# PROJET DE SYNTHESE D'IMAGE

RAPPORT DE TP

Thomas DELIOT | 11305720

# SOMMAIRE

# INTRODUCTION

Ce projet de synthèse d'image a pour but de générer un rendu de scène réaliste en temps réel.

Ceci a nécessité un travail sur quatre grands axes de développement : création d'un moteur de rendu de scène 3D, implémentation d'un modèle d'éclairage réaliste, implémentation d'une simulation d'occlusion ambiante à l'écran, et implémentation d'une simulation des réflexions entre les différents objets à l'écran.

Ces quatre axes de développement sont abordés et détaillés dans cet ordre par la suite de ce rapport.

## STRUCTURE DU MOTEUR DE RENDU

Pour faciliter la suite du TP et ajouter un axe intéressant à son développement, j'ai décidé de commencer par implémenter un moteur de jeu vidéo en API et en surcouche de *gkit*, avec pour objectif final de supprimer la dépendance à *gkit* (afin de gagner de l'expérience avec *OpenGL*) et d'avoir un début de moteur fonctionnel.

Ayant une certaine expérience avec Unity moi-même, j'ai décidé de copier certains de leur choix de structure. Ainsi le moteur fonctionne à l'aide d'une hiérarchie de scène, composée de *GameObject* ayant un parent et des fils. Chaque *GameObject* contient une position, rotation (stockée en quaternion) et une échelle locale par rapport à son parent, combinés dans une matrice TRS, ainsi qu'un ensemble de fonction pour interagir avec. Ils contiennent chacun une matrice Objet->Monde, utilisée lors du rendu et calculée récursivement en combinant les matrices TRS locales des parents. Un changement de la matrice TRS sur un *GameObject* entraîne la mise à jour des matrices Objet->Monde de cet objet et chacun de ses fils.

Pour ajouter à ces objets les différents composants faisant un jeu vidéo (scripts, maillage, caméra, lumière, etc.), chaque *GameObject* contient une liste de *Component*. La classe *Component* est une classe générique dont chaque composant doit hériter afin de fonctionner, et implémente des fonctions virtuelles de base appelées par le moteur de jeu (*OnStart(), OnDestroy(), Update())*. Ainsi, par exemple, un objet caméra dans la scène est un *GameObject* possédant deux *Component* : *Camera*, un composant permettant le rendu de la scène dans un *framebuffer* avec toutes les fonctions associées, et *FlyCamera*, un composant permettant de déplacer le *GameObject* associé à l'aide de la souris et du clavier.

L'ensemble de la scène est initialisé dans la fonction *InitScene()* de *Engine.h*. Lors d'une *frame*, le moteur effectue différentes étapes :

- Mise à jour des matrices Objet->Monde de chaque *GameObject* marqué comme modifié.
- Appel de la fonction *Update()* de chaque *Component* présent dans la scène.
- Rendu de la scène par la caméra en appelant la fonction *Draw()* de tous les composants *MeshRenderer* trouvés dans la scène.

Enfin pour permettre la suite de ce TP, la méthode de rendu appelée *deferred rendering* a été implémentée. Par conséquent le rendu de la scène se déroule comme suit :

- Rendu de chaque objet dans deux *GBuffer* (+*zbuffer*): l'un contenant la couleur albedo + le coefficient de *roughness*, l'autre contenant la normale + le coefficient de *metallic*.
- Passe finale du rendu en *Post-Effect* calculant l'éclairage et la couleur finale de chaque pixel dans les *GBuffer*, ainsi que la contribution du *SSAO* et du *SSR*.
- Copie de l'image finale dans un *buffer* secondaire afin de pouvoir être utilisée par le *SSR* lors de la *frame* suivante à l'aide d'une reprojection temporelle.

# PHYSICALLY BASED RENDERING

## METHODE

Pour le modèle de matériaux/éclairage lors de ce TP j'ai choisi d'implémenter un pipeline PBR *metallic/roughness* combinant différentes méthodes.

Chaque objet affiché est lié à trois textures :

- Couleur albedo
- Coefficient métallique
- Coefficient de rugosité

Ces trois paramètres sont stockés dans les *GBuffer* et fournis en entrée au calcul d'éclairage final, qui additionne plusieurs contributions lumineuses.

La contribution de la lumière directionnelle présente dans la scène est calculée par la méthode GGX-Smith pour simuler le modèle des micro-facettes.

La contribution ambiante de l'environnement est basée sur la méthode *IBL (Image-Based Lighting)* et ajoute l'irradiance diffuse de l'environnement (avec échantillonnage de la *Skybox* dans la direction de la normale) et la radiance spéculaire reflétée (avec échantillonnage de la *Skybox* dans la direction réfléchie ou réflexion venant du *SSR* lorsqu'elle est valide et disponible).

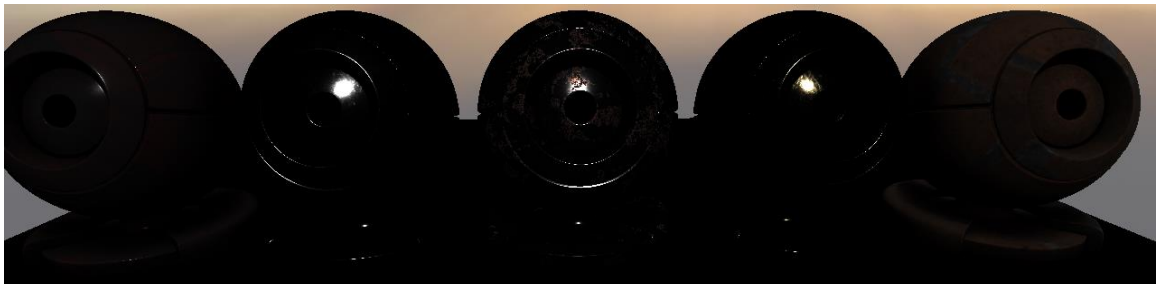
Ces contributions sont sommées pour obtenir la couleur finale à afficher.

## RESULTATS

Pour illustrer les résultats obtenus dans ce rapport, une scène de démo avec caméra fixe a été mise en place. La scène est composée de cinq *shaderball*, dont les matériaux sont, de gauche à droite : plancher vernis, aluminium, fer rouillé, or, mur de brique. Le sol, lui, est en aluminium.

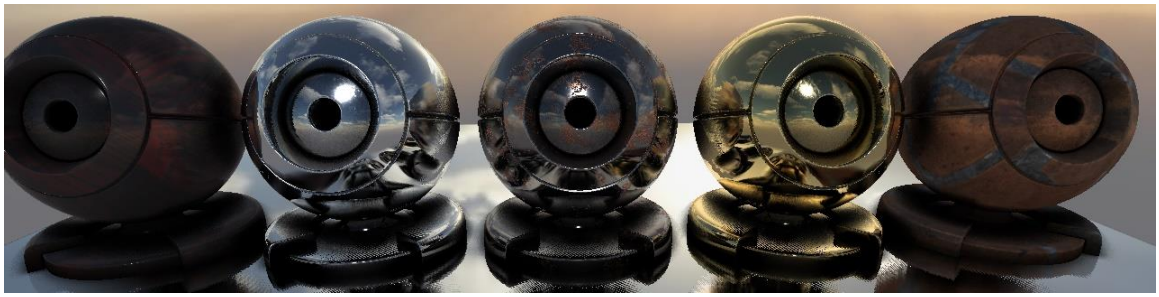
Les figures suivantes montrent l'ajout progressif des différentes contributions lumineuses menant à l'image finale. On observe que les matériaux non métalliques sont majoritairement éclairés par l'irradiance diffuse, alors que les matériaux métalliques et lisses le sont par la radiance spéculaire.

*Sans contribution lumineuse*



*Lumière directionnelle*

*Lumière directionnelle + irradiance diffuse*

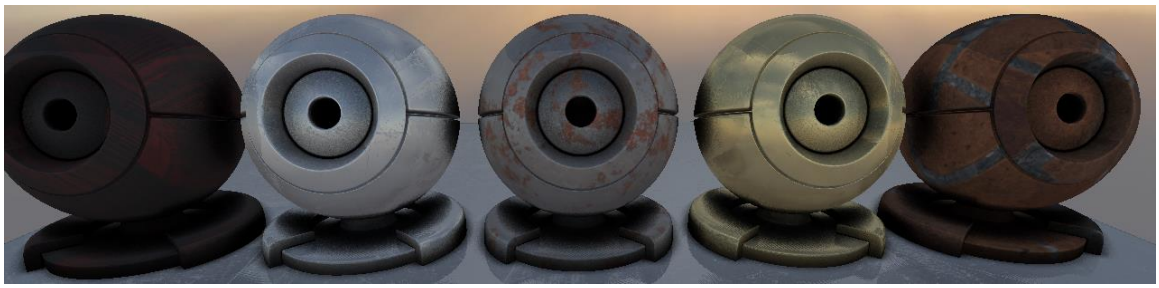


*Lumière directionnelle + irradiance diffuse + radiance spéculaire*

Les figures suivantes montrent les différentes apparences obtenues lorsque l'on modifie les valeurs de métallicité et de rugosité de tous les objets présents dans la scène :

*Valeurs de rugosité/métallicité originales des matériaux*

*Rugosité augmentée*



*Rugosité augmentée et métallicité diminuée*

# SCREEN-SPACE AMBIENT OCCLUSION

## METHODE

L'étape d'occlusion ambiante se déroule après la passe finale de rendu, une fois l'éclairage calculé, et multiplie la couleur finale par un facteur d'atténuation.

Pour calculer ce facteur, un certain nombre de rayons sont générés dans un hémisphère orienté autour de la normale de la surface. Ces rayons sont répartis de manière régulière dans l'hémisphère, mais sont associés à une longueur générée aléatoirement selon une distance maximale qui définit la taille de l'effet à l'écran.

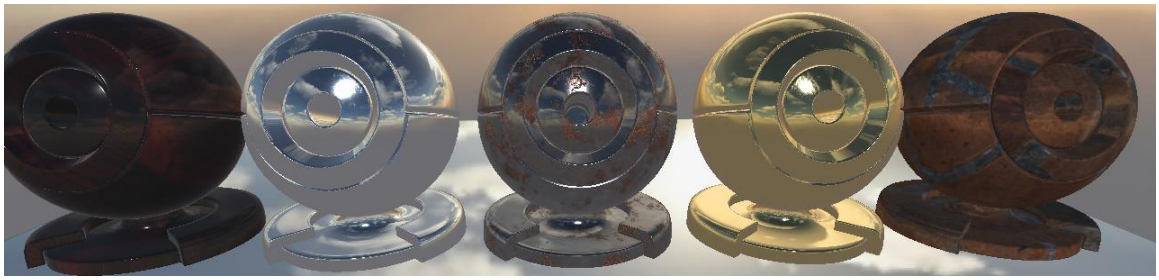
Pour chaque rayon, on teste à la distance générée si l'on se trouve derrière le zbuffer et si tel est le cas, on considère ceci comme une collision avec l'environnement et on incrémente un compteur.

Le facteur d'atténuation est intégré à la fin de la boucle en tant que nombre de collisions trouvées divisé par le nombre total de rayon lancés.

Avec cette méthode seulement, un certain nombre de rayons sont nécessaires pour obtenir une occlusion non bruitée, et le coût en performance est non-négligeable. Ceci n'a pas été réalisé dans ce rendu, mais pour obtenir une occlusion lisse sans bruit de manière bien moins coûteuse, il conviendra de réaliser la technique généralement utilisée dans les jeux vidéo, et exécuter une étape de floutage sur un *buffer* contenant l'occlusion ambiante calculée avec peu de rayons. Comme cet effet est généralement dépourvu de détails de haute fréquence, il serait aussi utile de calculer l'occlusion ambiante dans un *buffer* de taille réduite.

## RESULTATS

Les figures suivantes montrent les résultats du SSAO à différents paramètres de qualité, ainsi que le temps moyen de rendu par *frame* associé, sur un système équipé d'une carte graphique GTX 760 :



*Sans SSAO, 3.5ms*

*12 échantillons hémisphériques, 5ms*

*144 échantillons hémisphériques, 10.2ms*

*1024 échantillons hémisphériques, 61ms*

# SCREEN-SPACE REFLECTIONS

## METHODE

L'algorithme de *SSR* implémenté effectue un *ray-marching* dans l'espace de l'écran en parcourrant le *zbuffer* pixel par pixel, plutôt qu'un *ray-marching* dans l'espace 3D de la scène (qui entraînerait une quantité d'information échantillonnée plusieurs fois sans intérêt).

En premier lieu, la position 3D du point vu par la caméra à un pixel donné est recosntruite grâce au *zbuffer*. Connaissant également la normale à ce point, on calcule la direction réfléchie entre le vecteur caméra->point et la normale de la surface. Ceci nous donne la direction dans laquelle rechercher une collision avec la scène afin de trouver la couleur réfléchie, si elle existe à l'écran. La couleur trouvée est fournie au modèle d'éclairage PBR comme une source de radiance spéculaire et est donc utilisée de manière physiquement correcte.

Le *SSR* est un algorithme qui trouve des réflexions au mieux avec les informations présentes à l'écran, mais un grand nombre de réflexions restent introuvable avec cette méthode. Pour éviter des cassures nettes visibles dans les réflexions lorsqu'on sort des conditions optimales, la force de la réflexion est progressivement réduite dans différents cas :

- Lorsqu'on approche le nombre d'itérations maximal
- Lorsque la position de collision trouvée approche le bord de l'écran
- Lorsque la direction réfléchie est tournée vers la caméra (*worst-case scenario*)

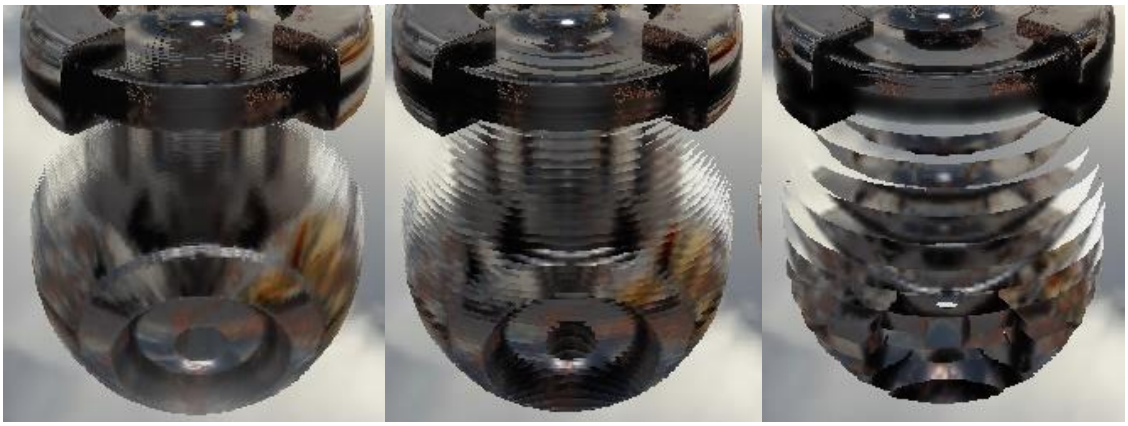
La figure suivante montre le résultat obtenu par l'implémentation du *SSR* avec des paramètres par défaut. On peut observer le *fade out* progressif de la réflexion au bord de l'écran, sur la réflexion des deux *shaderballs* situés aux extrémités.

*Résultats avec SSR et SSAO*

Les sous-parties suivantes aborderont les différentes optimisations implémentées, la nécessité de la reprojection temporelle et l'exploration de différentes solutions pour la problématique des reflets diffus.

## OPTIMISATIONS

Un bon moyen de réduire le coût du *SSR* est d'augmenter la taille du pas dans l'espace écran : au lieu d'avancer de pixel en pixel, on avance d'un certain nombre de pixels, donné par le paramètre *pixel stride* :



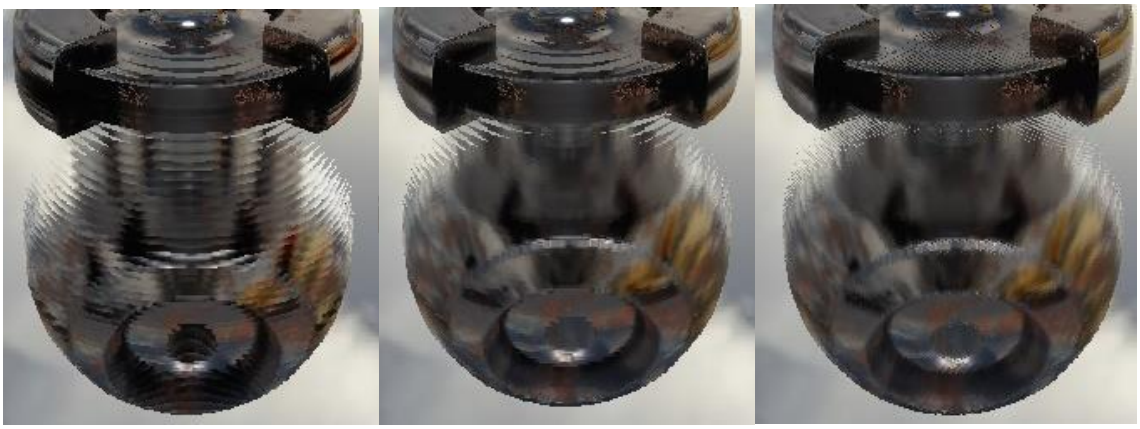
*De gauche à droite : pixel stride à 1 (10.1ms), 8 (8.4ms), et 32 (6.5ms)*

Ceci entraîne évidement des artefacts dus aux collisions ratées ou trouvées avec un décalage, qu'il convient de mitiger. Deux optimisations ont été implémentés à ce but : une recherche binaire et une perturbation.

La recherche binaire effectue un certain nombre de pas supplémentaire dans la boucle du *SSR* une fois la collision trouvée, en revenant en arrière puis en avançant à nouveau avec une taille de pas divisée par deux. A chaque pas de recherche binaire, on regarde si l'on est devant ou derrière le *zbuffer*, et on avance ou on recule en conséquence, en divisant à chaque itération la taille de pas par deux. Quatre pas supplémentaires de recherche binaire sont souvent suffisants pour grandement réduire l'erreur généré par un *pixel stride* supérieur à 1.

La perturbation rajoute simplement un décalage pseudo-aléatoire de la position de départ à l'écran de la recherche du *SSR*. Ceci permet de mélanger les pixels proches, limitant les effets de bande visible de la même manière qu'un écran limiterait le *color banding* par *dithering*.

La figure suivante illustre progressivement les améliorations ajoutées lorsqu'une taille de pas supérieure à 1 pixel est utilisée pour optimiser les performances :



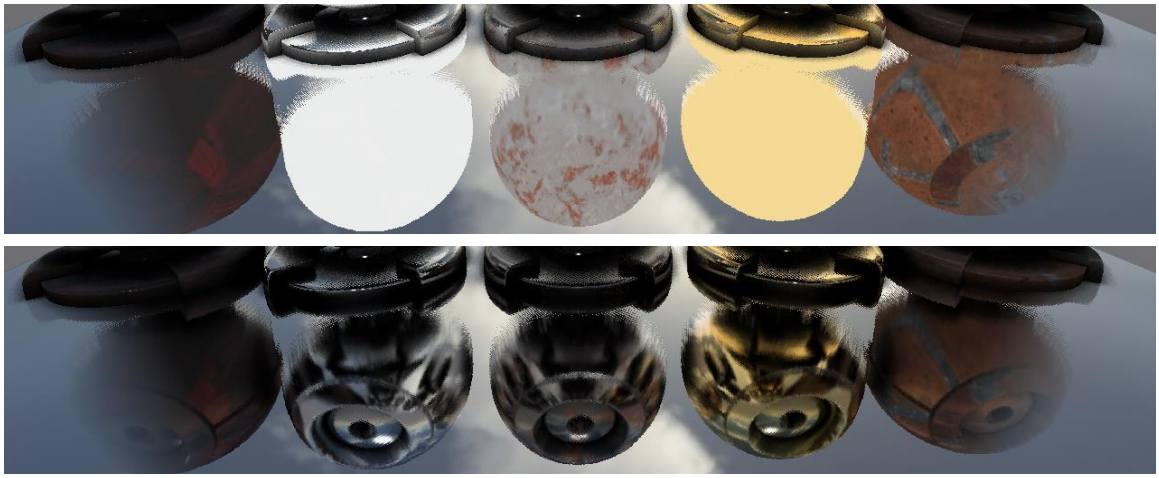
*De gauche à droite : pixel stride à 8 sans amelioration, avec 4 pas de recherche binaire, avec perturbation*

Avec ces deux optimisations, même un *SSR* avançant avec un pas de 32 peut donner un rendu acceptable, pour un coût supplémentaire négligeable :

*De gauche à droite : pixel stride à 32 sans et avec améliorations*

## REPROJECTION

Pour que le *SSR* fonctionne convenablement, il est nécessaire d'utiliser l'image générée précédemment par le moteur de rendu une fois la collision avec la scène trouvée, plutôt que l'image actuelle. Ceci est particulièrement vrai lorsqu'un modèle d'éclairage PBR est implémenté, avec des matériaux métallique dont la couleur finale diffère très fortement de leur couleur albedo avant éclairage, comme le montre la figure suivante :



*De haut en bas : utilisation de l'image actuelle et utilisation de l'image précédente reprojetée*

Pour utiliser l'image précédente plutôt que l'actuelle, l'algorithme du *SSR* n'a pas été modifié. On utilise plutôt le point de collision retourné par la fonction d'une manière différente : au lieu d'échantillonner directement l'image actuelle aux coordonnées du point, on repasse le point en coordonnées du monde à l'aide des matrices de projection et de vue inverses, puis on reprojette le point avec les matrices de vue et de projection de la *frame* précédente. Enfin, on utilise ces nouvelles coordonnées pour échantillonner l'image précédente.

Ceci permet d'obtenir une réflexion exacte des objets rendus, avec leur éclairage final et les effets supplémentaires qui leur sont appliqué, comme le *SSAO*. Le taux d'erreurs de reprojection est très négligeable (ce n'est qu'en se déplaçant très rapidement que l'on cherchera à échantillonner une information n'existant pas dans l'image précédente). Un effet secondaire est toutefois la réflexion du *SSR* lui-même au sein des réflexions : ceci donne une impression de réflexions multiples qui a l'air intéressant et rempli agréablement l'image de détail, mais ces réflexions sont totalement fausses. Ceci est un problème relativement négligeable dû à la petite taille de ces réflexions secondaires dans l'image.

Des problèmes plus importants peuvent se poser si l'on souhaite rajouter des objets transparents ou volumétriques dans la scène : on obtient alors une information qui n'est plus confinée à une seule profondeur. La reprojection utilisant la profondeur du *zbuffer* pour fonctionner, les informations volumétriques ou transparentes rendues dans la scène à une profondeur différente ne pourront pas être reprojetées correctement en mouvement (bien que des rotations seules de la caméra n'entraîneront pas d'erreurs).

## REFLETS DIFFUS

Un problème important du *SSR* est qu'il s'agit d'un effet limité aux réflexions parfaitement nettes, normalement réservées aux miroirs purs. En effet, plus le matériau réfléchissant est rugueux, moins la réflexion sera nette et confinée à sa forme initiale. La réflexion devient diffuse.

Il existe plusieurs moyens de simuler ceci en dehors de la seule approche correcte (approche brute, lancer de multiples rayons). J'ai personnellement implémenté une astuce simple : échantillonner l'image précédente au point de collision à un niveau de détail plus ou moins faible selon la rugosité du matériau concerné. Il faut donc s'assurer que des *mip-maps* soient disponibles pour les deux éléments échantillonnés : l'image précédente et la *skybox*.

La figure suivante montre le résultat de cette modification en augmentant la rugosité des objets présents à l'écran. On observe que l'effet est intéressant au sein même des réflexions, ou le détail est correctement réduit et la réflexion prend une apparence moins nette (mais l'interpolation linéaire des *mip-maps* ne donne pas un rendu idéal). On voit toutefois nettement la limitation de la méthode utilisée avec le *SSR* : les contours des réflexions restent parfaitement nets. D'autres astuces pourraient être utilisées (calculer les réflexions dans un *buffer* séparé et y appliquer un flou adaptif selon la rugosité du matériau, par exemple).

*Simulation des reflets diffus par mip-map, de haut en bas : rugosité originale, rugosité augmentée*

J'ai personnellement décidé de tenter une implémentation rapide de la méthode *brute-force* afin de pouvoir visualiser une référence de l'effet à simuler. Pour réaliser cela, au lieu de ne faire qu'un test de collision dans la direction réfléchie avec le *SSR*, j'ai intégré le *SSR* au sein d'une boucle effectuant un nombre donné d'échantillons, chacun avec une direction plus ou moins perturbée aléatoirement par rapport à la direction de la réflexion centrale, selon la rugosité du matériau. La couleur finale de la réflexion est la moyenne de tous les rayons testés.

Le coût en performance est très important (de plus l'implémentation a été faite rapidement sans chercher à optimiser), mais l'effet recherché est bel et bien présent. On observe des réflexions diffuses, qui sont nettes lorsque la distance entre l'objet et sa réflexion est petite (au pied des *shaderball*, par exemple) et deviennent de plus en plus diffuses lorsque la distance augmente. Même avec 128 échantillons toutefois, le bruit reste visible. Il serait possible également de calculer les réflexions dans un *buffer* temporaire avec un faible nombre de rayons et d'y appliquer un flou pour obtenir un effet approximatif mais sans bruit et moins coûteux.

Les figures suivantes montre les résultats obtenus avec cette approche *brute-force* pour 16 et 128 échantillons, avec une rugosité faible et élevée, ainsi que le temps moyen de rendu par *frame* associé :

*16 échantillons, de haut en bas : rugosité originale (115ms), rugosité augmentée (165ms)*

*128 échantillons, de haut en bas : rugosité originale (900ms), rugosité augmentée (1500ms)*

## SOURCES


- https://freepbr.com/
- http://casual-effects.blogspot.fr/2014/08/screen-space-ray-tracing.html
- http://www.kode80.com/blog/2015/03/11/screen-space-reflections-in-unity-5/
- https://github.com/JoshuaSenouf/GLEngine