ENSICAEN
6, bd maréchal Juin
F-14050 Caen cedex 4

Spécialité Informatique - 2e année

Rapport de projet

Rendu HDR et illumination par image

Adrien Calendron
Alexis Legrand
Stéphane Saffré

Suivi Ensicaen : Cédric Pelvet

Présentation

Dans de nombreux jeux vidéo récents, on peut voir apparente, dans les publicités ou teasers de promotion, la mention «HDR Rendering». Qu'est-ce que cela signifie ? On se doute que cela a à voir avec les graphismes, mais qu'en est-il vraiment ?

Le HDRR, autrement appelé High Dynamic Range Rendering, est un procédé numérique permettant d'améliorer les contrastes d'une photo ou d'une séquence vidéo. Elle admet de nombreuses applications, dont les jeux vidéo qui en sont un bel exemple. Mais il ne s'arrête pas à ce domaine: sa présence se remarque partout où l'audiovisuel a trait. Les films d'animation l'utilisent, le cinéma, la publicité, la photographie en général...

Dans ce projet, nous nous proposons d'étudier cette technique. Nous allons donc expliquer ce qu'est le HDR et une image HDR, puis nous verrons une application de cela dans la photographie. Enfin, nous expliquerons un format de fichier spécialement conçu pour enregistrer une image HDR, le OpenEXR.

I. Le principe du Rendu HDR (HDRR)

1. Introduction

Le rendu HDR (ou HDRR : High dynamic Range Rendering), est un principe de rendu calculant l'illumination d'une scène dynamiquement permettant un éclairage d'une scène 3D beaucoup plus réaliste. En effet, cette technique permet de respecter la grande échelle de variation de la luminosité d'un environnement réel. Prenons un exemple:

Fig 1.1 Prise du jeu Half-Life 2 comparant le HDRR et le LDRR (Source: Wikipedia).

Si cette scène était réelle, ce que nous verrions serait le résultat de la première image. En effet, la luminosité à l'intérieur du bâtiment est très faible comparée à celle de l'extérieur, qui est éclairé directement par le soleil. C'est pourquoi notre oeil tente de s'adapter à la luminosité intérieure pour lui permettre de voir quelque chose ; la lumière extérieure paraît alors éblouissante, c'est-à-dire très blanche, et laisse apparaître des traînées de lumière autour de la fenêtre. Ceci est un phénomène physique complexe que l'on n'expliquera pas ici mais que l'on tentera de reproduire, qui est appelé effet bloom (aussi appelé effet glare).

Le taux de contraste dans un environnement réel peut être énorme : plusieurs millions pour un. L'oeil humain possède un taux de contraste d'environ 1 000 000 : 1. Mais l'adaptation de l'oeil à la luminosité étant assez lente (puisque due à des variations de propriétés chimiques), le taux de contraste se trouve plutôt à tout instant autour de 10 000 : 1.

Cependant, il est inutile pour le moment d'essayer de représenter des contrastes plus élevés, puisque la plupart des écrans actuels atteignent un taux de 10 000 : 1, au mieux.

Prenons un autre exemple, celui de la photographie : de même que pour les téléviseurs, les capteurs d'un appareil numérique sont incapables de reproduire ce taux de contraste naturel. Il est alors parfois difficile de prendre une photo qui puisse parfaitement recréer le paysage observé. Sur les deux photographies suivantes, le problème se pose : à gauche, le premier plan de la photo est sous-exposé, alors que le second plan est suffisamment éclairé. A l'inverse, sur la photographie de droite, le premier plan reproduit bien la lumière naturelle, mais cette fois, le second plan est largement surexposé. La différence entre les deux photographies a été obtenue en allongeant le temps de pose, afin de chercher à obtenir la reproduction la plus parfaite possible du paysage.

Fig 1.1 Photo de gauche : temps de pose faible, sous-exposition du premier plan.
Photo de droite : temps de pose long, surexposition du second plan.

Regardons les histogrammes des photos :

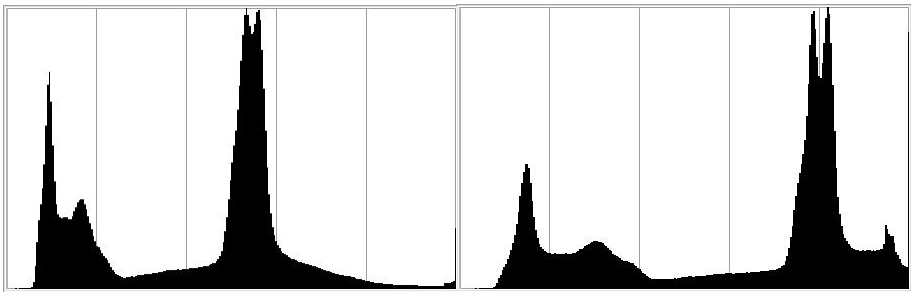


Fig 1.2 A gauche : histogramme de la photo sous-exposée.
A droite : histogramme de la photo surexposée.

Le premier histogramme montre que la photo de gauche contient de très nombreux pixels de luminosité intermédiaire (grand pic au milieu) mais également beaucoup de pixels sombre (pic dans la partie gauche). Celui de gauche montre que la photo correspondante comporte beaucoup trop de pixels clairs (grand pic dans la partie droite), une quantité non négligeable de pixels sombres (pic dans la partie gauche), et quasiment aucun de luminosité moyenne (partie centrale). L'idéal serait, en photographie courante, d'avoir une courbe gaussienne.

Pour rééquilibrer les contraste, le HDRR est alors un outil des plus efficaces.

Lors d'un rendu classique LDRR (Low Dynamic Range Rendering), chaque image est codée sur 24 bits, c'est à dire 8 bits par composante de la lumière (RVB). La couleur est donc codée pour chaque composante sur l'intervalle [0, 255], ou sur [0.0, 1.0] en flottant. Le noir est donc représenté par 0.0 et le blanc par 1.0, ce qui laisse peu de place pour toutes les luminosités intermédiaires sachant que le taux de contraste peut atteindre plus de 10 000 : 1.

Avec ce type d'encodage, les zones sombres d'une image sont saturées de noir et les zones très lumineuses, de blanc. Or en réalité, l'oeil permet de voir une grande variété de détails, qu'il regarde une zone lumineuse ou une zone sombre.

Le but du rendu HDR est justement de permettre de voir aussi bien les détails d'une zone très lumineuse que ceux d'une zone très sombre. Pour cela, les composantes RVB de la lumière sont encodées par exemple sur 32 bits chacune, étalant l'échelle des valeurs entre 0 et 65 535. Ainsi ce format permet de coder une grande variété d'intensités lumineuses et on pourra accentuer lors du rendu, les zones sombres ou les zones lumineuses de l'image suivant la luminosité de la zone d'observation de la caméra. On doit alors mettre en place un système de gestion de l'exposition de la caméra qui simulera l'adaptation de l'oeil à la luminosité de la scène afin de ne pas obtenir une

image sous-exposée (noire) ou surexposée (blanche).

Lors du rendu HDR de photos, on prend en compte les contraste de chaque photo, puis à l'aide d'alorithmes, on obtient un résultat dont l'histogramme se retrouve équilibré, centré.

Mais le rendu HDR n'est pas seulement important pour l'illumination directe d'une scène. En effet le HDRR est très utile pour simuler des effets de réflexion de la lumière, de flou et d'illumination indirecte réalistes. En effet, si on prend l'exemple de la réflexion lumineuse sur l'eau, en réalité, seule une faible partie de la lumière est réfléchie. Si ce rayon lumineux provient d'un soleil dont la couleur blanche est codée avec 8 bits par composantes, le rayon réfléchi aura une intensité très faible à cause du facteur de réflexion lui aussi très faible. Or, la luminosité réelle du soleil est très élevée et peut être codée dans une image HDR. Le rayon réfléchi aura alors une luminosité élevée, proche de la réalité et cela malgré le coefficient de réflexion faible de l'eau.

Fig 1.3 Prise du jeu Half-Life 2 illustrant l'intérêt du HDRR
pour la simulation de la réflexion lumineuse (Source: Wikipedia).

L'intérêt du rendu HDR devient donc évident pour la simulation de tout phénomène physique faisant intervenir le passage de la lumière à travers des matériaux aux propriétés différentes comme la réflexion, la réfraction, l'effet bloom, la radiosité, les caustics, le subsurface scattering, etc...

2. Utilisation d'une cube map

2.1 Introduction

Le but du rendu HDR est d'illuminer les objets d'une scène 3D à partir de l'environnement alentour afin d'obtenir un éclairage réaliste.
En effet, l'éclairage le plus simple et le plus rapide consiste à modéliser la source lumineuse supposée ponctuelle par une ou plusieurs lampes que l'on dispose dans une scène. Or ce type d'éclairage donne des résultats peu satisfaisant en terme de réalisme puisque qu'en réalité les sources lumineuses sont loin d'être ponctuelles et les objets éclairés participent eux même aussi à l'illumination de la scène.

La technologie HDR, c'est à dire, le respect de l'étendue des variations d'intensité lumineuse d'une scène elle seule ne suffit pas; il est aussi nécessaire de modéliser l'environnement grâce à une cube map.

En effet la cube map n'est rien de plus qu'un cube entourant la scène 3D sur laquelle on aurait plaqué la texture représentant son environnement. Par exemple, imaginez vous dehors, comme étant le seul objet 3D de la scène que vous occupez. La Cube Map est alors un cube dont vous seriez le centre et sur lequel on aurait peint en trompe l'oeil tout le paysage environnant, vous faisant croire vraiment que cet environnement est réel.

Nous savons tous bien, de manière plus ou moins empirique qu'un objet particulié est éclairé par l'environnement qui l'entoure. C'est ce phénomène que l'on tente de modéliser grâce à une cube map.

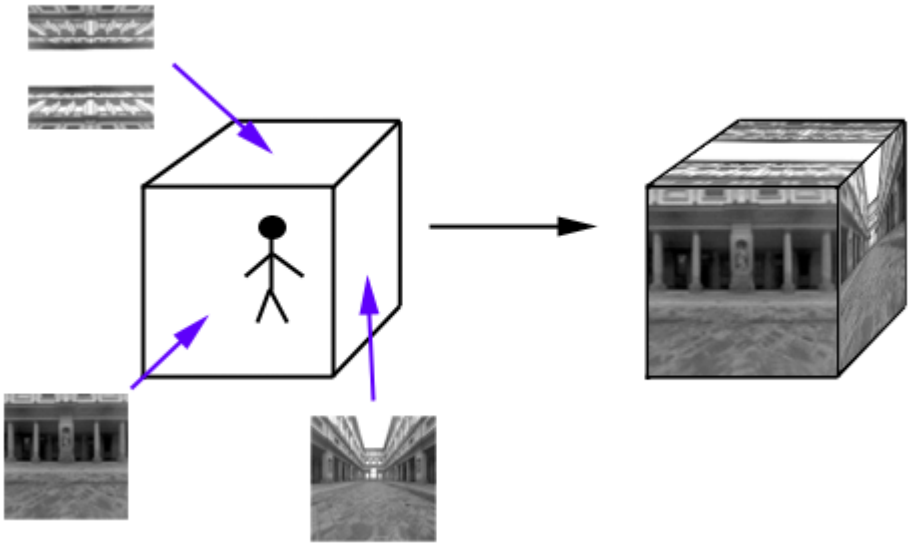


Fig 2.1 Illustration du principe de la cube map.

On comprend ici toute la signification du rendu HDR. En effet il devient évident qu'il suffit alors de plaquer les textures HDR d'un environnement particulier pour en éclairer les objets de manière réaliste, c'est à dire de façon à ce qu'ils s'intègrent bien à l'éclairage procuré par la scène.

Pourtant un lourd problème s'impose ici. Comment va t'on calculer la couleur d'un point d'un objet de la scène quand on sait que chacun d'entre eux sont éclairés par tout l'environnement visible autour d'eux? Il existe une technique qui consiste à modéliser ce phénomène en pré calculant une cube map de réflexion diffuse. Cette cube map permet de déterminer la couleur diffuse de n'importe quel point d'un objet 3D de la scène moyennant la connaissance de la normale à sa surface.

Il existe le même type de map pour le cas de la réflexion spéculaire.

2.1 Création d'une map de réflexion diffuse (diffuse reflection map)

***Rappel**:* La réflexion diffuse, désigne la réflexion de la lumière par une surface non régulière. C'est à dire que la lumière réfléchie est diffusée dans toutes les directions à partir du point d'incidence de la lumière émise à la surface.

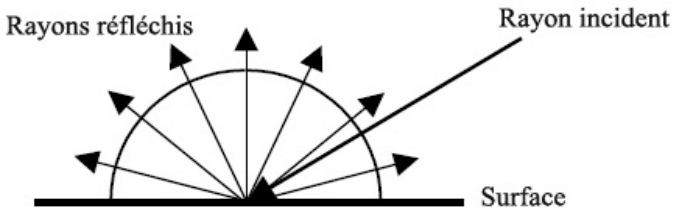


Fig 2.2 Illustration de la réflexion diffuse.

Le modèle d'éclairage de Phong postule que les rayons réfléchis pour la composante diffuse ont une intensité lumineuse proportionnelle à l'intensité lumineuse du rayon incident, au coefficient de réflexion diffuse du matériau ainsi qu'à l'angle entre la normale à la surface du matériau et la direction du rayon incident. On suppose aussi que tous les rayons réfléchis ont la même intensité.

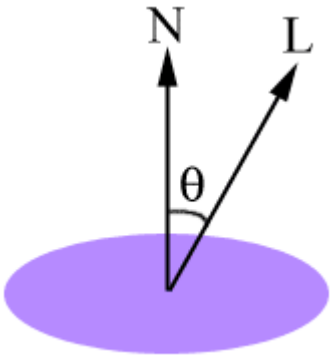


Fig 2.3 : Calcul de l'irradiance

N : La normale à la surface du matériau
L : La direction du rayon lumineux incident
θ : L'angle entre N et L
N et L sont normalisés.

L'intensité, **I** des rayons réfléchis est alors :

$$\mathbf{I = L_d * M_d * \cos(\theta);}$$

Où :
$\mathbf{L_d}$ est la l'intensité de la source lumineuse.
$\mathbf{M_d}$ est le coefficient de réflexion diffuse du matériau.

Il n'est pas difficile de déterminer la valeur de cos(θ) grâce aux vecteurs N et L. En effet cos(θ) n'est alors rien d'autre que le produit scalaire de N par L :

$$\cos(\theta) = N.L$$

Ce modèle de calcul de l'intensité diffuse extrêmement simple, fonctionne dans le cas où la source d'éclairage est une source lumineuse ponctuelle. Or l'illumination à partir d'une image consiste à éclairer une scène en considérant que l'environnement entier est une source lumineuse modélisée par une cube map.

La cube map comporte sur chaque face une image représentant une partie de l'environnement. C'est alors chacun des texels de la cube map qu'il faut considérer comme une source lumineuse et le calcul de l'intensité diffuse d'un matériau en un point P de normale N_p est alors une somme :

$$I(P, N_p) = (1/4\pi)*\sum_i L(P, \omega_i)*A_i*F_d$$

$I(P, N_p)$: L'intensité diffuse au point P de normale N_p.

$L(P, \omega_i)$: L'intensité de la source lumineuse i au point P et de direction ω_i.

A_i : L'angle solide représenté par la source lumineuse i.

F_s : Correspond à cette fonction (voir précédemment) :

$$N_p.\omega_i * M_d$$

L'intensité diffuse est calculée pour toutes les normales possibles. Celle ci est donc stockée dans une cube map où chaque texel représente l'intensité diffuse d'un point de normale correspondante à celui ci. Pour effectuer ce pré-calcul on a donc supposé que le point P est fixe et placé au centre de la cube map. Cette approximation est acceptable si on considère que la scène observée est suffisamment petite comparée à la taille de l'environnement représenté par la cube map.

Voici ce que donne le calcul de la cube map de la réflexion diffuse:

Fig 2.4 : A gauche, les 6 faces de la cube map de l'environnement.
A droite, la cube map de l'intensité diffuse.

2.3 Création d'une map de réflexion spéculaire (diffuse specular map)

Rappel: La réflexion spéculaire est la réflexion parfaite de la lumière sur un miroir. Un rayon incident sur la surface d'un matériau est complètement réfléchi dans une seule direction à partir du point d'incidence. La réflexion spéculaire est le complément de la réflexion diffuse.

Le modèle d'éclairage de Blinn-Phong propose une approche permettant de calculer l'intensité spéculaire en un point de la surface d'un matériau de normale N grâce au demi vecteur H réprésentant le vecteur moyen entre la direction de l'oeil et celle de la source lumineuse.

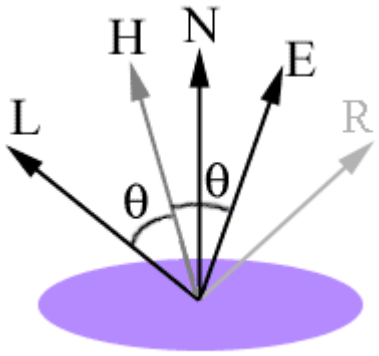


Fig 2.5 : Calcul de l'intensité spéculaire

L : La direction du rayon lumineux.
H : Le demi vecteur
R : le rayon lumineux réfléchi par sur la surface.
N : La normale à la surface.
E : la direction de l'oeil

On a alors:

H=normalise(E+L)

Tous vecteurs entrant en jeu doivent être normalisés car ceux si servent à calculer des produits scalaires. On normalise donc le vecteur H calculé.

Et l'intensité spéculaire au point d'incidence est:

$$I = (N.H)^{s} * L_s * M_s$$

L_s : Intensité spéculaire de la source lumineuse
M_s : Coefficient de réflexion spéculaire du matériau.
s : Coefficient de brillance.

Si le rayon réfléchi R est exactement la direction de l'oeil E, l'intensité spéculaire est donc maximale et plus R s'éloigne de E plus l'intensité spéculaire est faible. Ceci se traduit par l'éloignement de H par rapport à la normale N. En effet si H=N alors R=E et H.N=1. Plus H.N se rapproche de 0, plus la direction du rayon réfléchi est éloigné de celle de l'oeil.

Mais sans ce coefficient s à la puissance, cette formule serait apparente à la formule calculant l'intensité diffuse des rayons réfléchis. Or contrairement au cas de la réflexion diffuse, les rayons réfléchis spéculairement sont très directifs suivant la brillance du matériau. En effet plus le

matériau sera poli tel un miroir, plus la réflexion spéculaire sera directive. On observera une tache de brillance à la surface réduite à un point si on considère la source lumineuse ponctuelle.

Ce coefficient s à pour but de modéliser la brillance du matériau. Plus ce coefficient sera élevé, plus N.H va tendre rapidement vers 0 et donc plus l'intensité de la tache spéculaire va se dégrader rapidement.

s=8 s=64 s=128

Fig 2.6 : Influence de la valeur de la brillance, s sur la taille de la tache spéculaire.

Au sein d'une cube map, l'intensité spéculaire d'un matériau en un point P de normale N_p est alors une somme :

$$I(P, N_p) = (1/4\pi)*\sum_i L(P, \omega_i)*A_i*F_s$$

$I(P, N_p)$: L'intensité spéculaire au point P de normale N_p.

$L(P, \omega_i)$: L'intensité de la source lumineuse i au point P et de direction ω_i.

A_i : L'angle solide représenté par la source lumineuse i.

F_s : Correspond à cette fonction (voir précédemment) :

$$(N.H)^s * M_s$$

De même que pour la création de la diffuse cube map, chaque texel de la specular cube map stocke la valeur de l'intensité spéculaire pour tout point P au centre de celle ci, quelque soit sa normale N_p.

2.4 Bilan

Le calcul d'une telle cube map peut être très long suivant la taille de cette dernière. C'est pourquoi on limite la taille de la cube map de base (représentant l'environnement) pour effectuer le calcul. En effet, il est inutile d'utiliser une cube map «haute-définition» pour ce calcul puisque tous les détails de l'environnement vont être amenés à disparaître à cause de l'effet de flou que créé par les produits scalaire dans les fonctions de convolution F_d et F_s.

On peut aussi mettre en place quelques optimisations lors du calcul de l'intensité de chaque point des cube maps :

- ➔ En supposant que l'environnement visible du point P se limite à l'hémisphère supérieur au dessus de lui. Cette approximation fonctionne très bien pour les surfaces peu courbées.

➔ En complément de l'hypothèse précédente, il arrive un moment ou les fonctions de convolution F_d et F_s sont tellement faibles que l'influence de l'environnement sur la couleur du point P varie imperceptiblement. On peut alors limiter le calcul avec des directions lumineuses ω_i limitées à un cône dont P est le sommet.

3. Le contrôle de l'exposition

Nous avons vu précédemment que la gamme de luminosité du scène réelle est très vaste. L'oeil comme une caméra est incapable d'observer toute cette étendue d'un seul coup et n'en observe qu'une partie. Cela lui est possible en modifiant la taille de l'iris afin de faire rentrer plus ou moins de lumière suivant le taux de luminosité.

Dans une caméra, il est possible d'effectuer la même chose que l'oeil mais aussi de modifier le temps d'exposition contrairement à l'oeil. C'est à dire, la durée pendant laquelle le capteur photosensible va recevoir les rayons lumineux pour l'acquisition d'une image. Les caméras sont capables de gérer des temps d'exposition plus ou moins élevés suivant la rapidité d'acquisition de celle ci car cela se répercute aussi sur le nombre d'images par secondes.

Le but du contrôle de l'exposition est de contrôler l'intensité moyenne d'une image afin qu'elle ne se soient ni trop lumineuse (saturation) ni trop sombre. Lors d'un rendu HDR, le framebuffer est donc une image HDR et n'est donc pas visualisable directement sur nos écrans actuels qui ne supporte pas l'affichage HDR. Il convient donc de passer le HDR framebuffer en LDR framebuffer.

chaque pixel du HDR framebuffer de couleur **couleur_HDR,** est converti en la couleur **couleur_LDR** dans le LDR framebuffer grâce au facteur d'**exposition**:

$$couleur\_LDR = couleur\_HDR * exposition$$

Il convient donc de bien choisir ce facteur d'exposition. Si on considère l'intensité lumineuse moyenne dans une image HDR **Imoy**, on peut dire que celle ci correspond à la valeur **0.5** dans une image HDR, l'intensité y variant entre 0.0 et 1.0. On obtient donc:

$$exposition = 1/(2*Imoy)$$

En effet pour **couleur_HDR=Imoy**, on a : **couleur_LDR=0.5**.

On détaillera dans la partie III la manière dont est calculée **Imoy** grâce à OpenGL.

4. L'effet Bloom ou effet Glare

L'effet Bloom, aussi appelé effet Glare, consiste à modéliser l'effet de halo lumineux que l'observe à la périphérie des sources de lumière et aussi des matériaux fortement lumineux. Ce phénomène est très souvent observé dans la réalité et donc lors d'un rendu HDR.

Fig 4.1 : Illustration de l'effet bloom dans le jeu Farcry. On observe ici le soleil couchant s'étaler sur les bordures de l'hélicoptère.

L'implémentation OpenGL de l'effet Bloom sera expliquée dans la partie III.

II. Création et stockage d'une image HDR

1. Le HDRR en photographie

Dans cette partie, nous allons expliquer une utilisation du HDRR. Celui-ci va nous permettre d'obtenir une photographie du paysage exposé dans la partie I.1, et dont les contrastes auront été rétablis grâce au HDRR. L'idée à suivre est de prendre plusieurs clichés du paysage à des temps de pose différents. On obtiendra donc des photos sous-exposées et surexposées. Le HDRR utilisera chacune d'elle pour en retirer les détails et les intégrer dans une photographie dont les zones mises en défaut par la luminosité seront correctement visibles.

La manoeuvre consiste à utiliser un trépied afin d'éviter les problèmes de mouvements de l'appareil, puis à prendre une photo de référence. A partir de celle-ci, on en prendra au moins deux supplémentaires, avec des temps de pose encadrant la première prise. Ainsi, les photos seront par exemple à -1EV, 0EV, et +1EV.
(Note : EV : Exposure Value : combinaisons de réglage (temps de pose et ouverture) de l'appareil permettant de donner la même exposition. On a la relation :

$$EV = \log_2 N^2/t$$

Où N = ouverture du diaphragme, nombre f dans le quotient 1/f, et t = temps de pose.)

Dans le cadre de ce projet, les photos ont été prises avec une ouverture 1/4, et un temps d'exposition de 1/50 à 1/2000 (secondes), ce qui donne des photos de 9,6 EV (1/50s) à 15 EV (1/2000). En particulier, pour les deux photos précédemment exposées, on a 11 EV (1/250s) et 12 EV (1/500s) respectivement.

Pour le traitement des images, on utilisera FDRTools Basic ou Qtpfsgui, deux logiciels permettant la création d'images HDR. Chacun d'eux permettent la création d'une image HDR, mais leurs fonctionnalités sont différentes, et peuvent utilisés de manière complémentaire. C'est pourquoi nous exposons la technique de rendu HDR pour ces deux logiciels.

Après avoir importé les images dans le logiciel, et après un alignement rapide des photos, la manoeuvre sera sensiblement la même, quelque soit le logiciel utilisé : il faut tout d'abord créer l'image HDR en sélectionnant les photos que l'on souhaite prendre en compte.

Pour cela, FDR Tools donne beaucoup plus de libertés puisqu'il permet d'importer toutes les images dans un premier temps, puis de décocher celles que l'on souhaite retirer de l'image HDR à créer. Il donne également les histogrammes de chaque photo, ce qui permet de faciliter la sélection. Avec Qtpfsgui, on doit importer les seules photos voulues dès le début de l'opération.

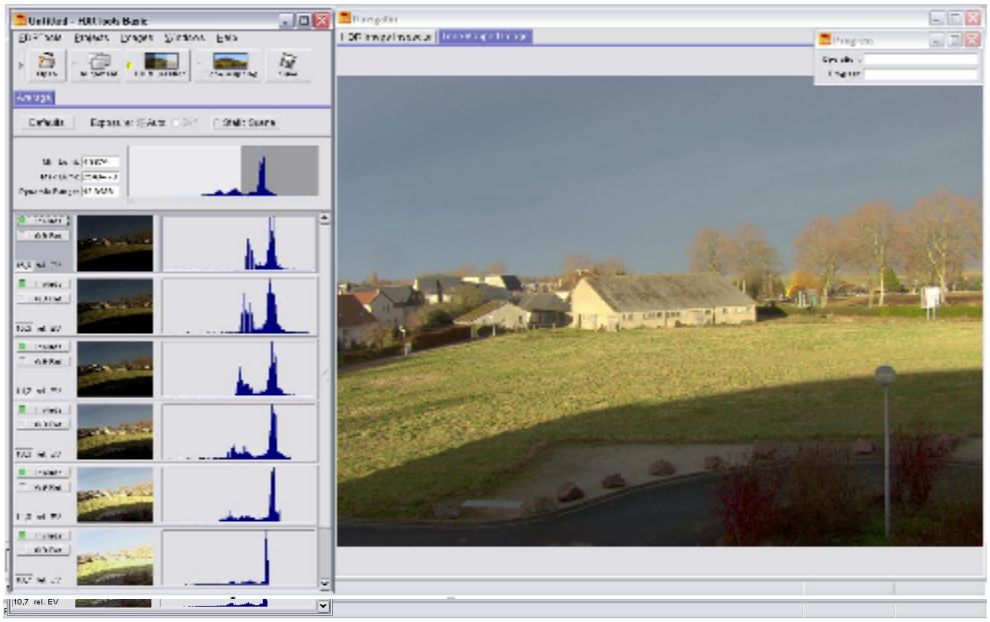


Fig 1.1 FDR Tools : création de l'image HDR

Il est à noter que FDR Tools donne le rendu immédiat de l'image Tonemappée, terme que l'on expliquera par la suite. Ceci peut être un désavantage.

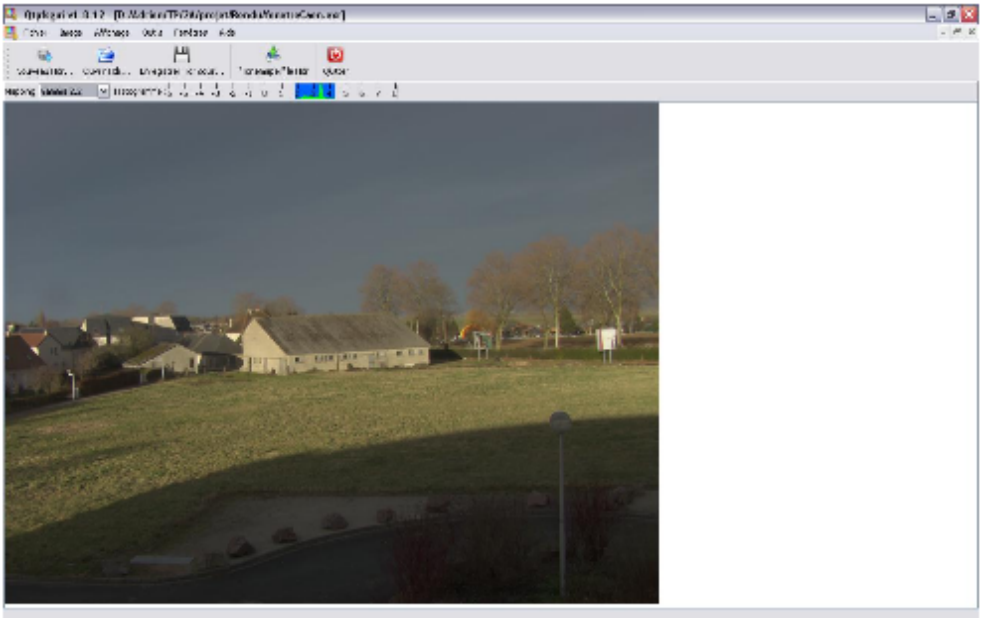


Fig 1.2 Qtpfsgui: création de l'image HDR

Dans les deux cas, on pourra dès ce moment régler la portion d'histogramme que l'image HDR doit prendre en compte: avec FDR Tools, cela consiste principalement à retirer les photos trop claires ou trop sombres, avec Qtpfsgui, on déplacera la réglette bleue au-dessus de l'image:

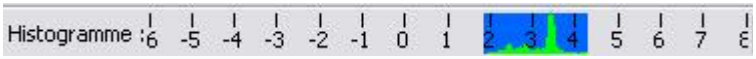


Fig 1.3 Utilisation de la réglette pour déterminer l
a portion d'histogramme à conserver.

Ce réglage permet ainsi d'éliminer les zones trop claires ou trop sombres, mais il faut faire attention à ne pas en abuser: si on restreint trop la zone à conserver, toutes ces manipulations reviendront à prendre une simple photo sans aucun post-traitement numérique tel que le HDRI. Il vaut donc mieux conserver l'ensemble de la gamme dynamique, afin de garder le maximum d'informations possible.

A partir de cette étape, il est possible d'enregistrer l'image HDR dans le format qui lui est propre, OpenEXR. Le fichier obtenu permettra l'utilisation de l'image HDR dans d'autres applications, comme le cube mapping, la création de Light Probe…

Mais si on veut récupérer une photo contenant tous les contrastes de la Nature, il faut procéder au Tone mapping de l'image HDR. Ceci consiste à transformer le codage de l'image sur 16 ou 32 bits en un codage approprié aux écrans travaillant sur 8 bits. Il faut donc utiliser des algorithmes rétablissant l'éventail des contrastes sur la basse gamme dynamique.

En utilisant nos deux logiciels, on peut obtenir des résultats très différents: FDR Tools ne nous autorisera que peu de réglages sur ces transformations, tandis que Qtpfsgui permet de considérer huit algorithmes, travaillant sur la lumière, la saturation des couleurs, la transparence…

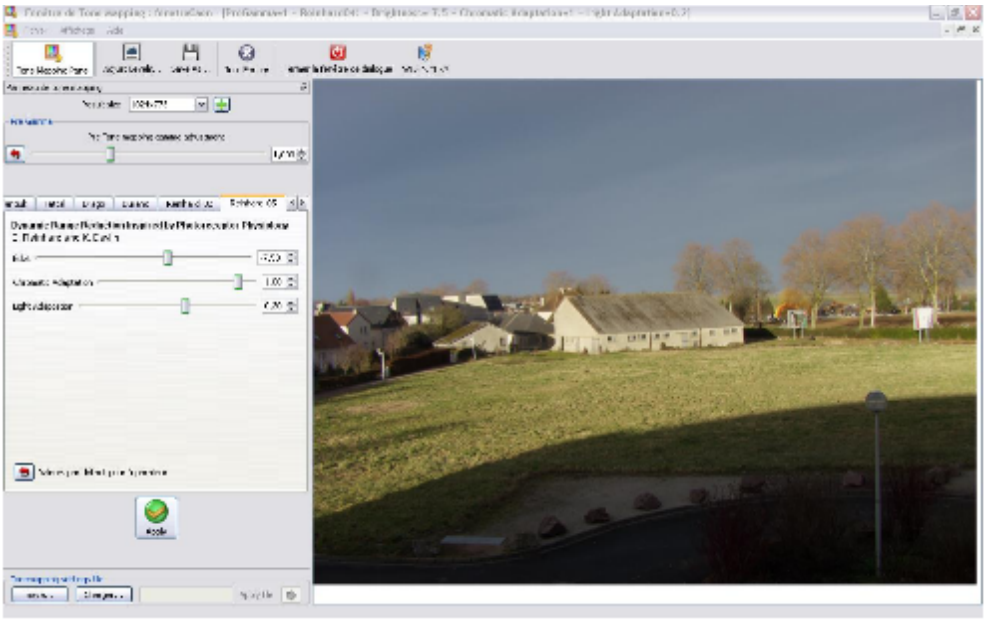


Fig 1.4 Tone mapping sous Qtpfsgui : application de l'algorithme de Reinhard

Mais cette étape ne nécessite plus que le goût de l'utilisateur. Ainsi, à partir des photos suivantes:

Fig 1.5 diverses prises avec un temps de pose croissant.

On a créé ceci:

Fig 1.6 Utilisation de Qtpfsgui, algorithme de Reinhard '05

Fig 1.7 Utilisation de FDR Tools

L’utilisation de l'image HDR et son tone mapping a ainsi éclairci les zones sous-exposées et assombri les zones surexposées; l’équilibre entre les contrastes est rétabli.

2. Stockage d'une image HDR

OpenEXR est un format open source permettant de stocker une image High Dynamic Range. Il a été conçu par Industrial Light & Magic.

2. Format de fichier

Le format OpenEXR permet de stocker des images 48 ou 96 bits, avec respectivement 16 et 32 bits codant chaque couleur rouge, vert et bleu. Les pixels peuvent être stockés par scanlines ou tiles. Si le fichier est en mode tiles, il est possible de stocker plusieurs version d'une même image, toutes avec une résolution différente, ce qui permet d'accélérer des opérations comme le zoom sur un gros fichier ou les opérations de filtrage dans le cas d'un moteur de rendu 3D.

OpenEXR permet aussi de stocker des informations additionnelles sur le fichier, dont certaines sont optionnelles. On peut tout à fait, lors de l'implémentation dans une application, ignorer certaines informations contenues dans le fichier. Néanmoins, certains champs du header sont obligatoires (par exemple, les champs displayWindow, dataWindow ou encore pixelAspectRatio) pour assurer la compatibilité des fichiers OpenEXR générés avec d'autres applications.

2. Lecture d'une image au format OpenEXR

ILM fournit une librairie C++, Ilmlmf, permettant d'écrire ou de lire des fichiers au format OpenEXR. La classe `RgbaInputFile` permet de lire un fichier en scanlines, où chaque couleur du pixel est codée sur 16bits. Dans un tel fichier, il y a autant de scanlines que de pixels en hauteur dans la data window du fichier, et chaque scanline a une longueur en pixels égale à celle de la data window du fichier. Ensuite, tous les pixels sont stockés dans un `Array2D`, modèle de classe fournit dans la librairie d'ILM. Afin de lire notre fichier, on utilisera un `Array2D` paramétré par la structure `Rgba`, permettant de stocker les informations d'un pixel, fournie aussi dans la librairie:

```
struct Rgba{
      half r; // red
      half g; // green
      half b; // blue
      half a; // alpha (opacity)
};
```

Array2D possède une méthode `resizeErase(int height, int width)` se chargeant de l'allocation. Et une fois ce tableau alloué, on peut lire l'image. L'appel de la méthode `setFrameBuffer(Rgba base, int xStride, int yStride)` de `RgbaInputFile` permet d'indiquer la façon d'accéder aux pixels dans le buffer. Par exemple, pour notre image stockée en scanlines, on donnera comme base le premier pixel de l'image, comme xStride 1 (la hauteur d'une scanline) et comme yStride width (la largeur d'une scanline). Ainsi pour accéder au pixel (x,y) la `RgbaInputFile` ira voir à l'adresse base + 1*x + width*y.

Enfin, la lectre effective du fichier se fait par l'appel de la méthode `readPixels(int y1, int y2)` qui lira toutes les scanlines de coordonnée y comprise entre y1 et y2 pour les écrire dans notre `Array2D`. Il est également plus rapide de lire les scanlines dans l'ordre où elles ont été écrites. Pour cela, on dispose d'une methode `lineOrder()` renvoyant `INCREASING_Y` ou `DECREASING_Y` suivant que les scanlines ont été écrites en y croissant ou décroissant.

Pour donner un exemple concret d'une fonction permettant de lire un tel fichier, ILM fournit également quelques codes source en C++. Celui permettant de lire le type de fichier OpenExr qui nous intéresse se présente comme suit:

```
void readRgba (const char fileName[], Array2D<Rgba> &pixels, int
&width, int &height){
      RgbaInputFile file (fileName);
      Box2i dw = file.dataWindow();
      width = dw.max.x - dw.min.x + 1;
      height = dw.max.y - dw.min.y + 1;
      pixels.resizeErase (height, width);
      file.setFrameBuffer (&pixels[0][0] - dw.min.x - dw.min.y *
      width, 1, width);
      file.readPixels (dw.min.y, dw.max.y);
}
```

On note également l'utilisation de la classe `Box2i` qui permet de représenter des `dataWindow` ou des `displayWindow`, disposant de méthodes pour connaître les coordonnées des points extrêmes de la fenêtre.

III. Implémentation du Rendu HDR

***Attention** : Dans cette partie, on considère que le lecteur possède une bonne connaissance de OpenGL des Frame Buffer Object (FBO) et des shaders GLSL. Le cas échéant on trouvera en annexe des liens vers des cours sur ces sujets.*

1. Le contrôle de l'exposition

Comme on l'a vu plus haut dans le projet, une image HDR contient beaucoup trop d'informations pour pouvoir être affichées directement à l'écran ; l'enregistrement des contrastes élevés sur 16 ou 32 bits ne convient pas aux écrans. L'utilisation du tonemapping est donc nécessaire. C'est ce que nous avons fait dans un des shader de notre code. Le tonemapping permet de réajuster l'enregistrement des couleurs sur 8 bits, et l'algorithme présenté en première partie peut donner un bon résultat.

Ainsi, dans le shader tonemapping.frag (voir en annexe), on a ces trois lignes :

```
vec4 avColor= texture2D(Iav, vec2(0,0));
vec4 expo=0.5/(avColor+delta);
gl_FragColor= expo * texture2D(tex, gl_TexCoord[0].st);
```

On récupère le pixel de la texture Iav, d'intensité moyenne. Puis on applique la formule simple de tonemapping vue en première partie sur le pixel en cours de traitement :

$$couleur\_LDR = couleur\_HDR * exposition$$

avec :

$$exposition = 1/(2*Imoy)$$

Imoy étant l'intensité lumineuse moyenne de l'image.

Ce rendu permet de contraster l'image suffisamment, mais il subsiste quelques problèmes avec cette méthode qui est assez brutale.

En effet, cette méthode est statique, c'est-à-dire qu'elle s'adapte pas à l'intensité moyenne de l'image ; celle-ci est mémorisée puis utilisée directement sans être analysée. Il en résulte donc, lors de l'animation que nous avons faite, des moments très sombres, ou d'autres particulièrement clairs.

En utilisant par exemple l'image HDR kitchen_diffuse.exr, la luminosité de la verrière, lorsque celle-ci passe derrière la théière, est beaucoup trop intense. On obtient un débordement beaucoup trop important sur les polygones de notre objet, qui disparaît ainsi presque entièrement sous une vague conséquente de blanc.

Nous avons donc cherché à atténuer cet effet, afin de rendre le rendu plus réaliste. Une méthode consiste donc à rendre l'exposition dynamique vis-à-vis de l'intensité moyenne de l'image. Nous avons alors cherché une formule jouant sur le facteur d'exposition en fonction de l'intensité moyenne. Cependant, aucune d'entre elles ne s'est révélée efficace.

On a également cherché à ajuster l'exposition en fonction de l'intensité maximum de l'image, mais nous n'avons pas réussi à implémenter ce système : cela demande en effet beaucoup de ressources système, car il faut parcourir tous les pixels pour en trouver l'intensité maximale. Au final, nous avons donc conservé cette première formule, certes un peu brutale, mais tout de même efficace.

2. L'implémentation de l'effet Bloom

L'implémentation de l'effet Bloom consiste à réaliser un rendu de la scène dans un buffer auxiliaire de taille $2^n \times 2^p$ que l'on va flouter à l'aide d'un algorithme de flou implémenté dans un shader. Le principe de l'algorithme consiste à copier le buffer auxiliaire dans un autre buffer. Ce dernier est alors plus ou moins flouté suivant la quantité de Bloom demandée.

Mais le fait de flouter une image prend énormément de temps et ceci doit être fait à chaque frame. Il existe des quantités d'algorithmes de flou d'image comme le flou Gaussien quoi doit être le plus connu. Pour notre démo nous avons retenu le flou directionnel que l'on applique verticalement et horizontalement. Celui ci est plus rapide que le flou Gaussien mais de moins bonne qualité. Pourtant en prenant des noyaux assez grand (de demi-taille 16 par exemple), l'algorithme de flou directionnel est déjà très lent implémenté dans un shader et la constitution des shaders GLSL ne nous permettent pas de l'optimiser.

Il existe une autre technique pour appliquer un gros flou sur une image en temps réel. Celle ci consiste à utiliser le filtrage linéaire des textures offert par la carte graphique. Nous savons bien que lorsque l'on observe une texture à l'écran avec OpenGL dont la taille est plus grande que sa taille d'origine, le filtrage linéaire à tendance à flouter justement cette texture pour supprimer l'effet désagréable de la pixellisation.

Cette propriété va servir pour effectuer de très gros flous en un minimum de temps car géré en hardware par la carte graphique qui est optimisée pour ce genre de calculs au contraire du cpu. Cela consiste à flouter légèrement le buffer de base grâce à un flou directionnel. Ensuite on copie ce buffer, dans un autre de taille moitié moins grande que le buffer d'origine. Ce dernier est flouté légèrement avec le flou directionnel, puis il est redimensionné à la taille d'origine du buffer et est finalement ajouté à ce dernier avec un facteur d'alpha donné (pour ne pas remplacer complètement le buffer d'origine. On répète cette opération jusqu'à le buffer ait une taille unitaire. Voici un diagramme synthétisant les différentes étapes de la réalisation du Bloom

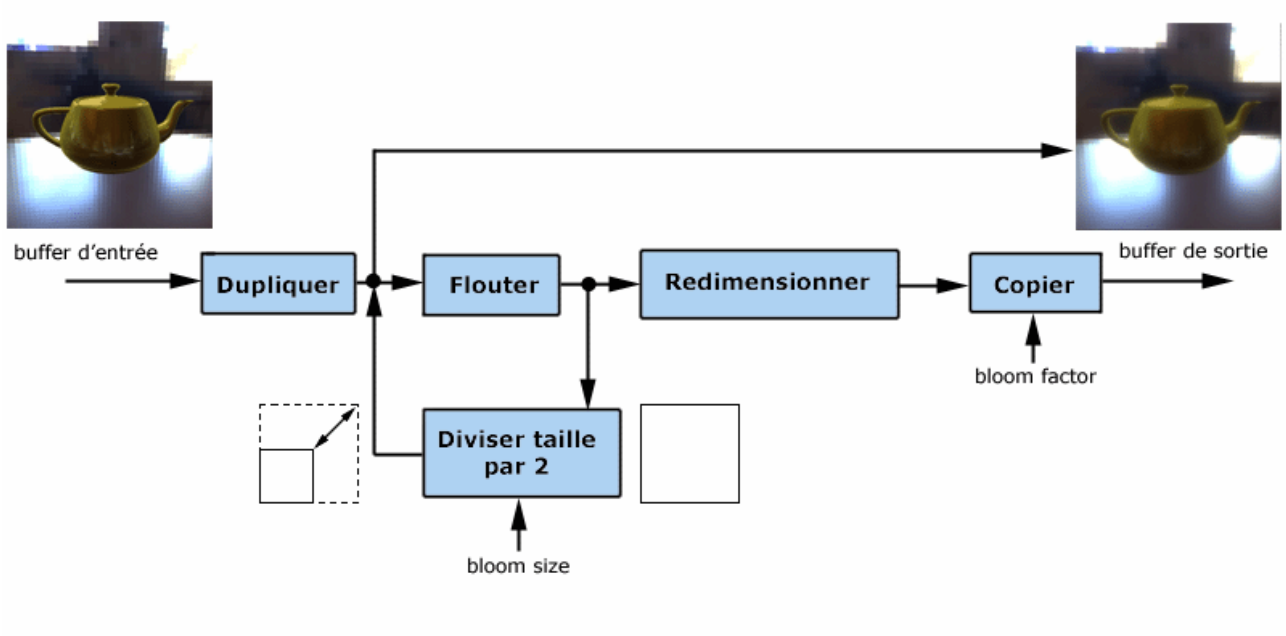


Fig 2.1 Les étapes de la réalisation de l'effet Bloom

Il n'est pas nécessaire que la taille du buffer auxiliaire soit aussi grande que le frame buffer. En effet, l'effet de flou final efface tous les détails du frame buffer, il n'est pas donc utile d'avoir un buffer en haute résolution, le calcul du flou étant un calcul lourd qui doit être effectué le plus rapidement possible, à chaque frame.

3. Présentation de la démo du rendu HDR

Après avoir étudié de manière théorique le rendu HDR nous avons décidé d'illustrer nos propos en réalisant une démonstration de ce type de rendu.

Voici un aperçu des ses fonctionnalités:

- Chargement d'images au format OpenEXR 48bits vers une texture OpenGL au format flottant 16 bits.
- Chargement d'images au format TGA et JPG vers une texture OpenGL.
- Chargement d'images de cube map disposés en une croix verticale aux formats JPG, TGA, et OpenEXR.
- Illumination globale à partir d'images HDR ou LDR (IBL: Image Based Lighting) à partir de cube maps de réflexion spéculaire et diffuse précalculées suivant l'environnement désiré.
- Production d'effet Bloom sur images LDR et HDR.
- Tone mapping avec gestion automatique de l'expos ition.
- Gestion dynamique des effets visuels en production et post-production. Possibilité d'ajouter de nouveaux effets au moteur de rendu existant dynamiquement. Le support des shaders GLSL est nécessaire pour tous les effets.
- Rendu dans une texture. Le support des framebuffer objects (FBO) est nécessaire.
- Gestion de la caméra.

Evolutions futures:

- Chargement d'objets 3D dans divers formats.
- Support des Vertex Buffer Objects (VBO).
- Support du rendu dans un format quelconque (Actuellement, limité au rendu dans un buffer de taille $2^n \times 2^p$).

4. Difficultés rencontrées

Au cours de ce deuxième semestre, notre but était d'implémenter un programmer d'utilisation des images HDR sur un objet dans une scène simple. Après avoir cherché de la documentation sur l'utilisation des shaders et de leurs implémentation en OpenGL, Stéphane a tapé la majorité du code, ce qui lui a été plus aisé qu'à Adrien et Alexis, ce premier ayant déjà une connaissance accrue d'OpenGL.

Lors de la production de ce code, plusieurs difficultés ont été rencontrées : l'utilisation des shaders n'est pas simple, et la production d'un code clair, manipulable et réutilisable n'est pas facile.

Cependant, nous sommes arrivés à un résultat largement satisfaisant pour la démonstration que nous voulions avoir.

Mais lors des essais, un autre point nous a ralenti : la carte graphique de Stéphane est une ATI, qui ne gère pas le filtrage linéaire des textures au format 16 bits flottants. Ainsi, le placage de l'image HDR sur la skybox dans notre scène apparaît très grossière, pixellisée. Cela n'empêche toutefois pas l'algorithme codé d'afficher une théière comportant le reflet de son environnement, et ce, avec une pixellisation moindre ; mais il faut toutefois souligner le fait que le codage de l'effet bloom a dû ainsi être fait avec une texture LDR, pour réaliser l'effet de flou à l'aide du filtrage linéaire de la carte graphique.

Fig 4.1 A gauche: rendu avec filtrage linéaire sur geforce 7600
A droite rendu sans filtrage linéaire sur Radeon mobility X1600

Nous avons alors fait des essais avec l'ordinateur d'Adrien, mais un autre problème s'est posé: il nous fut dans un premier temps totalement impossible d'installer la librairie IlmImf sur son ordinateur. Le problème n'a été résolu qu'au bout de plusieurs semaines de recherches et d'essais infructueux. Finalement, l'utilisation d'une version plus ancienne (version 4 au lieu de version 6) et du bidouillage avec le dossier d'installation des librairies (/usr/lib au lieu de /usr/local/lib) nous a permis de faire des tests sur un ordinateur comportant une carte nVidia, gérant, elle, le filtrage linéaire.

Cependant, nous obtenons des warnings obtenus non pas à la compilation, mais à l'exécution du programme :

```
Vendor: NVIDIA Corporation
Renderer: GeForce Go 7600/PCI/SSE2
Version: 2.1.1 NVIDIA 100.14.19

Ready for GLSL
Shading language version supported: 1.20 NVIDIA via Cg compiler

(17) : warning C7532: global variable gl_TextureMatrixInverse requires "#version
110" before use
Fragment info
-------------
(17) : warning C7532: global variable gl_TextureMatrixInverse requires "#version
110" before use
(21) : warning C7011: implicit cast from "int" to "cfloat"
(21) : warning C7011: implicit cast from "int" to "cfloat"
(27) : warning C7011: implicit cast from "int" to "float"
(34) : warning C7011: implicit cast from "int" to "float"
(38) : warning C7011: implicit cast from "int" to "float"
(12) : warning C7532: global function exp requires "#version 110" before use
Fragment info
-------------
(21) : warning C7011: implicit cast from "int" to "cfloat"
(21) : warning C7011: implicit cast from "int" to "cfloat"
(27) : warning C7011: implicit cast from "int" to "float"
(34) : warning C7011: implicit cast from "int" to "float"
(38) : warning C7011: implicit cast from "int" to "float"
(12) : warning C7532: global function exp requires "#version 110" before use
```

Ces warnings n'apparaissent pas sur l'ordinateur de Stéphane, ce qui nous laissent supposer que ATI et nVidia ne gèrent pas de la même manière les flottants. Nous n'avons pas eu suffisamment de temps pour résoudre ce problème, mais nous pensons que cela provient du fait que les shaders sont compilés via le compilateur Cg propre à nVidia.

Conclusion

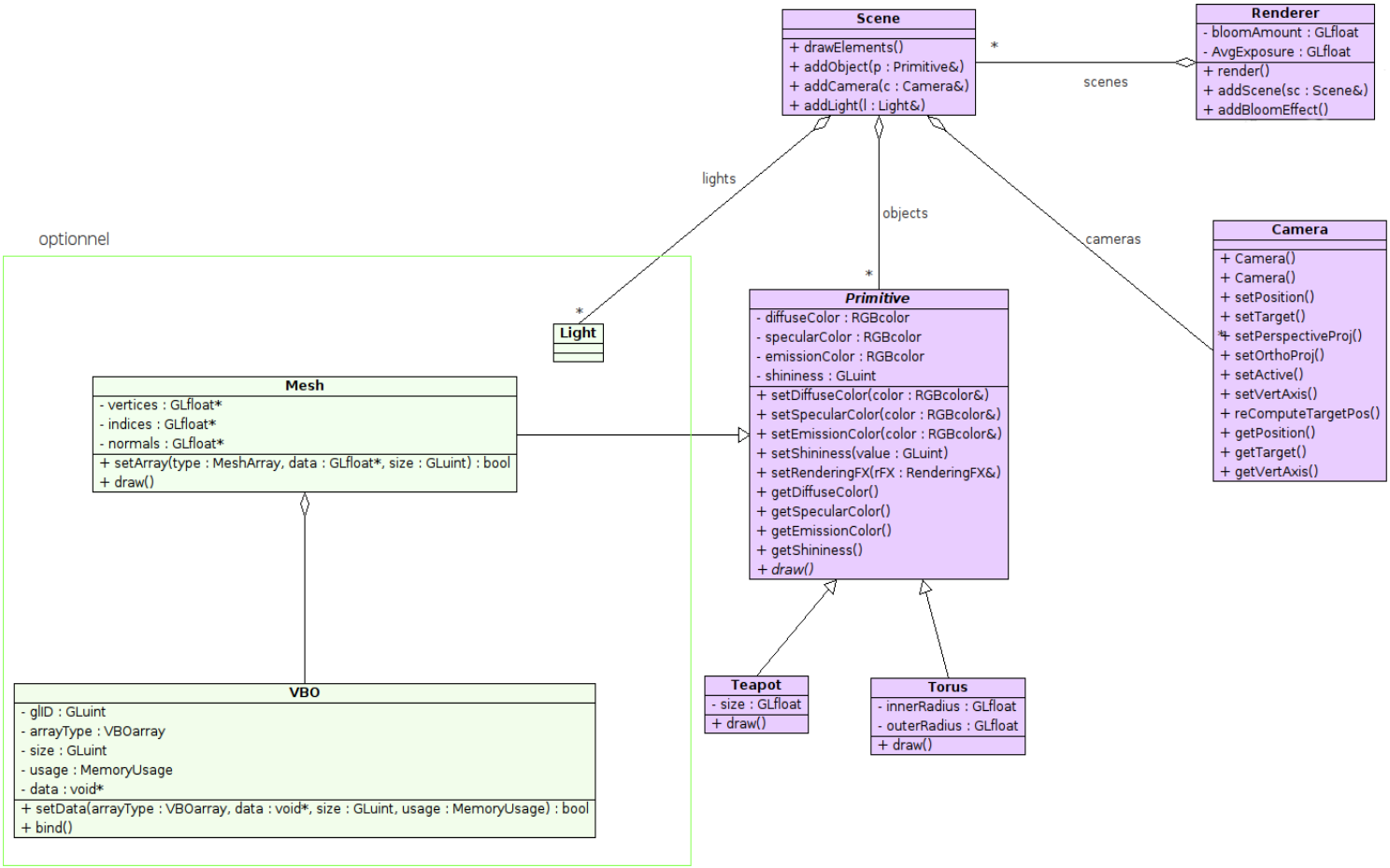
Le rendu HDR est de plus en plus utilisé dans les jeux vidéo ou dans la simple photographie. Grâce à l'amélioration des performances des cartes graphiques et aux innovations technologiques tels que les shaders, les calculs au niveau des cartes graphiques peuvent plus nombreux, afin d'obtenir un rendu photoréaliste. Férus de jeux vidéo, nous avons voulu en savoir un petit peu plus...

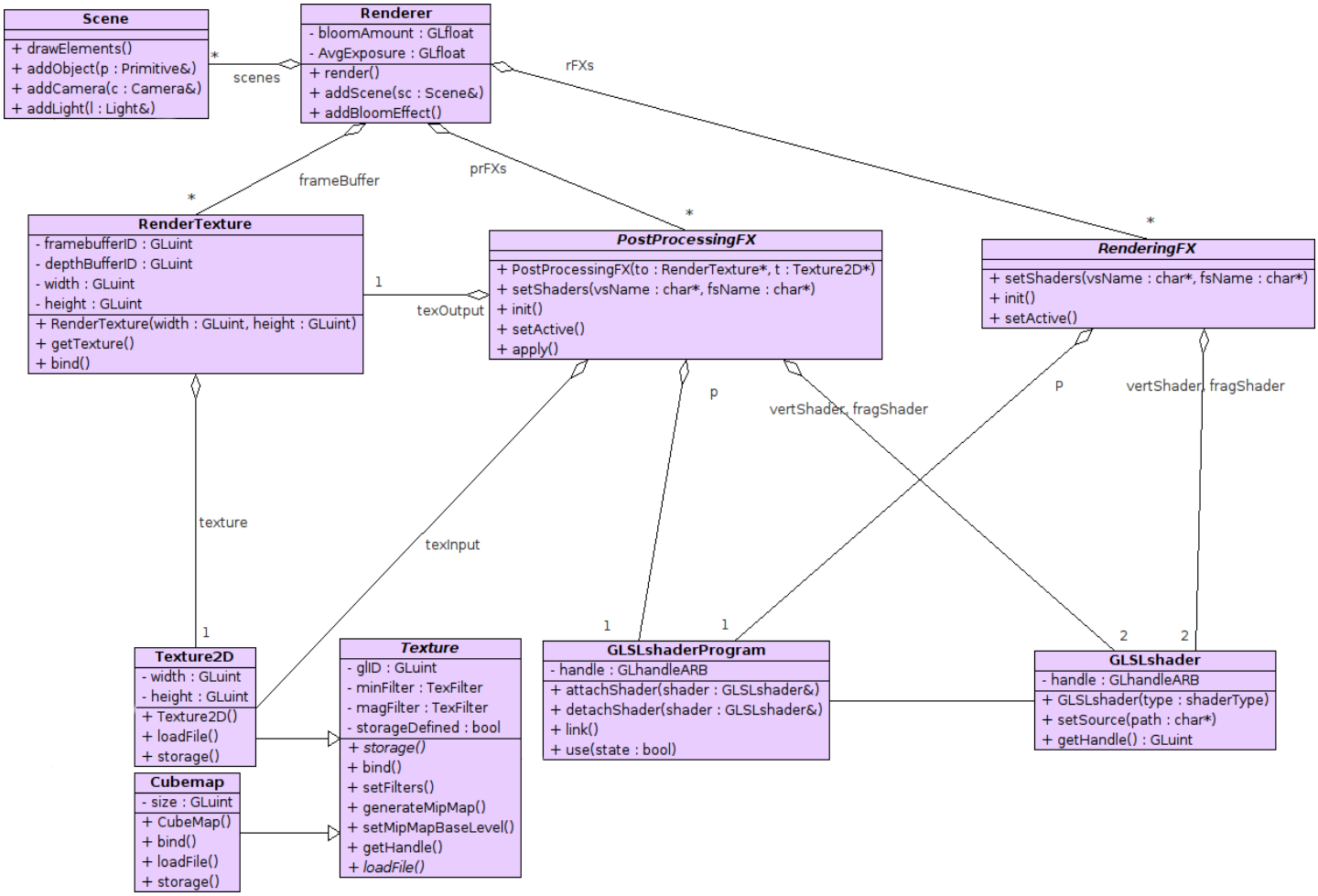
Nous avons ainsi découvert que le HDRR permettait la conservation de toutes les caractéristiques de la luminosité réelle, grâce notamment à un codage sur 16 ou 32 bits flottants par pixel. Nous avons alors réalisé une expérience photographique qui a permis de se rendre compte des possibilités offertes par la technologie HDR. La réalisation d'une démonstration sur une petite scène 3D nous a permis d'illustrer notre étude : stockage et lecture d'une image HDR dans le format OpenEXR, application de l'effet bloom et tonemapping...

Ce projet nous a ainsi permis de nous initier plus avant dans la synthèse d'images, que cela soit dans la partie programmation ou dans la partie gestion de projet, avec tous les problèmes encourus à résoudre.

Annexes

1. Diagramme de classes de la démo





2. Explication du diagramme

Comme indiqué la partie encadrée est optionnelle. Elle décrit les évolutions possibles de la démo qui auraient pu être réalisées si il restait encore du temps après avoir implémenté la première partie.

La classe **`Scene`** est un conteneur décrivant tout ce que peut contenir une scène 3D dans le cadre de la démonstration. Ici elle contient des objets 3D, des lampes et des caméras. Cette classe est amenée à évoluer pour contenir plus d'objets. Elle est associée à la classe **`Renderer`**.

La classe **`Renderer`** quant à elle se charge du rendu d'une scène et est donc associée à un ensemble d'instances de **`Scene`**. De plus elle est associée à la classe **`RenderTexture`** qui gère le rendu vers une texture car le rendu ne doit pas être envoyé directement vers le frame buffer afin de pouvoir appliquer les effets de post-production comme le Bloom et le Tone mapping.

Les classes **`RenderingFX`** et **`PostProcessingFX`** qui sont associées à la classe **`Renderer`** décrivent des modules se chargeant respectivement des effets visuels au rendu (production) et à la post-production. L'intérêt de ces deux classes réside dans le fait que l'on peut aisément ajouter et supprimer des effets visuels au rendu de manière dynamique en pleine

exécution. cela permet aussi de créer des nouveaux effets qui s'adapteront facilement au moteur de rendu existant sans toucher au code de ce dernier.

Les classes **GLSLshader** et **GLSLshaderProgram** sont des classes de plus bas niveau et ont pour objectif de gérer facilement les fonctions fournies par OpenGL pour l'implémentation de shaders GLSL.

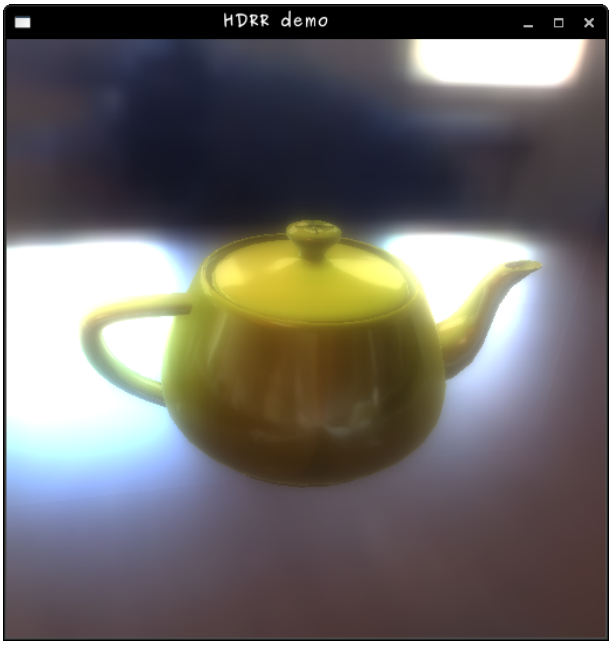
La classe abstraite **Texture** donne une description générale des textures d'opengl et offre des fonctions de chargement de divers formats de textures (JPG, TGA et OpenEXR actuellement). Les classes **Texture2D** et **CubeMap** en héritent pour gérer respectivement l'utilisation de textures 2D et des cube maps sous OpenGL.

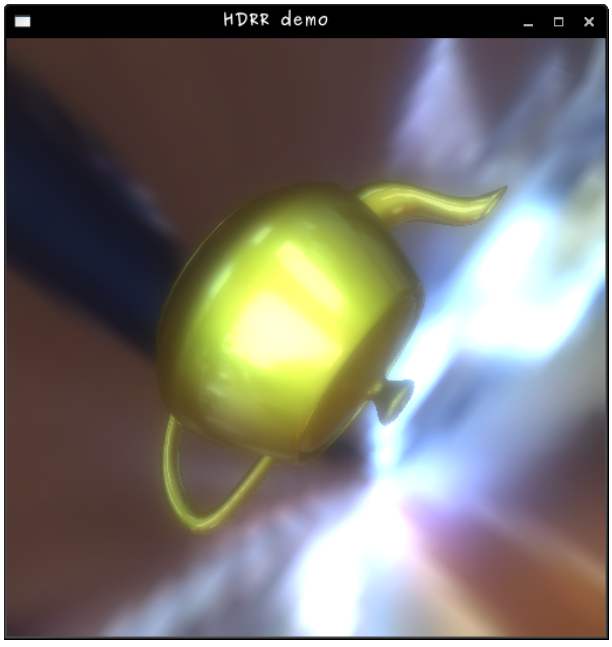
La classe **RenderTexture** permet le rendu dans une texture sous OpenGL grâce à l'utilisation des FBO. Pour le moment cette dernière est limitée au rendu dans des textures au format $2^n \times 2^p$.

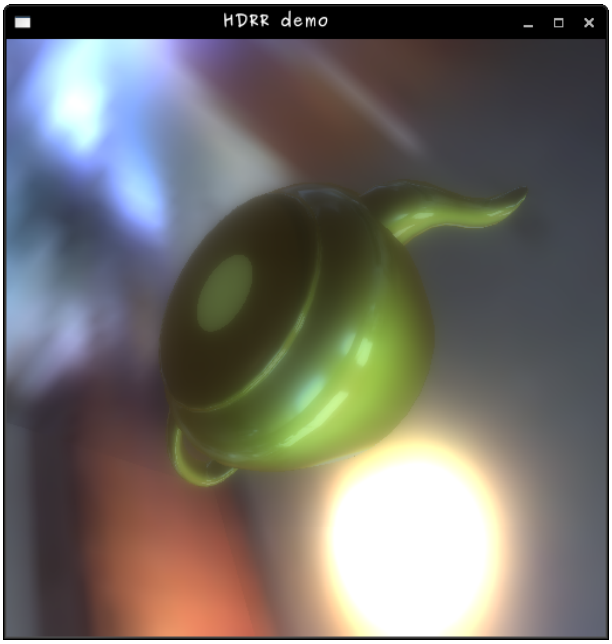
La classe **Primitive** décrit les propriétés de base d'un objet 3D comme ses propriétés matérielles.

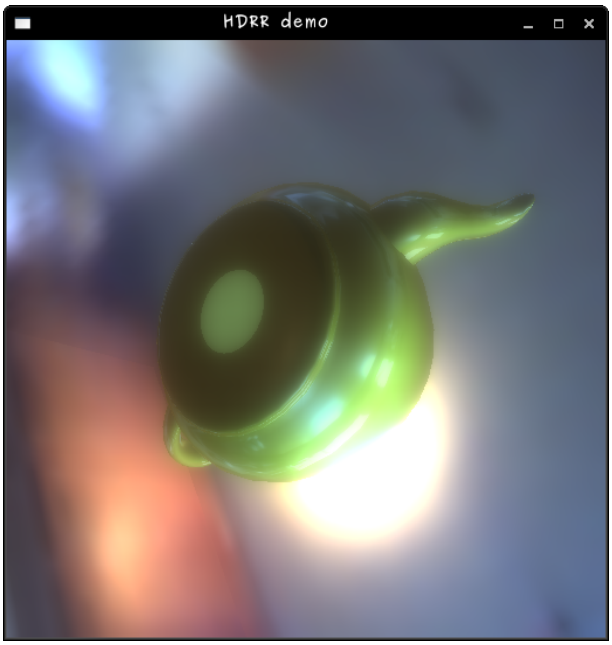
3. Résultats

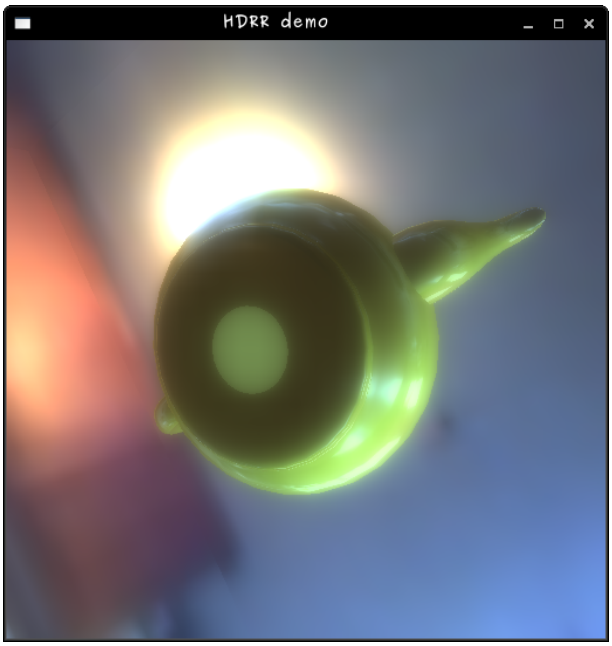
Voici quelques captures d'écran réalisées lors de l'exécution de la démo:





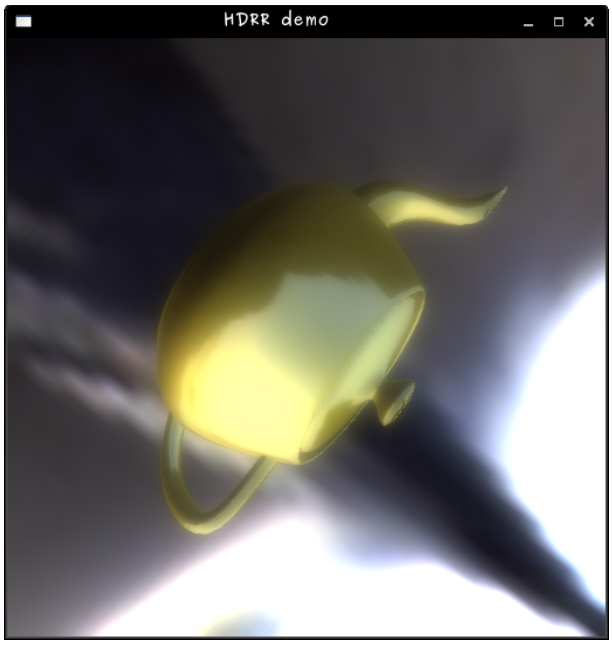

HDRR demo

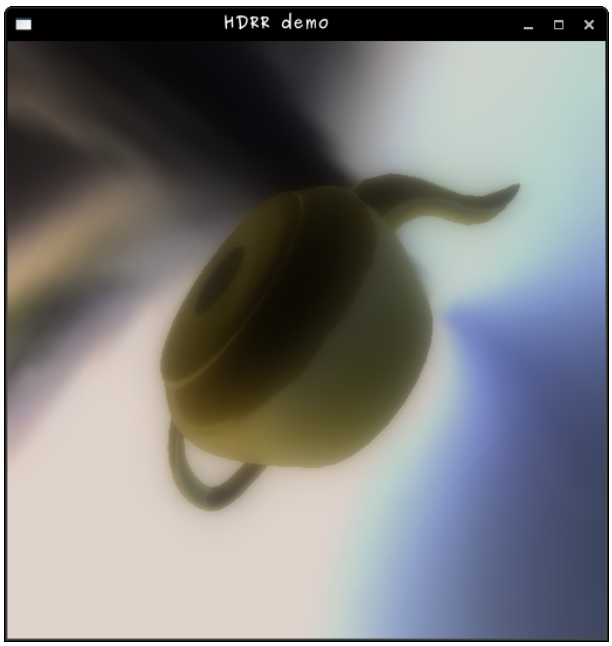

HDRR demo

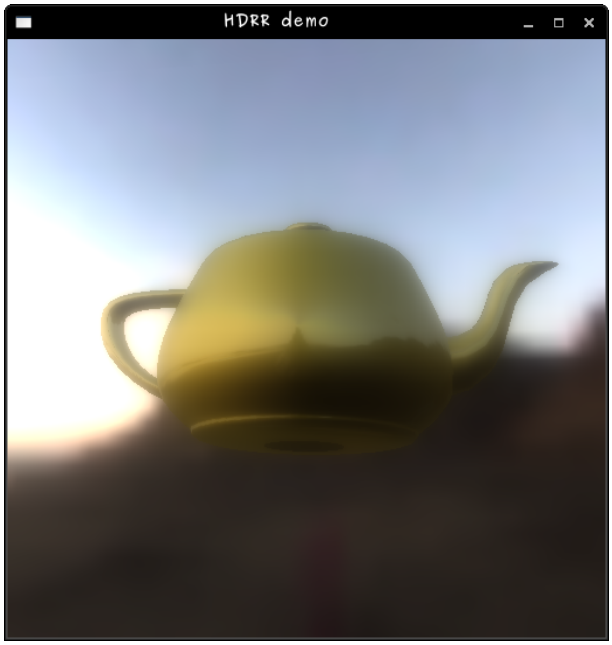

HDRR demo


HDRR demo


HDRR demo


HDRR demo


HDRR demo


HDRR demo

4. Références

[1] Site du logiciel Qtpfsgui avec sa documentation
http://www.mpi-inf.mpg.de/resources/pfstools/

[2] Qtpfsqui sur sourceforge
http://qtpfsgui.sourceforge.net/

[3] Tutoriel sur la photographie HDR avec une démonstration de l'utilisation de Qtpfsgui
http://bellette.tuxfamily.org/pixelpost/index.php?x=page&title=hdr1intro

[4] Le site de Paul Debevec: Ses travaux sont nombreux, notamment sur le rendu HDR
http://www.debevec.org/

[5] Le HDRR sur wikipedia
http://en.wikipedia.org/wiki/High_dynamic_range_rendering

[6] Papier sur la création de diffuse et specular cube maps
http://www.debevec.org/ReflectionMapping/IlluMAP84.html

[7] Une présentation par ATI sur l'illumination par image
http://ati.amd.com/developer/gdc/GDC2005_PracticalPRT.pdf

[8] Le site officiel de OpenGL et son forum très utile
http://www.opengl.org/
http://www.opengl.org/discussion_boards/

[9] Une bonne synthèse des FBO
http://www.gamedev.net/reference/articles/article2331.asp
http://www.gamedev.net/reference/articles/article2333.asp

[10] Un bon cours sur les shaders GLSL en anglais
http://www.lighthouse3d.com/opengl/glsl/

[11] Un cours très complet sur les shaders GLSL en anglais
http://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/