-2

7.2 Sitographie

Articles opengl :
Gamma correction : https://learnopengl.com/Advanced-Lighting/Gamma-Correction
Bloom : https://learnopengl.com/Advanced-Lighting/Bloom
Shadow mapping : https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
SSAO : https://learnopengl.com/Advanced-Lighting/SSAO
Skybox et environment map : https://learnopengl.com/Advanced-OpenGL/Cubemaps
Exposition : https://learnopengl.com/Advanced-Lighting/HDR

Mais aussi WebGL2 fundamental : https://webgl2fundamentals.org/
Cubemap : https://webgl2fundamentals.org/webgl/lessons/webgl-cube-maps.html
Reflexions : https://webgl2fundamentals.org/webgl/lessons/webgl-environment-maps.html

Et le site/livre introduction to Computer Graphics : https://math.hws.edu/graphicsbook/
Réflexions dynamiques https://math.hws.edu/graphicsbook/source/webgl/cube-camera.html

Source bloom :
wikipedia : https://en.wikipedia.org/wiki/Bloom_%28shader_effect%29
Image du disque d'Airy :
Version anglaise : https://en.wikipedia.org/wiki/Airy_disk
Version française: https://fr.wikipedia.org/wiki/Tache_d%27Airy
Par Fffred sur Wikipédia français — Transféré de fr.wikipedia à Commons par Bloody-libu utilisant CommonsHelper., Domaine public, https://commons.wikimedia.org/w/index.php?curid=16057464

Courbe du disque d'Airy :

Figure 25 : Par Edgar Bonet — Travail personnel, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=44717178

Courbe gaussienne : https://learnopengl.com/Advanced-Lighting/Bloom

Source SSAO :

Article pour la correction d'erreurs de Philip Fortier :

https://mtnphil.wordpress.com/2013/06/26/know-your-ssao-artifacts/

Source Blinn Phong

Figure 43 : https://en.wikipedia.org/wiki/Shading

Source PBR

Figue rugosité et lobe spéculaire :

Filament : https://google.github.io/filament/Filament.html

Github de Sébastien Beugnon :

https://github.com/sbeugnon/pbr-cg-2020

Source subsurface :

https://threejs.org/examples/webgl_materials_subsurface_scattering.html

Clearcoat :

https://threejs.org/examples/?q=aniso#webgl_materials_physical_clearcoat

Autres projet similaires :

Three.js : https://fr.wikipedia.org/wiki/Three.js

Verge3D : https://fr.wikipedia.org/wiki/Verge3D et https://www.soft8soft.com/

"Experience curiosity" est fait avec Verge 3D : https://eyes.nasa.gov/curiosity/

OneShader : https://oneshader.net/