UNIVERSITE DE REIMS

Deferred Rendering

Travail d'étude et de recherche

Paul Demeulenaere

22/06/2009

Sommaire

I. Introduction

Le projet d'étude et de recherche concerne la méthode du deferred rendering. En programmation graphique pour le temps réel, le deferred rendering est une approche visant à réduire la complexité d'une scène en découpant les étapes du rendu et en réduisant l'exécution des algorithmes à ce qui est visible.

Le deferred rendering s'oppose à la méthode de rendu classique parfois appelée foward rendering. Le deferred rendering n'est pas un concept récent, son implémentation l'est plus. En effet, le développement des cartes graphiques récentes a tendance à favoriser le deferred rendering. Par ailleurs, dans un contexte où les applications vidéo-ludiques sont, en terme de shader, de plus en plus complexes et où elles impliquent souvent une importante équipe de développeurs, le deferred rendering peut être une bonne solution pour assurer une application sûre, d'un design clair et de développement aisé.

Ce rapport s'articulera en quatre grandes parties. Tout d'abord, le rendu déferré sera présenté de manière générale, ses avantages, ses inconvénients mais aussi la façon de l'implémenter en profitant des fonctionnalités des cartes graphiques de nouvelle génération. La partie suivante traitera des implémentations particulières aidées ou contraintes par le deferred rendering. Ensuite, quelques exemples de rendu post-traitement sont exposés. Enfin, une dernière partie traitera des variantes sur certaines parties du deferred rendering et sur le concept même du rendu déferré.

II. Deferred Rendering

1. Présentation générale

Résumé

Le « deferred rendering » est une technique de rendu temps réel s'opposant clairement au « foward rendering ». Le foward rendering est la méthode classique pour rendre des maillages texturés et éclairés : pour chaque objet, les triangles sont rendus, pour chaque pixel de ces triangles, la contribution de chaque lumière est sommée puis combinée au matériel de l'objet courant.

Le deferred rendering est une approche différente. Définissons tout d'abord un élément essentiel : le Geometry-Buffer[1] (ou G-Buffer) est une texture contenant un certain nombre d'informations pour chaque pixel, le plus communément, la position, la normale et la couleur. La technique du deferred rendering peut être présentée en deux passes, la première concerne le rendu du G-Buffer, la seconde utilise ce dernier pour ajouter l'éclairage à chaque pixel du rendu final.

Origines

L'idée du deferred rendering a été introduite par Michael Deering dans une recherche publié en 1988 appelé: "The triangle processor and normal vector shader : a VLSI system for high performance graphics." Même si le terme de « deferred » n'y est jamais utilisé, l'idée était bien là. C'est seulement depuis quelques années que cette technique devient populaire, notamment dans les jeux de nouvelle génération où la complexité des scènes en vertex et éclairement est importante : S.T.AL.K.E.R, Tabula Rasa, Dead Space, Killzone 2 ou le Cry Engine 3[2].

Avantages et Inconvénients

Quand le sujet du deferred rendering est traité, les avantages les plus souvent évoqués concernent la complexité de l'illumination proportionnelle à l'espace couvert sur l'écran, ou la gestion de scène simplifiée. En revanche, le fait que le nombre de batches peut grandement être réduit, notamment lors de projections de shadow map, est souvent oublié.

En effet, la simplicité des éléments en entrée du G-Buffer permet de compiler au maximum les données et donc limiter le nombre de batches, d'autant plus avec DirectX 10 où il est possible d'assigner plusieurs buffers en un seul batch. Les passes suivantes sont des

[1] Le G-Buffer peut être assimilé à ce qui est souvent appelé « Render Element » dans les logiciels de modélisation et de rendu, c'est un fichier contenant des informations complémentaires sur l'image en sortie, il est ensuite utilisé par les logiciels de post-production tels que « Autodesk Combustion ».

[2] Le Cry Engine 3 est basé sur la variante « Light Pre-Pass Rendering » du deferred rendering.

rendus de quad ou autres maillages simples (pour délimiter les lumières), il ne peut y avoir de calcul inutile car masqué par un autre objet par la suite. La plupart des applications temps réel sont limitées par le CPU. Limiter le nombre de batches limite l'activité du CPU, ce qui en plus d'améliorer les performances permet d'améliorer les parties prises en charge uniquement sur le processeur central de traitement comme l'intelligence artificielle.

Néanmoins, il est vrai que l'illumination est très efficace en deferred rendering, surtout lors que la scène est composée de nombreuses petites lumières. En effet, l'éclairage est réalisé sur une texture à deux dimensions représentant la géométrie dans l'espace écran. Pour chaque lumière, il est simple de limiter le nombre de pixels où les équations d'illumination seront appliquées.

Par ailleurs, le deferred rendering a l'avantage de fournir beaucoup d'informations pour chaque pixel, des informations pouvant être complémentaires. Par exemple, la vitesse du pixel peut être stockée de manière à réaliser du flou de mouvement[3]. De la même manière, travailler dans l'espace écran est indispensable pour certaines techniques comme l'occlusion ambiante dans l'espace écran. Aussi, rendre dans une texture est essentiel pour d'autres techniques telles que l'imagerie à grande dynamique[4] ou le flou de profondeur[5]. C'est souvent une des raisons pour lesquelles le deferred rendering est choisi dans des applications vidéo ludiques où la plupart de ces effets sont des standards.

D'autre part, le deferred rendering souffre de certaines difficultés. La première est l'anticrénelage. En effet, de base, le deferred rendering est incompatible avec l'anti-aliasing. De même, la transparence est un problème, le G-Buffer ne contient qu'une seule information par pixel. Néanmoins, certaines solutions sont possibles, elles seront présentées un peu plus loin.

2. Schéma explicatif

Un schéma récapitulatif du principe du deferred rendering est présenté ci-dessous. Il s'agit d'un schéma simple permettant de visualiser le fonctionnement général, il ne peut en aucune cause être une représentation exhaustive. On constate clairement que les maillages ne sont envoyés qu'une seule fois à la carte graphique lors du rendu du G-Buffer. C'est seulement à partir de ce dernier que l'illumination est calculée. Les parties de rendu du G-Buffer et de l'accumulation du light buffer sont détaillées plus loin.

[3] Motion blur
[4] HDR pour High Dynamic Range
[5] DoF ou Depth on field

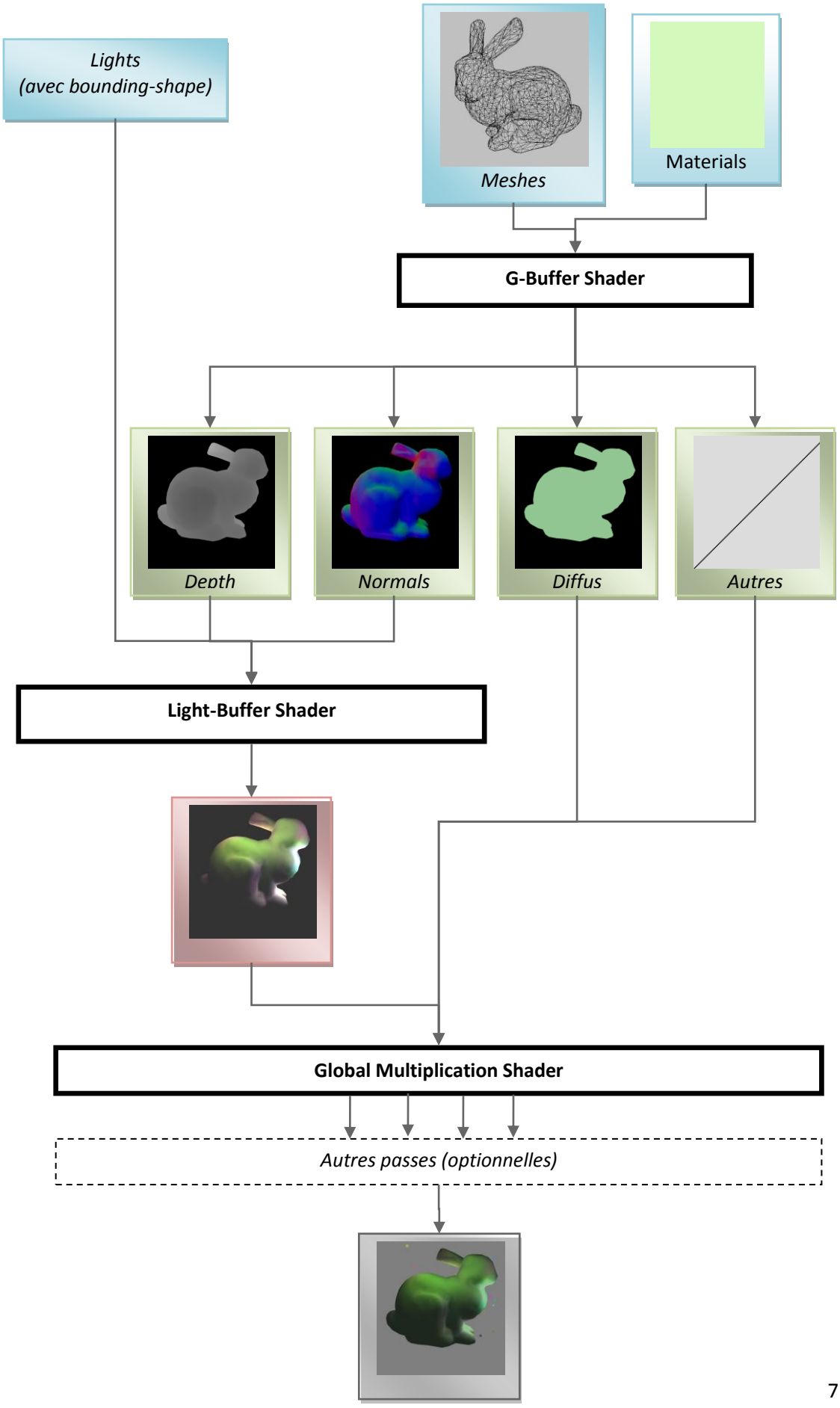


Figure 1 : Schéma explicatif de la méthode du deferred rendering

3. G-Buffer

Le G-Buffer est un élément essentiel du deferred rendering. Le terme a été introduit par Saito et Takahashi en 1990. Il s'agit d'une texture, un ensemble de textures, ou un tableau de textures, où chaque pixel correspond à une série d'informations telles que la normale, la position, la couleur ou encore d'autres informations comme l'intensité spéculaire.

Méthode de stockage

La première question qu'il vient à se poser lors de l'utilisation d'un G-Buffer est la méthode de stockage à utiliser. Rappelons que les informations pour chaque pixel sont stockées dans des textures. Beaucoup de formats sont disponibles en DirectX 10, il est possible de rendre en sortie du pixel shader vers plusieurs textures en même temps[6].

La plupart des méthodes de deferred sont présentées avec un tableau de texture, quatre textures de quatre composantes huit bits. DirectX 10 peut supporter jusqu'à huit textures en sortie du pixel shader. Néanmoins, DirectX 9, qui est encore plus généralement utilisé, n'en supporte que quatre. Ci-dessous, vous trouverez un exemple d'organisation des valeurs dans un quadruplet de texture 8bits.

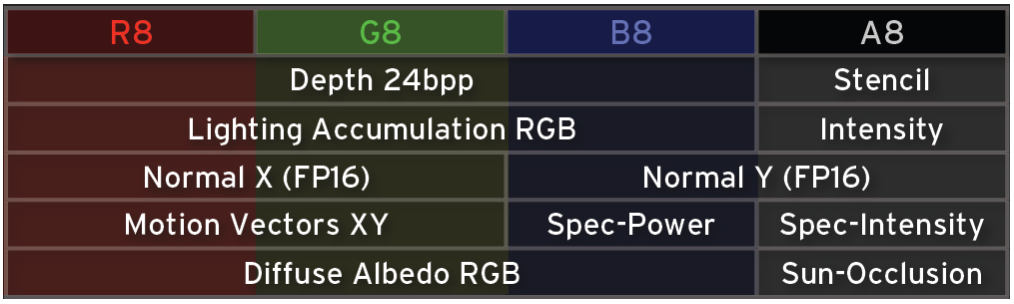


Figure 2 : Répartition des valeurs dans le G-Buffer de Killzone 2

La première approche implémentée fut semblable à cette organisation, mais après plusieurs expérimentations, il a été décidé d'utiliser les nouvelles fonctionnalités de DirectX 10 pour finalement stocker un maximum de données dans une seule texture à quatre composantes 32bits, soit l'équivalent en espace mémoire, de quatre textures à quatre composantes 8bit.

R32	Couleur diffuse (rouge-vert-bleu)		Coef-Speculaire
G32	Profondeur		
B32	Normale X		Normale Y
A32	Vitesse X	Vitesse Y	Normale Z

Figure 3 : Répartition des informations dans une texture 128bits

[6] Appelé communément MRT ou multi-render target

Les informations dans alpha sont optionnelles, leur intérêt est montré plus loin. Pour réaliser cet ordonnancement, les opérations bitwise ont été utilisées. Elles sont apparues en HLSL avec les shader model 4.0 et donc DirectX10. Les opérations bitwise, parfois appelées opérations bit à bit, sont utilisées pour effectuer des transformations et opérations sur des nombres binaires sans prendre en compte le format d'origine. Ces fonctions nous permettent de stocker, par exemple, quatre valeurs comprises entre 0 et 255 dans un flottant de 32 bits[7]. Illustrons le principe par un exemple : deux valeurs A et B respectivement à 0.5 et 0.8 sont à stocker dans un flottant de 32 bits. Première étape, on les passe entre 0 et 2^{16} par simple multiplication. Les nombres binaires sont écrits de manière hexadécimale par souci de clarté.

$$A = 0.5 \; et \; B = 0.8$$

$$A^{'} = 0.5 * 0xFFFF = 0x\mathbf{7FFFF} \; et \; B^{'} = 0.8 * 0xFFFF = 0x\mathbf{CCCC}$$

La seconde étape consiste à ajouter 16 zéros à droite de A', en pratique, cela correspond à utiliser l'opérateur de bit shifting. Ensuite, les 16 zéros à droite de A' sont remplacés par le résultat de B', c'est un « ou logique » qui est utilisé.

$$A^{'} = A^{'} \ll 16 = 0x\mathbf{7FFF}0000$$

$$A^{'} = A^{'} | B' = 0x\mathbf{7FFFCCCC}$$

Le cast-binaire (asfloat) est appliqué sur A' pour finalement stocker le résultat. On retrouve les valeurs d'origine grâce à l'opérateur du « et logique » et du bit shifting.

$$A \cong \frac{(A^{'} \; \& \; 0xFFFF0000) \gg 16}{0xFFFF} \; et \; B \cong \frac{(A^{'} \; \& \; 0x0000FFFF) \gg 0}{0xFFFF}$$

En HLSL, ces techniques correspondent aux deux fonctions suivantes.

```
float pack_float2(float2 f)
{
        uint2 res = uint2(f*65535.0f);
        return asfloat( res.x<<16 | res.y );
}

float2 unpack_float2(float f)
{
        uint unpack = asuint(f);
        float4 res;
        res.x = (float)((unpack & 0xffff0000)>>16);
        res.y = (float)((unpack & 0x0000ffff));

        return res/65535.0f;
}
```

[7] Le langage Cg de NVidia permet d'utiliser des fonctions de packing : pack_2half ou unpack_2half

C’est la même méthode qui est appliquée pour stocker trois ou quatre flottants dans un ; évidement, la précision dépend de l’espace alloué. Pour stocker des valeurs signées, la valeur entre -1 et 1 est incrémentée de un puis divisée par deux, l’opération inverse est réalisée après la décompression des données. Notons qu’il faut veiller à ne jamais réaliser d’interpolation sur les flottants des textures, elles ne seront pas cohérentes.

Question du Z

Il existe deux manières de stocker la position dans l’espace. Les deux méthodes sont basées sur le stockage de la profondeur pour reconstruire la position grâce aux coordonnées du pixel *xy* dans l’espace écran.

La première solution consiste à stocker le couple z et w à chaque pixel. Avec les valeurs de x et de y, il est possible de reconstruire la position dans l’espace projeté dans un premier temps. Puis, par multiplication des matrices inverses vues et projection, reconstruire la position dans le repère monde.

La seconde solution est de stocker la distance avec l’œil divisée par la distance au plan de clipping lointain (far plane) afin d’obtenir une valeur entre 0 et 1. Ensuite, le rayon partant de l’œil passant par le pixel du plan proche est reconstruit pour déterminer avec exactitude la position dans le repère monde. C’est cette seconde méthode qui a été choisie car plus intuitive et permettant l’accès à une profondeur linéaire.

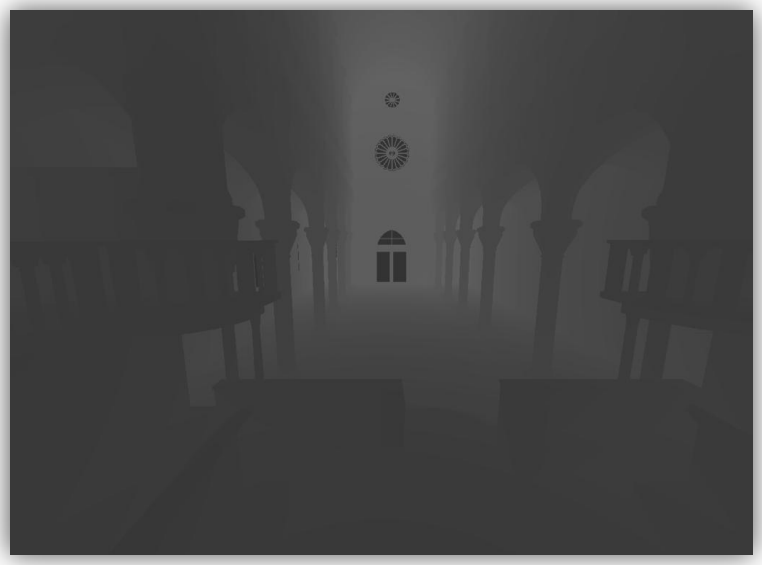


Figure 4 : Profondeur linéaire sur la scène de la cathédrale

Stockage de la normale

Les normales étant normalisées, c'est-à-dire que leur longueur est égale à 1, il est possible de stocker seulement deux composantes pour retrouver la troisième :

$$z = \pm\sqrt{1 - x^2 - y^2}$$

Le problème est de retrouver le signe de cette valeur. Intuitivement, si la normale est stockée dans le repère de l'observateur, il semble que la valeur z soit toujours du même signe puisque les faces arrière ne sont pas affichées. Or, en conséquence de la mise en perspective, ce n'est pas forcement le cas. Ci-dessous, un schéma le met en évidence : la normale rouge aurait un z négatif alors que le signe de la verte est positif.

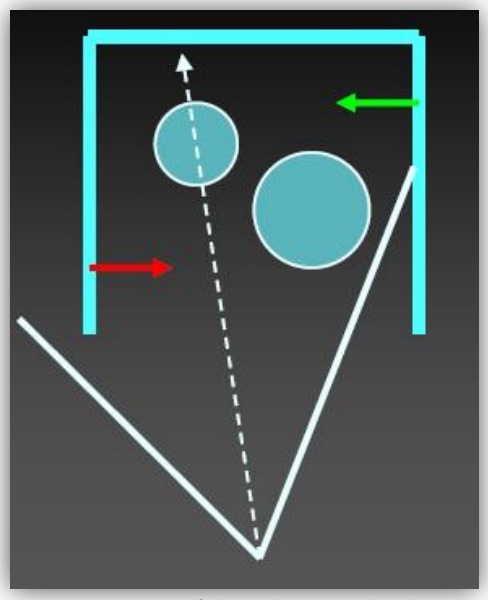


Figure 5 : Un cas où les z des normales dans le repère de l'observateur n'ont pas le même signe

En pratique, on pourrait simplement conserver un bit pour stocker le signe du z. Dans notre implémentation, la place ne manquait pas dans la texture à composantes 32bits, ainsi, les trois composantes x, y et z de la normale sont complètement stockées dans le repère du monde.

4. Light Buffer

Une fois le G-Buffer rendu, le calcul de l'illumination est possible. En effet, le G-Buffer fournit les informations nécessaires et suffisantes au calcul de l'illumination. Par ailleurs, sa complexité est indépendante de la complexité en nombre de vertex. Chaque lumière ajoute sa contribution dans une texture d'accumulation appelée « Light Buffer ». Cette opération est réalisée avec les opérations de blending (mélange) : opération « add » et coefficient à « one ». Une fois toutes les lumières ajoutées au light buffer, une étape de « multiplication globale » est faite, elle consiste à multiplier le light buffer par la couleur diffuse, cette étape peut être réalisée avec un shader ou par une opération de blending.

Deux types de lumières ont été implémentés, les sources ponctuelles non directionnelles d'une part, et les projecteurs avec shadow map d'autre part.

Calcul d'illumination en un pixel

Dans un premier temps, il convient d'exposer le principe de l'illumination en un pixel particulier pour remarquer les points importants. Le modèle empirique d'éclairage utilisé est celui de Blinn-Phong, mais il serait tout à fait possible d'utiliser un autre modèle. En entrée, le modèle de Blinn-Phong a besoin d'une normale, contenue dans le G-Buffer, d'un vecteur pointant vers la lumière et un autre pointant vers l'observateur. Ces deux derniers sont reconstruits à partir de la position dans l'espace monde du pixel à illuminer. En sortie, ce modèle fournit un coefficient spéculaire et diffus pour chaque lampe.

Ici, on ajoute un facteur d'atténuation qui dépend de la distance à la position de la lampe, sur une source ponctuelle, cela correspond à un rayon d'influence. Ici, le facteur d'atténuation est défini par la fonction suivante :

$$A = \left(1 - \left(\frac{x}{r}\right)^2\right)^3 \; sur\; [0:r]$$

$$Sinon\; A = 0$$

Où x correspond à la distance à la lampe et r le rayon d'influence de la lampe.

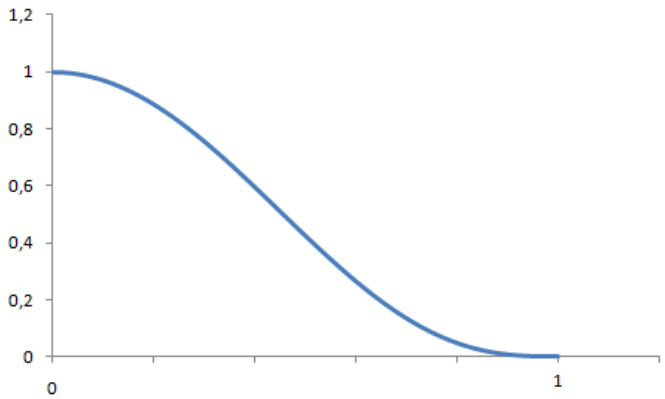


Figure 6 : Le facteur d'atténuation avec r=1 entre 0 et 1

Il est important de remarquer que le facteur d'atténuation s'annule quand la distance à la lampe est égale au rayon. On peut donc s'assurer que la lumière n'éclairera pas plus loin que son rayon.

Limitation du nombre de pixels

Bien que tous les pixels du G-Buffer soient utiles, c'est-à-dire qu'il n'y a pas de fragment qui soit masqué par un autre objet comme ce peut-être le cas en foward rendering, le calcul de l'illumination ne s'effectuera pas sur tout l'écran pour chaque lumière.

En effet, dans la partie précédente, il est montré que les lumières ponctuelles n'éclairent pas plus loin que leurs rayons. Ainsi, au lieu de dessiner un quad devant l'écran pour calculer l'illumination en chaque pixel, une sphère sera transformée de manière à correspondre aux limites de la lumière.

La première approche consiste à utiliser le stencil, on y dessine la sphère puis l'illumination n'est calculée qu'aux pixels marqués dans le stencil, cette méthode fonctionne mais il est possible d'éliminer beaucoup plus de pixels d'atténuation nulle.

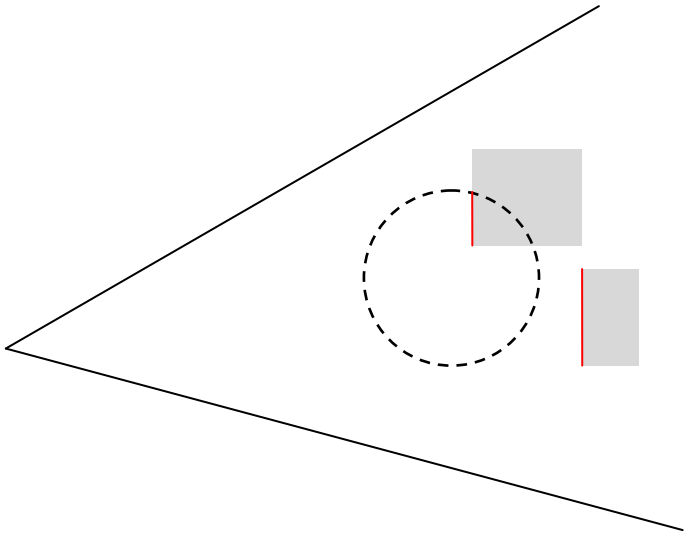


Figure 7 : En pointillé le volume de lumière, en rouge, les pixels où l'éclairage est calculé

Sur le schéma précédent, on remarque qu'un des deux objets n'est pas dans le volume, pourtant, le pixel shader y interviendra. Il est possible d'arrêter le calcul de l'illumination dès lors que l'atténuation est nulle, néanmoins, il est possible de faire beaucoup mieux en utilisant le depth buffer utilisé lors du rendu du G-Buffer. En effet, si les faces arrières de la sphère sont rendues et que seuls les pixels où le test de profondeur ne passe pas[8], on éliminera la plupart des pixels à l'extérieur du volume. Il reste un cas où des pixels d'atténuations nulles seront rendus, une autre méthode utile contre ce problème sera expliquée plus loin.

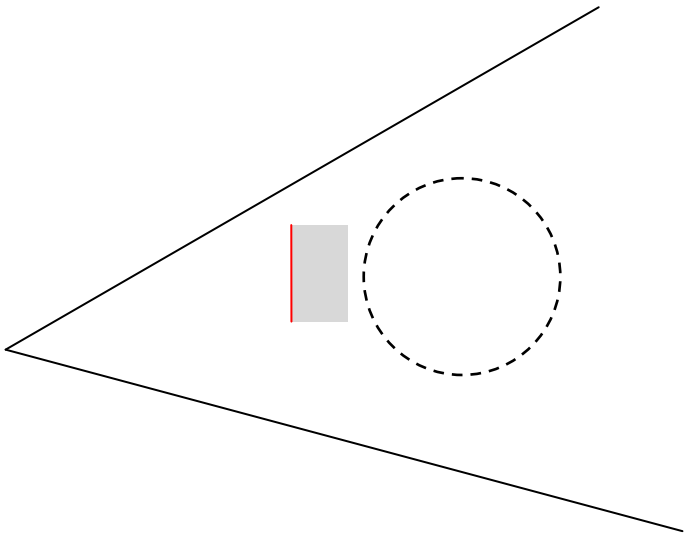


Figure 8 : Un cas où la méthode utilisant le z-fail n'est pas idéale

Malgré cette limite, cette méthode élimine beaucoup de pixels où il n'est pas nécessaire de calculer l'éclairage. Il est possible de réaliser la même opération en une passe sans utiliser le stencil. Pour cela, il suffit encore d'assigner le depth buffer à pipeline, rendre la sphère, et surtout, inverser le test de profondeur dans les options du rasterizer[9].

Figure 9 : Exemple de rendu impliquant 1024 lumières

[8] Communément appelé Z-Fail

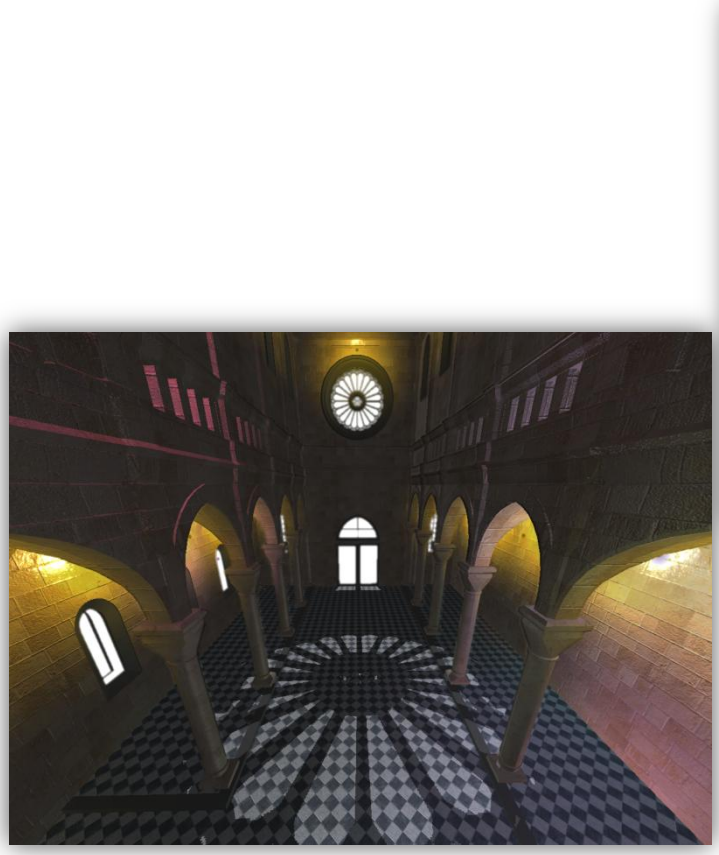
[9] En passant de `D3D10_COMPARISON_LESS` à `D3D10_COMPARISON_GREATER`

Ombres

La technique des shadow map n'est pas incompatible avec le deferred rendering, au contraire, son utilisation est simplifiée grâce au G-Buffer. Cette technique consiste à rendre le tampon de profondeur depuis la position de la lumière pour ensuite, par comparaison de la valeur du Z, déterminer si un fragment est dans l'ombre ou pas. Cette technique est très courante en programmation graphique pour le temps réel[10].

Les lumières qui utilisent des shadow map sont des projecteurs. Comme les lumières ponctuelles, les shadow map possèdent un volume englobant ; la différence réside dans le fait qu'il s'agit de cônes ou de pyramides et non de sphères.

L'intérêt du deferred rendering est mis en évidence lors de l'utilisation de shadow map. En effet, en foward rendering, il faut prendre en compte dans tous les shaders l'application de la shadow map. Ici, la shadow map est une lumière comme une autre, elle est accumulée au light buffer, elle est en rendue en une fois sur l'ensemble de la scène.



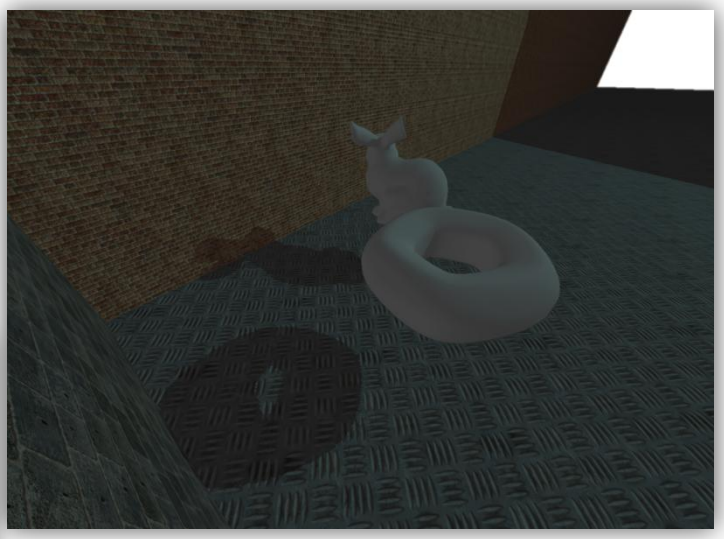


Figure 10 : Exemples d'application de la méthode des shadow map dans le moteur de deferred rendering

[10] Les samplers de comparaison des shaders model 4 peuvent être utiles ici

III. Particularités et Difficultés

1. Occlusion ambiante dans l'espace écran

L'occlusion est un type d'éclairage particulier qu'il est simple d'implémenter en deferred rendering. Il s'inscrit comme une passe dans l'accumulation du light buffer.

Principe

L'occlusion ambiante est une méthode aidant l'éclairage à être plus réaliste en ajoutant un facteur d'occlusion.

Par exemple, dans la réalité, l'intérieur d'un tube ne sera que rarement éclairé, l'occlusion à l'intérieur du tube sera plus forte qu'à l'extérieur. Cette méthode est courante dans les applications de lancer de rayons. Il s'agit d'émettre, en chaque point d'une surface, un grand nombre de rayons dans un hémisphère et de mesurer l'occlusion en fonction du nombre d'intersections.

Figure 11 : Exemple d'occlusion ambiante réalisée avec le moteur de lancer de rayon VRay

Espace écran

En temps réel, il est difficile d'appliquer un algorithme utilisé dans le lancer de rayon[11]. Ici, on va utiliser les informations du G-Buffer pour approcher la valeur d'occlusion ambiante. Il existe beaucoup de présentations du principe de l'occlusion ambiante dans l'espace écran, plus souvent abrégé SSAO pour Screen Space Ambiant Occlusion. Les différentes méthodes reprennent globalement le même principe : il s'agit d'utiliser le tampon de profondeur pour mesurer l'occlusion de chaque pixel en fonction des profondeurs voisines.

[11] Il existe une autre méthode, voir chapitre 14 du GPU Gems 2

Dans notre situation, l'ambiante occlusion remplace la passe de lumière ambiante qui initialise le light buffer avec une faible luminosité. Le G-Buffer fournit directement la profondeur. Le facteur d'occlusion est calculé en analysant les profondeurs dans huit directions à une distance de quelques pixels. Ces vecteurs de directions sont dépendants de la normale au pixel traité de manière à éviter les artefacts dus au nombre limité de pixels analysés et de la constance des directions initiales[12].

Résultat

En pratique, l'occlusion ambiante dans l'espace écran est un effet très intéressant, il permet d'ajouter du relief et de mettre en évidence les objets enclavés.

Néanmoins, il ne faut pas oublier que cet effet multiplie les accès au G-Buffer, et donc des accès texture qui sont le plus souvent l'opération la plus couteuse dans un shader. Pour tous les objets immobiles et à voisinage immobile, calculer l'occlusion de chaque fragment est inutile, il est possible d'associer une valeur d'occlusion par vertex ou de disposer d'une carte d'occlusion. Ce calcul peut être interne au programme ou plus aisément, généré par une application tierce[13]. Il pourrait alors être intéressant de combiner l'occlusion ambiante pré-calculée avec celle générée en temps réel dans l'espace écran en limitant l'effet de cette dernière aux zones concernées.

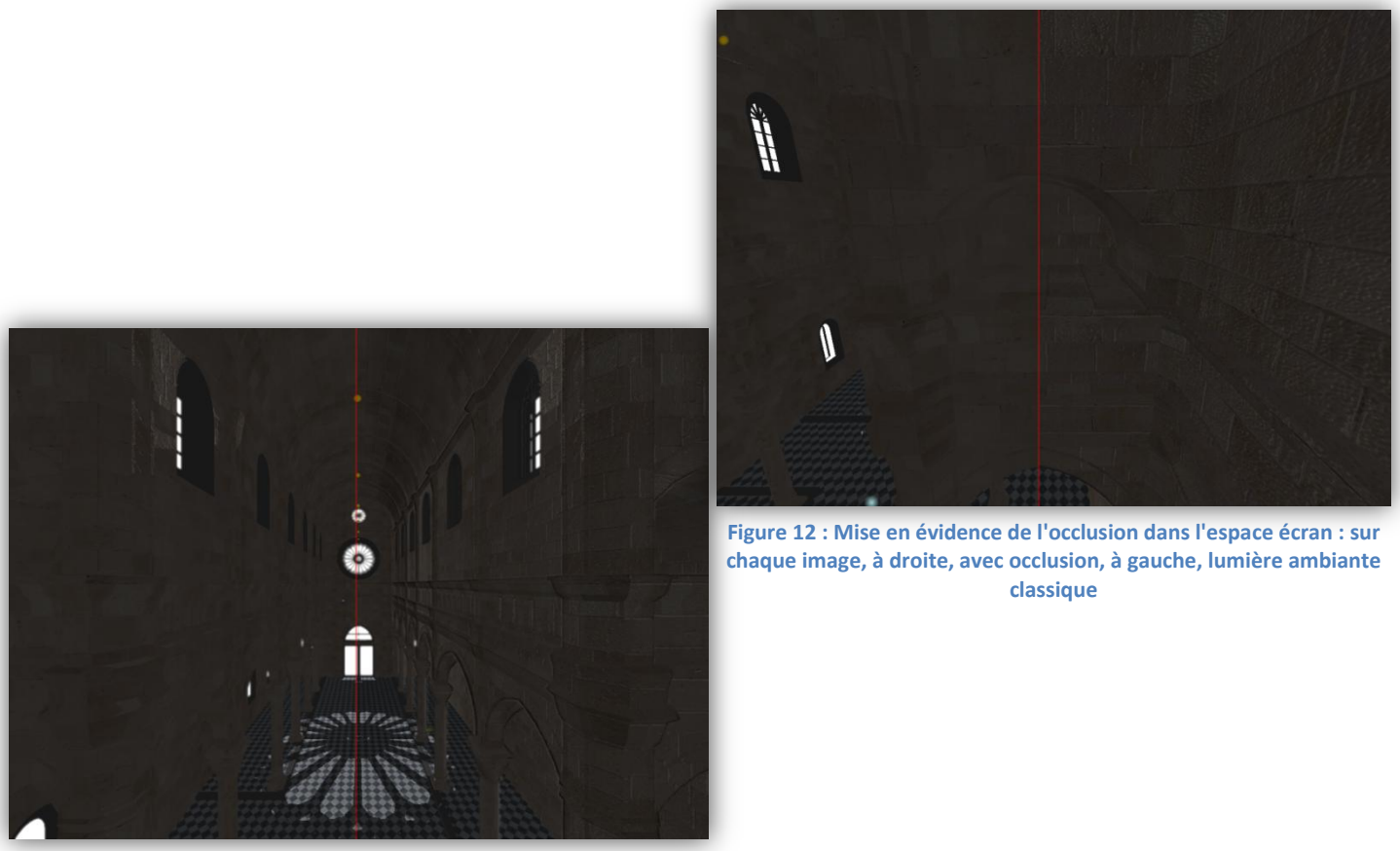


Figure 12 : Mise en évidence de l'occlusion dans l'espace écran : sur chaque image, à droite, avec occlusion, à gauche, lumière ambiante classique

[12] Voir la référence d'Iñigo Quilez pour plus de précisions
[13] Du coté des freewares : MeshLab a cette fonctionnalité.

2. Transparence

La transparence est un problème récurrent en programmation graphique et encore plus lors de l'utilisation du deferred rendering. En effet, de base, le G-Buffer est conçu pour contenir une et une seule structure d'informations correspondant à un fragment dans l'espace. Dans cette partie, nous présenterons plusieurs techniques à la manière de les implémenter en deferred rendering.

Foward rendering

La méthode la plus simple et probablement la plus utilisée consiste à ne pas rendre les objets transparents en deferred. Dans la première passe, ce sont uniquement les objets solides qui sont rendus dans le G-Buffer, ensuite, les objets transparents sont rendus (avec un éclairage limité) en activant les fonctions de mélange. C'est cette méthode qui est utilisée par les développeurs de S.T.A.L.K.E.R. Il est aussi possible de réaliser un shader utilisant la texture contenant le résultat pour estimer la déformation du fond en fonction des normales à l'objet. Cet effet ne peut être pas physiquement réaliste mais reste très peu coûteux et intéressant, c'est ainsi qu'a été implémentée la transparence dans notre application.

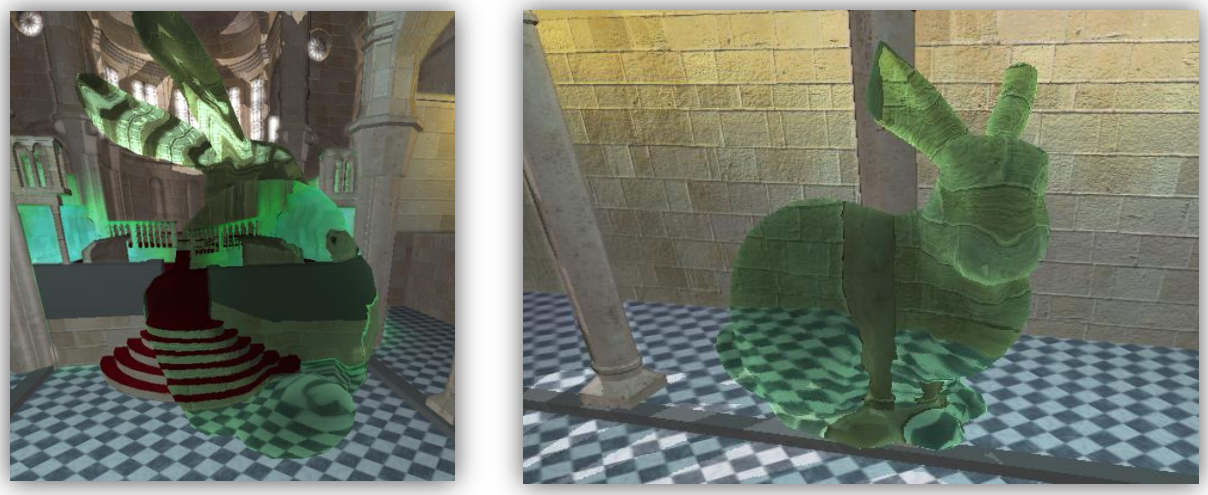


Figure 13 : Exemples de transparence avec dérivation des rayons

Depth Peeling

Une autre approche consiste à utiliser un algorithme de "Depth Peeling". Rappelons que pour rendre correctement des objets transparents il est nécessaire de les ordonner. Or, il est très difficile d'implémenter un tri efficace sur le GPU. Une solution est de le réaliser par l'application, mais cela a un certain coût.

C'est ici qu'intervient le depth peeling. Cette méthode est basée sur des calques correspondant à une profondeur minimale et une maximale, ces calques sont ensuite rendus du fond vers l'avant. Cette méthode est souvent utilisée pour le volume rendering notamment en imagerie médicale. En deferred rendering, il est possible d'écrire sur plusieurs calques à la fois, mais cela multiplie les textures et passes d'illumination, ce qui peut avoir un réel impact sur les performances.

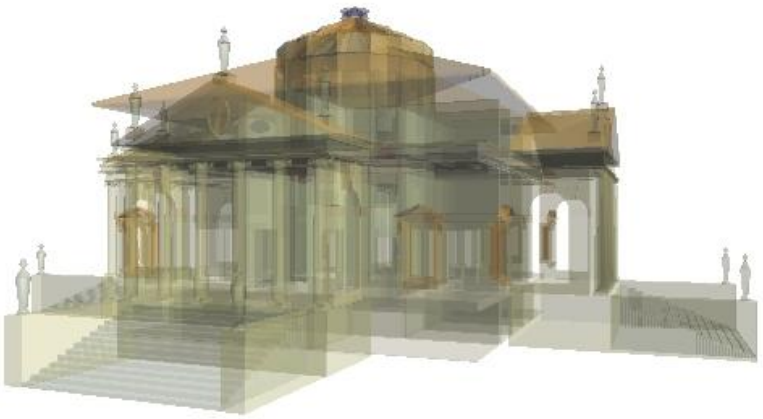


Figure 14 : Exemple de résultat utilisant le depth peeling

Screen door transparency

Il existe une autre technique rarement utilisée appelée "screen door transparency". Elle était très courante dans les jeux d'arcade comme Daytona USA ou certains jeux en 2D comme Golden Axe à l'époque où les opérations de blending n'étaient pas toujours supportées. On retrouve cette technique dans la méthode nommée "Alpha To Coverage" mais appliquée par du multi-sampling.

En réalité, il s'agit d'une fausse transparence où on pourrait comparer les objets transparents à un grillage. Un modèle moucheté (stippling pattern) est défini. Par exemple, on définit un schéma de colonnes paires à 0 et impaires à 1 pour réaliser une transparence de 50%. Ce schéma est ensuite appliqué dans l'espace écran sur les polygones transparents. Les pixels où le pattern est égal à 1 sont gardés, les autres sont supprimés. Ainsi, l'effet de transparence est surtout rendu par nos yeux en mélangeant les pixels proches. Cette méthode est directement implémentable en deferred et ne requiert pas de tri de profondeur.

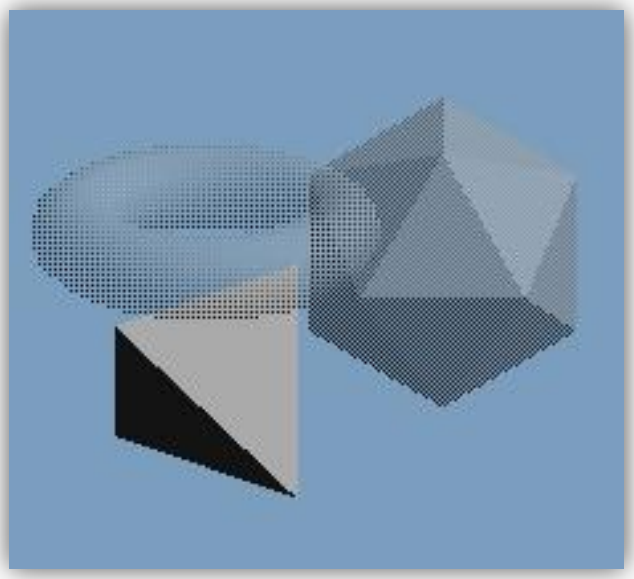
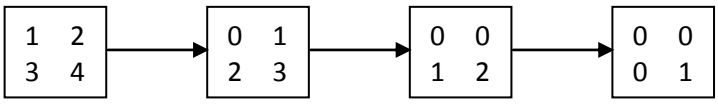


Figure 15 : Exemple d'application utilisant le screen door transparency

Stencil Routed A-Buffer

Probablement la technique la plus évoluée, le Stencil Routed A-Buffer utilise des textures multi-échantillonnées pour stocker plusieurs fragments par pixel. Cette technique se déroule en trois passes.

Premièrement, les N fragments, correspondant aux N échantillons par pixels, sont capturés dans l'ordre de la rastérisation. Ensuite, les fragments sont triés en fonction de leur profondeur. Enfin, la troisième passe s'occupe de mélanger les fragments pour restituer la couleur mélangée. La principale difficulté de cette méthode est la première passe. Pour que chaque pixel contienne des fragments différents, c'est le stencil qui est utilisé. Tout d'abord, il est initialisé avec une valeur différente par échantillons, de 1 à N. Ensuite, seuls les pixels ayant une valeur à 1 sont rendus et à chaque écriture, le stencil de l'échantillon est décrémenté (D3D10_STENCIL_OP_DECR_SAT). Ainsi, à chaque accès à un pixel donné, l'échantillon où la valeur est stockée est différent. Ci, dessous, un schéma montre l'effet de la décrémentation des quatre échantillons sur un pixel du stencil lors d'un accès.



Le Stencil Routed A-Buffer est en moyenne 8 fois plus rapide que le depth peeling. En deferred rendering, en admettant un Geometry Buffer multi-échantillonné, cette méthode est possible. Chaque pixel du G-Buffer contiendra alors N fragments, l'éclairage y serait effectué avant de les mélanger. L'étude de cette technique est à approfondir dans le contexte du deferred rendering. Il s'agit, a priori, de la méthode la plus intéressante pour réaliser de la transparence en temps réel.

Figure 16 : Exemple d'application utilisant le Stencil A Routed Buffer (depuis NVidia)

3. Anticrénelage

En programmation graphique, le crénelage est un effet nuisible résultant de l'échantillonnage de l'image. Il existe beaucoup de technique visant à améliorer le résultat comme par exemple, récemment, le Coverage Sampler Anti aliasing.

Les techniques d'anticrénelage matériel sont clairement incompatibles avec le deferred rendering en raison de la nature même du Geometry Buffer. Il peut être nécessaire de mettre au point une technique de substitutions. L'anti-aliasing reste une partie complexe, cette présentation n'est pas exhaustive en conséquence des nouvelles possibilités offrant DirectX 10.1 qui reste encore à étudier.

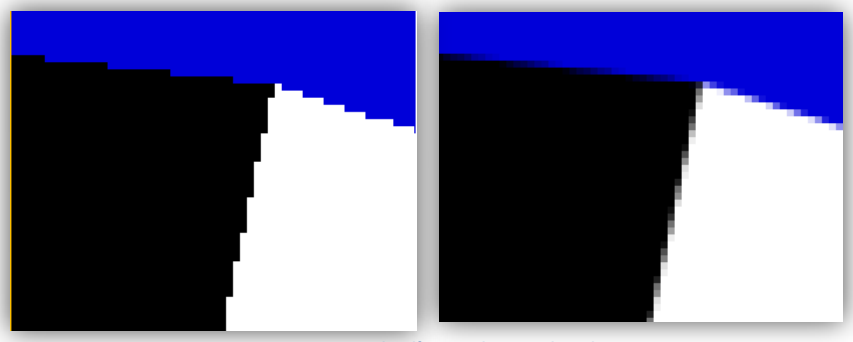


Figure 17 : Exemple d'anti-aliasing hardware

Over-Sampling

L'approche la plus simple est l'over-sampling : le rendu est réalisé avec une très grande résolution puis redimensionné et interpolé. Par exemple, le rendu peut être réalisé en 1600x1200 pour être affiché en 800x600 à la passe finale. Cette méthode élimine clairement les effets de crénelage mais requiert beaucoup de mémoire, ce qui peut être prohibitif dans beaucoup d'applications.

Edge Smoothing-Filter

Une autre solution appelée "edge-smoothing filter" consiste à flouter les contours après les passes d'illumination et de postprocessing. Elle peut être réalisée en deux passes.

La première passe est chargée de détecter les contours en analysant la discontinuité dans les positions et normales pour écrire un stencil buffer. La seconde passe se charge de flouter les pixels dans le stencil buffer. En pratique, cette technique peut être réalisée en une passe mais il est important de limiter le nombre d'accès texture aux endroits où l'effet n'est pas appliqué.

Cette technique est simple d'implémentation et très peu coûteuse, on peut noter qu'il est possible d'activer et de désactiver cet effet en temps réel ce qui est souvent difficile avec les méthodes d'anti-aliasing classiques. Elle a été implémentée dans l'application de démonstration du deferred rendering.

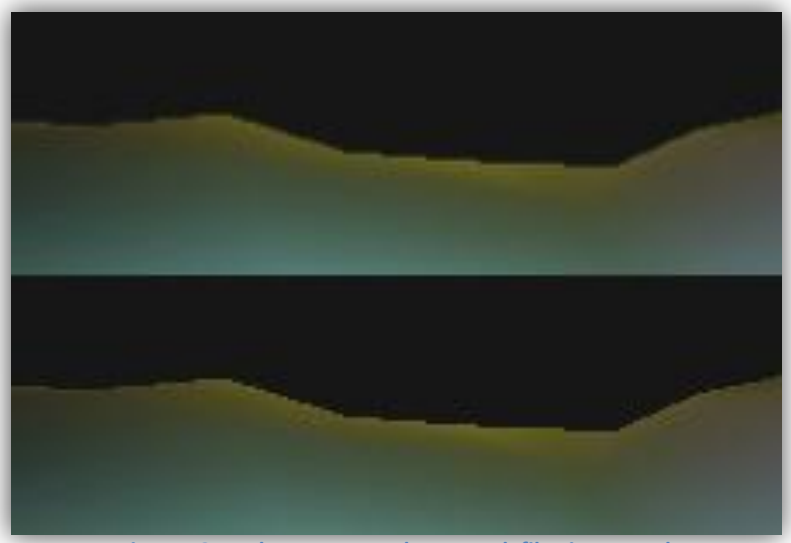


Figure 18 : En haut, avec « edge smooth filtering » ; en bas, sans « edge smooth filtering »

Anticrénelage sur la couleur

Enfin, une dernière solution consiste à rendre les couleurs dans une texture multi-samplée. En effet, dans la plupart des cas, les effets et lumière n'ont pas besoin d'anticrénelage, seuls les couleurs sont sensibles à l'aliasing.

Cette dernière méthode peut être combinée à la précédente (edge-smoothing filter) pour améliorer le résultat final. En pratique, cette méthode revient à utiliser la technique du « Light Pre-Pass Renderer » car elle impose de rendre les couleurs diffuses à part.

IV. Post-Processing

Après l'accumulation des lumières dans le light buffer, il est possible de réaliser des passes supplémentaires très facilement. Ces dernières peuvent profiter des informations contenues dans le light buffer et le geometry buffer.

1. Bloom

Le bloom (ou glow) est un effet visant à reproduire les artefacts des caméras réelles autour des zones très claires. En deferred rendering, c'est un bon exemple d'effet appliqué en post processing.

Principe

La raison physique du bloom réside dans le fait que les caméras réelles ne peuvent jamais régler parfaitement le focus. Même une lentille parfaite convolue l'image en entrée par un disque. Dans la plupart des cas, cet effet n'est pas visible. Quand des zones très illuminées contrastent avec des zones plus sombres, le glow devient perceptible. En programmation graphique, pour réaliser cet effet, les zones très claires sont isolées, puis floutées et enfin additionnées à l'image d'origine.

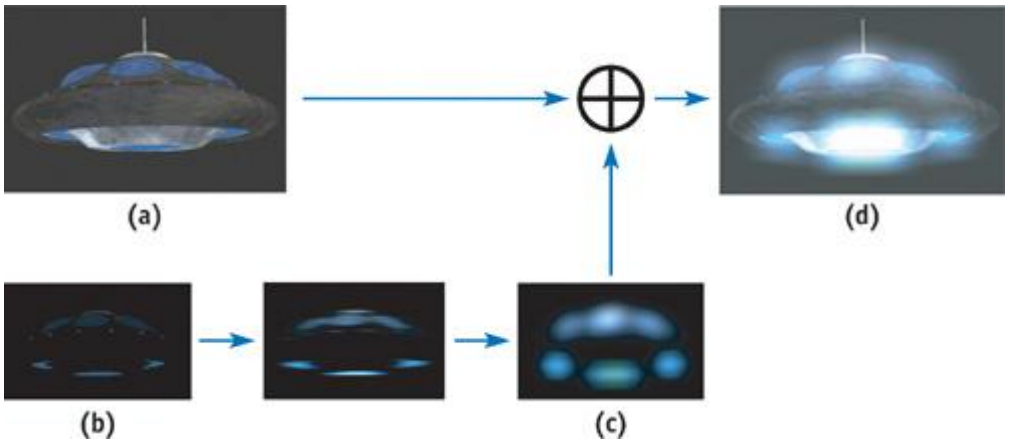


Figure 20 : Schéma du fonctionnement du bloom (depuis GPU Gems)

Figure 19 : Exemple de glow sur une image réelle : le blanc du fond "déborde"

Pratique

En pratique, cette technique est très simple en deferred rendering. Après avoir rendu l'illumination et effectué la multiplication globale, les parties de l'image où l'intensité spéculaire sature (supérieure ou égale à 1) sont isolées dans une texture puis floutées par simple convolution d'un noyau gaussien ou moyenneur. Le résultat est ensuite additionné à l'image finale.

Figure 21 : En haut, avec bloom, en bas, sans bloom

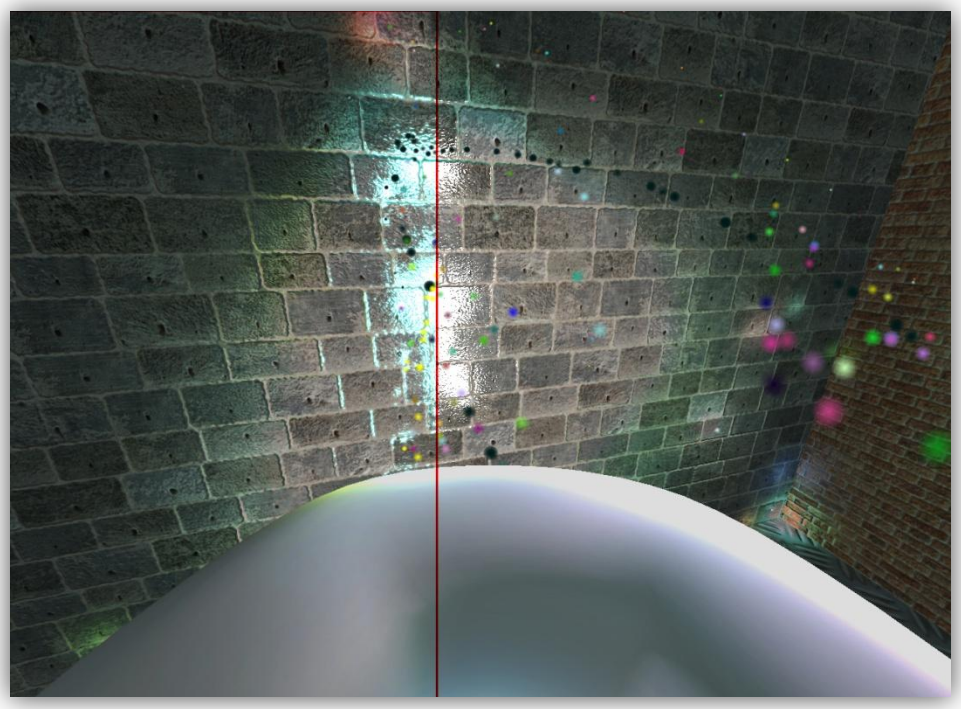


Figure 22 : A gauche, avec bloom, à droite, sans bloom

2. Flou de mouvement

Le motion blur ou flou de mouvement est, comme le bloom, à l'origine un artefact des périphériques de capture.

Principe

En photographie, le flou cinétique est le résultat d'un mouvement de l'objectif ou du sujet lors du temps d'exposition de l'appareil. En effet, en raison de contraintes techniques ou artistiques, une photographie correspond à un intervalle de temps plus ou moins long. Le flou de mouvement est aussi perceptible par nos yeux en conséquence de la persistance rétinienne.

En programmation graphique, plusieurs raisons peuvent mener à utiliser le motion blur. La première raison est d'accentuer un effet de vitesse sur un objet en mouvement, mais aussi lors du déplacement rapide de l'observateur : dans une simulation de conduite automobile, la périphérie du champ vision est souvent floutée à grande vitesse pour s'approcher des troubles de vue observées dans la réalité. La seconde raison est de simuler une certaine fluidité quand le nombre de frames par seconde est faible, c'est en quelque sorte l'astuce du cinéma. En effet, bien qu'il n'y ait, la plupart du temps, que 24 images par secondes, l'animation semble fluide, c'est en raison du flou de mouvement sur les images.

Pratique

Il existe beaucoup de techniques de motion blur, certaines sont par exemple basées sur un tampon d'accumulation. En deferred rendering, il est simplement réalisable en stockant la vitesse dans l'espace écran à chaque pixel puis, en post-processing, utiliser un filtre de flou directionnel. Notons qu'il peut être nécessaire d'appliquer un filtre moyenneur sur les vecteurs de vélocité afin d'atténuer les contours des objets impliqués dans cette technique.

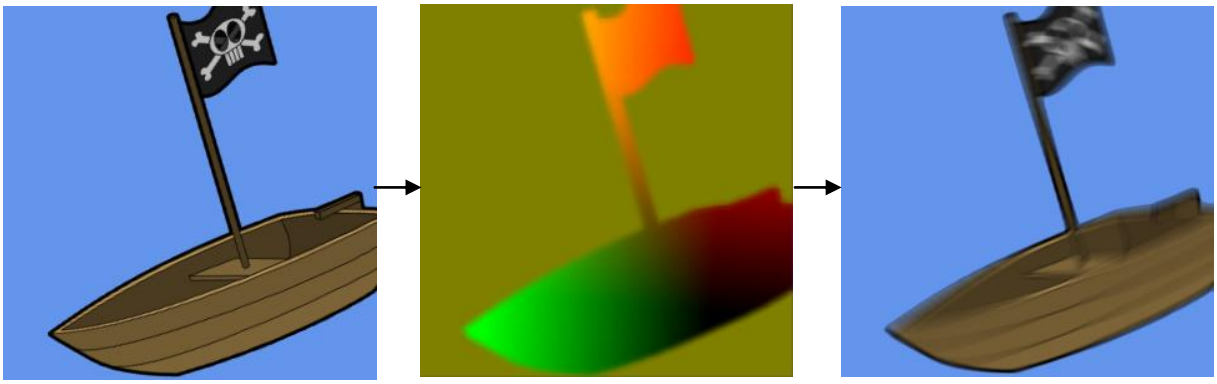


Figure 23 : Exemple d'application de motion blur

3. Crayonnage

Le crayonnage ou « hatching » ou encore « hachure » est originellement une méthode de représentation apparue pendant la renaissance visant à remplacer des couleurs par des motifs particuliers en noir et blanc comme des points ou des lignes.

Figure 24 : Veronica d'Albrecht Dürer en 1513 ; exemple de la methode de hachurage

Principe

En programmation graphique pour le temps réel, le hatching est une méthode de rendu non réaliste basée sur des textures de ton. Ces textures sont ensuite utilisées pour afficher des couleurs plus ou moins sombres.

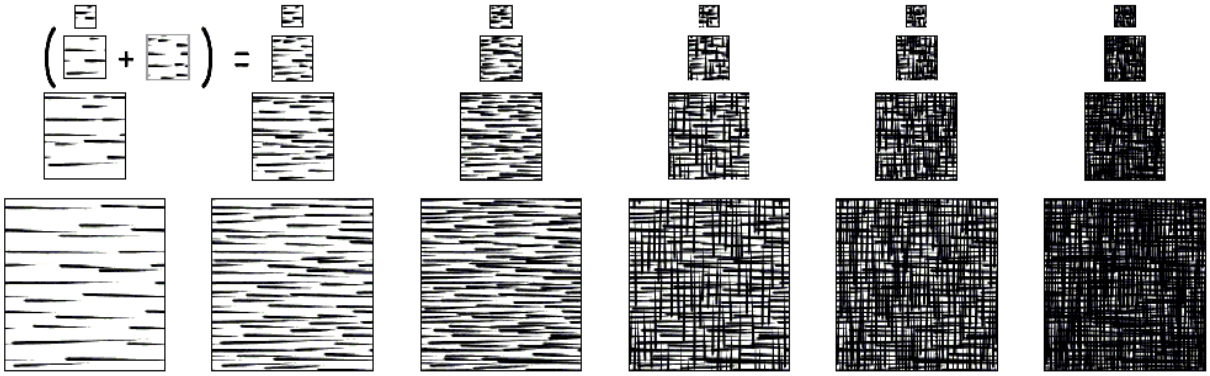


Figure 25 : Exemple de jeu de textures (avec mipmaps)

Pratique

Dans l'application de deferred rendering, ce procédé est utilisé en post-traitement. Pour chaque pixel, une valeur flottante entre 0 et 6 est calculée en fonction de la luminosité, c'est cette valeur qui indique quelle texture utiliser. Les motifs sont contenus dans un TextureArray et le mélange entre deux motifs se fait linéairement. Les couleurs sont désaturées mais pas complètement, il s'agit surtout d'un effet artistique, un exemple de rendu qu'il est simple de réaliser en deferred rendering.

Figure 26 : Exemple de rendu "hatching"

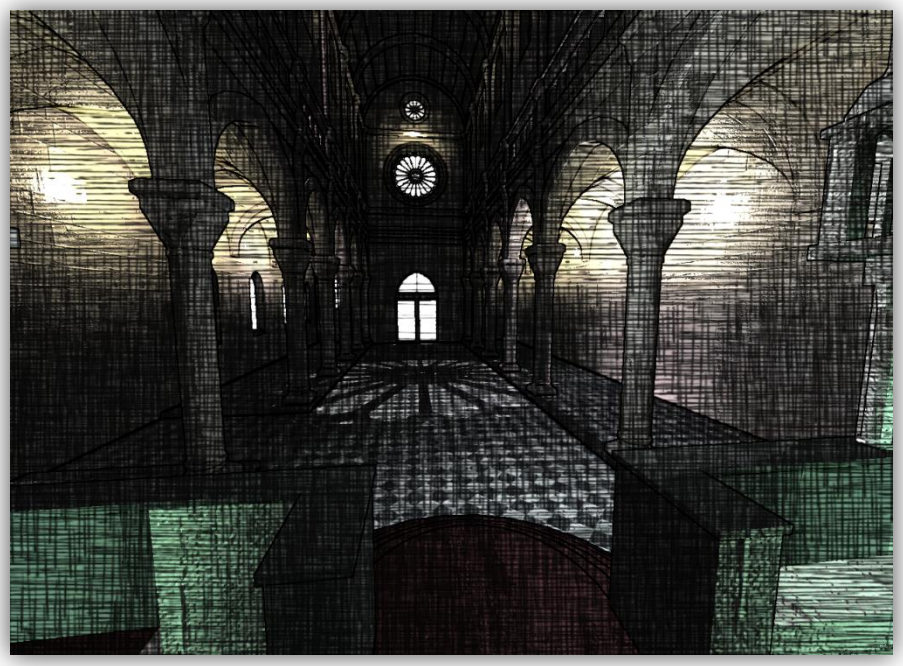


Figure 27 : Exemple de rendu "hatching"

V. Améliorations et Variantes

Dans cette partie, deux points seront traités, tout d'abord, deux modifications sur les passes de rendus des lumières seront présentées. Ensuite, sont mises en avant deux variantes du principe même du deferred, le light pre-pass renderer[14] est plus proche du deferred que le light indexed buffer.

1. Accumulation de l'éclairage variantes

Dans la partie sur le light buffer, il est question de la manière dont les lumières sont rendues. Ce sont des volumes englobant la lumière qui sont envoyé à la carte graphique pour limiter le nombre de fragments où l'illumination sera calculée.

Geometry shader

La première variante utilise les Geometry Shader pour générer à la volée les polygones englobant à chaque lumière. Les Geometry Shader ont été introduis avec DirectX 10.0 et la shader model 4.0. Ces shaders sont exécutés après le vertex shader mais avant l'étape de rastérisation, ils ouvrent beaucoup de perspectives. Deux particularités ont été utilisées, d'une part, il est possible d'avoir des points en entrée du GS mais des triangles en sortie, ce qui permet de rendre des sprites à la volée. D'autre part, le Geometry Shader permet de mettre à jour les vertex[15], cette fonctionnalité permet de facilement animer des particules.

Dans la première version de l'illumination, toutes les lumières ponctuelles sont indépendantes, seul le maillage, une sphère unité, qu'elles utilisent est commun. Il aurait été possible de les rendre en une seule fois par utilisation des objets multi-instanciés mais c'est une autre piste qui a été explorée. Les lumières sont stockées dans un vertex buffer et ne contiennent qu'un minimum d'informations comme la position, la couleur et la taille. Ce vertex buffer n'est associé qu'une seule fois au pipeline graphique. Des quads (deux triangles) sont générés face à la caméra et de taille dépendant du rayon de la lumière.

Evidement, certains pixels potentiellement inutiles car hors du rayon d'action sont rendus. Néanmoins, toutes les lumières sont rendues en un seul appel et surtout, comme il est précisé précédemment, il devient possible d'utiliser le GS pour mettre à jour ce vertex buffer et donc animer en temps réel les lumières comme étant des particules. Pour cela, deux autres informations ont été ajoutées pour chaque vertex, un vecteur vitesse et une

[14] Parfois appelé « prelighting »
[15] En utilisant les « Stream Output » ou SO

valeur de « durée de vie », ensuite, une passe préalable est ajoutée pour mettre à jour les positions des vertex par un autre Geometry Shader.

Cette méthode n'est pas encore optimale. Même si la perte de performance est minime, il serait intéressant de l'optimiser en ne générant pas forcément de quad, plutôt des disques, des cubes ou des sphères. Cette technique permet avant tout de montrer l'intérêt du deferred rendering dans l'éclairage dynamique et animé dans des conditions de performances plus que convenables (dans l'exemple ci-dessous, un peu plus de mille lumières sont animés entre 45 et 70 frames par secondes sur une 9800GT).

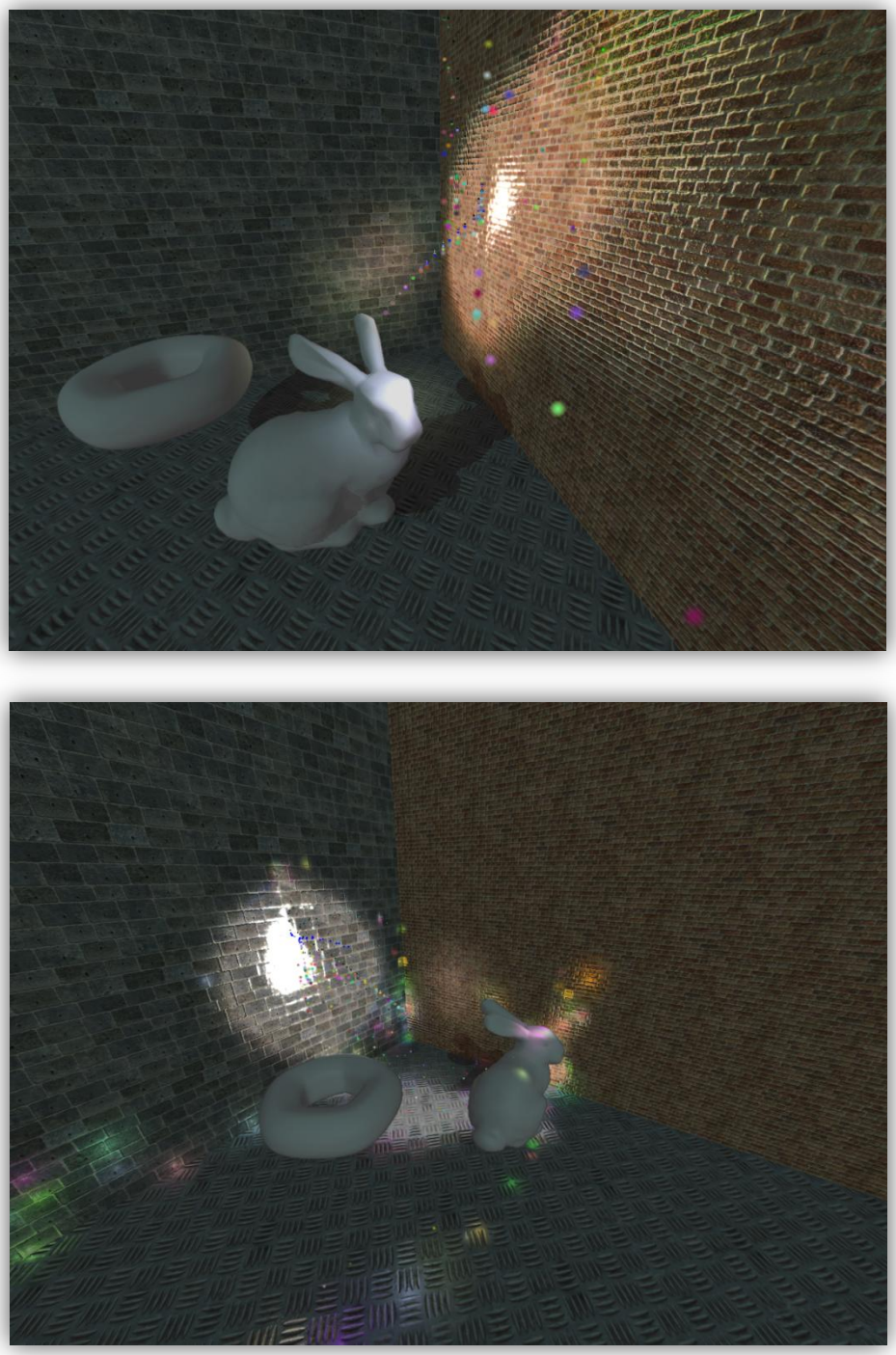


Figure 28 : Exemples de lumières animées

Stencil

Une autre méthode, plus optimisée en termes d'appel au pixel shader effectuant l'illumination, consiste à utiliser le stencil buffer pour déterminer les pixels à l'intérieur du volume de lumière. Le principe est simple et s'effectue en deux passes, la première écrivant dans le stencil buffer les pixels à illuminer, et la seconde le réutilisant pour calculer l'éclairage seulement aux endroits nécessaires.

La première passe doit isoler les pixels dans le volume de lumière. Pour cela, le tampon de profondeur des géométries (ce qui doit être illuminé) va être assigné. L'écriture dans le tampon de profondeur doit être désactivée et des opérations sur le stencil différentes entre les faces avant et arrière vont être attribuées. Les faces avant vont incrémenter le stencil, les faces arrière vont le décrémenter. Ainsi, seuls les pixels à l'intérieur du volume auront un stencil différent de 0. Les fragments entre le volume et la caméra ne seront pas nécessairement illuminés comme ce peut être le cas avec la méthode présentée dans la première partie. Notons qu'avec cette technique, les volumes sont nécessairement fermés.

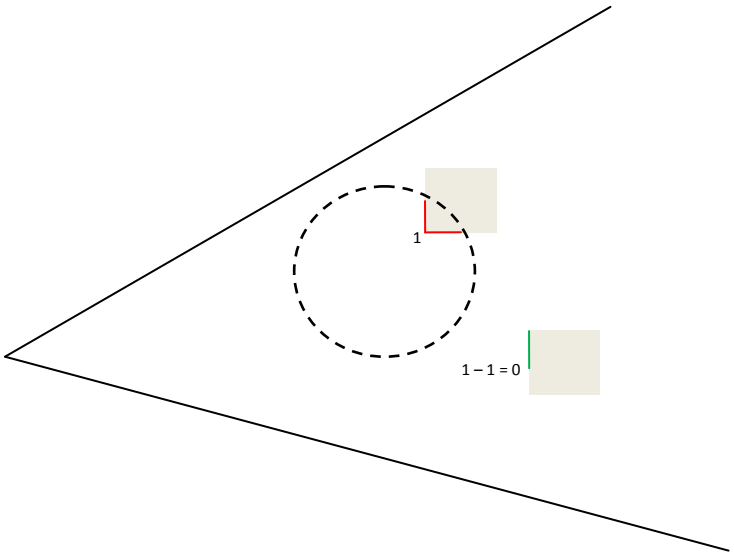


Figure 29 : Premier cas : le volume de lumière occulte deux éléments mais seulement un est à l'intérieur

Cette technique se rapproche clairement des shadow volumes. La technique des shadow volumes est une méthode visant à isoler les parties dans l'ombre d'un objet en utilisant le stencil. Les faces avant du volume incrémentent et arrière décrémentent, de la même manière que dans notre situation. De base, les shadow volumes ne peuvent pas fonctionner lorsque la caméra est située à l'intérieur du volume d'ombre. C'est le même problème ici, lorsque la caméra est située à l'intérieur du volume de lumière, seules les faces arrières sont visibles, sachant que l'on garde seulement les fragments passant le test de profondeur, les zones à éclairer ne pourront pas être isolées.

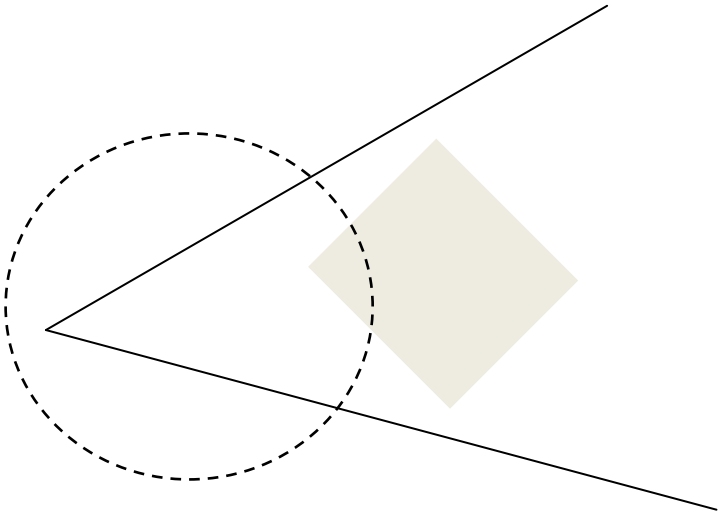


Figure 31 : Second cas : l'œil est à l'intérieur du volume, cette technique ne peut être assurée

Figure 30 : Exemple d'isolation des pixels à éclairer (en rouge)

2. Light Pre-Pass Renderer

Présentation

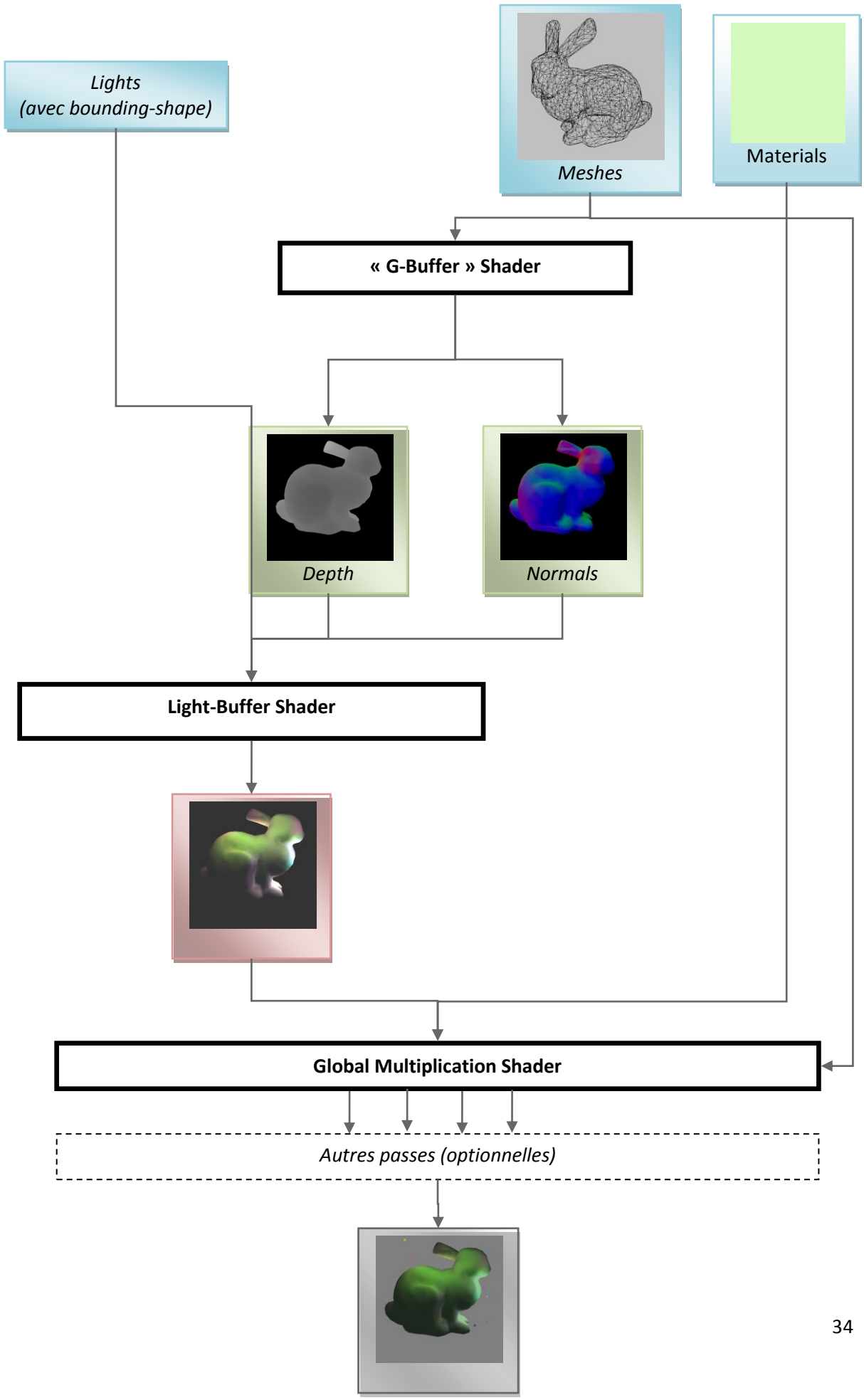
La méthode du Light Pre-Pass Renderer, introduite par Wolfgang Engel, est plus proche du deferred rendering que la technique de "Light Indexed Deferred lighting". Cette technique consiste à limiter le G-Buffer à deux données : la normale et la profondeur. Ce sont les seules informations nécessaires à la création du light buffer.

Il est possible de dégager trois passes pour expliquer cette méthode. La première consiste donc à rendre le G-Buffer contenant uniquement les normales et profondeurs pour chaque pixel, le G-Buffer est donc très facilement stockable dans une texture, et ce, sans forcément effectuer de packing particulier. Ensuite, la seconde passe se charge de calculer le light buffer de la même manière qu'en deferred rendering : les volumes des lumières sont rendus et les informations dans le light buffer sont accumulées. Ensuite, c'est ici que le rendu diffère du deferred rendering : la dernière passe rend les objets texturés ou colorés en foward rendering et utilise le light buffer pour afficher les matériaux.

Cette technique a l'avantage de bien supporter l'anti-aliasing hardware puisque, comme il est précisé dans la partie sur l'anti-crénelage, c'est le plus généralement sur les couleurs que l'effet de crénelage se produit. Par ailleurs, les opérations de blending sont alors simplifiées même si l'éclairage sur les surfaces transparentes ne peut être facilement assuré.

C'est cette technique qui a été désignée pour le moteur de rendu « Cry Engine 3 » présenté lors de la Game Developers Conférence de 2009, pour des raisons d'économie de mémoire et pour plus de flexibilité.

Schéma


Lights
(avec bounding-shape)
Meshes
Materials
« G-Buffer » Shader
Depth
Normals
Light-Buffer Shader
Global Multiplication Shader
Autres passes (optionnelles)

3. Light Indexed Deferred

Présentation

La technique "Light Indexed Deferred lighting" étudiée par Damian Trebilco est une variation du deferred rendering visant à être un bon compromis entre le foward et le deferred rendering.

La théorie de cette méthode est très simple : l'objectif est de stocker la liste des lumières par fragment. Pour cela, le tampon de profondeur (l'information suffisante pour reconstruire une position) est rendu dans une première passe. Ce rendu est en général très peu coûteux en termes de performance. Ensuite, les indices des lumières influentes sur chaque pixel sont définis. En d'autres termes, chaque pixel contient un tableau d'indices référents aux lumières dans lesquelles le pixel apparaît. Sur ce dernier point, plusieurs méthodes sont possibles, certaines utilisant le multi-passe tandis que d'autres utilisent le décalage de bit, le choix dépendra aussi du nombre maximal de lumières par fragment, il s'agit de la partie la plus complexe de ce procédé.

Enfin, les objets sont rendus en utilisant le foward rendering où l'illumination utilise le buffer précédent. Le principal défaut de cette technique réside dans la difficulté d'implémenter des lumières particulières, notamment les shadow maps. Par ailleurs, comme la technique du « Light Pre-Pass Renderer » toute la géométrie est assignée deux fois à la carte graphique. Néanmoins, les opérations de blending deviennent directement possibles. L'éclairage peut même y être effectué et l'anti-aliasing hardware n'est plus un problème.

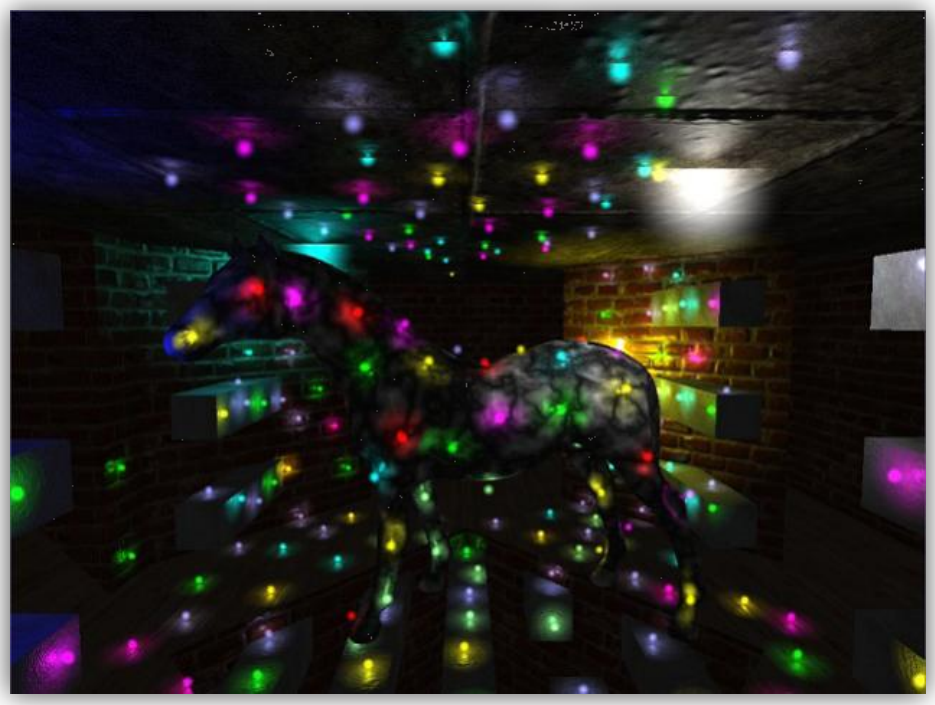
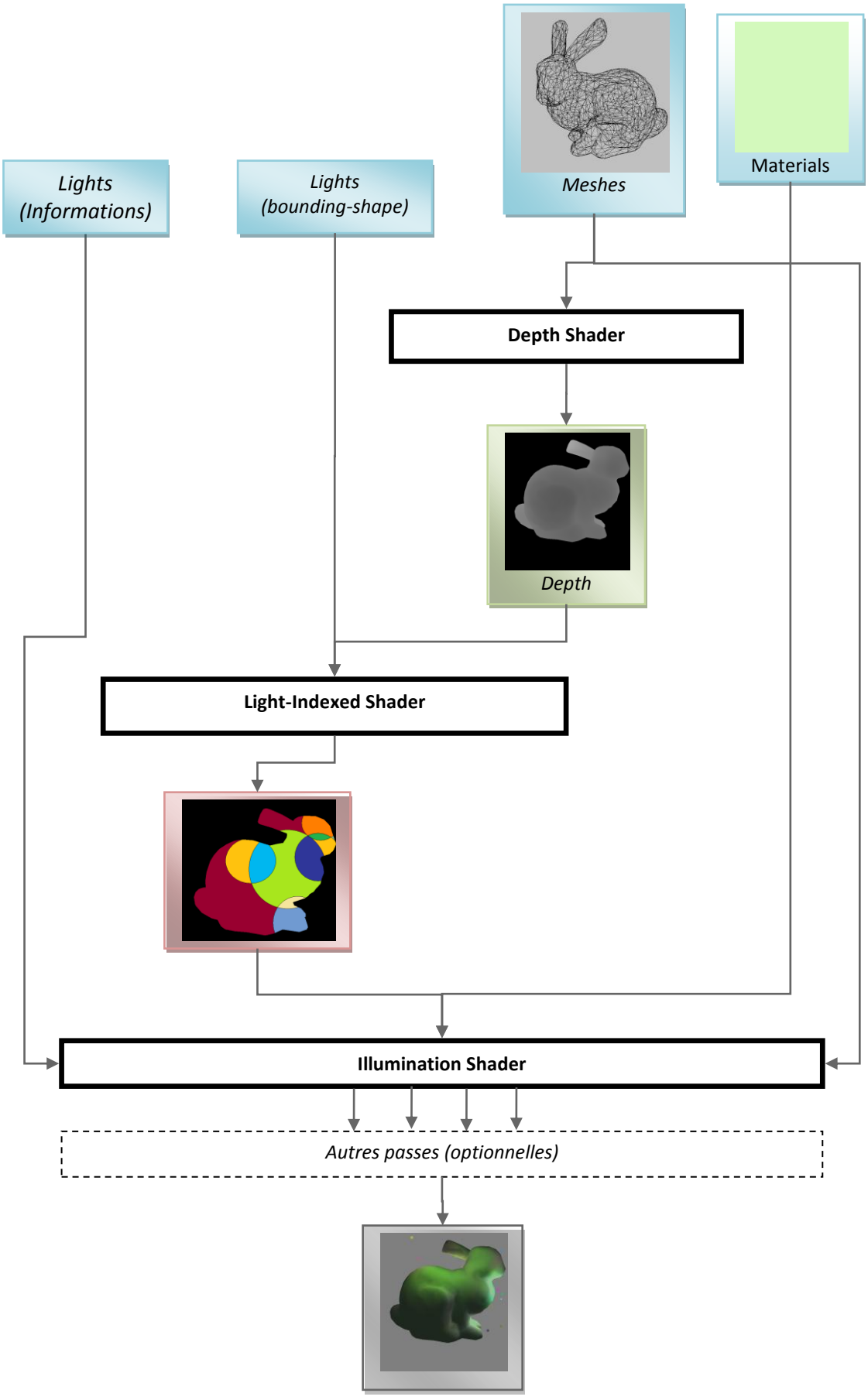


Figure 32 : Exemple d'application en Light Indexed Deferred

Schéma


Lights
(Informations)
Lights
(bounding-shape)
Meshes
Materials
Depth Shader
Depth
Light-Indexed Shader
Illumination Shader
Autres passes (optionnelles)

VI. Conclusion

Le deferred rendering n'est pas forcément le meilleur choix en toutes conditions, il reste toutefois une approche très intéressante en termes de performance mais aussi en vue de simplifier un développement. Dans ce rapport, il a été montré que les difficultés des effets complexes telles que le flou de mouvement ou l'occlusion sont grandement diminuées par l'utilisation du Geometry Buffer. De même, ce rapport montre que les problèmes de transparence et d'anti-aliasing ne sont pas une fatalité dans l'utilisation d'un moteur basé sur le deferred rendering.

En plus de m'apporter des compétences à l'utilisation de DirectX 10 et des shader model 4.0, ce projet de recherche m'a permis d'assimiler de nouveaux concepts de programmation graphique. Le développement d'un moteur de rendu générique n'est pas une tâche aisée, le deferred rendering permet de simplifier le design de ce genre d'application.

Je souhaite implémenter un moteur utilisant la technique du « Light Pre-Pass Renderer » car elle me semble être un très bon compromis entre le deferred et le foward rendering. Par ailleurs, cette technique est plus souple et ne souffre pas des difficultés d'un rendu déferré classique. En définitive, ce travail de recherche m'a permis de recevoir une très bonne préparation au programme du master II en imagerie.

VII. Bibliographie

1. **Thibieroz, Nicolas.** Deferred Shading with Multiple Render Targets. [auteur du livre] Wolfgang F. Engel. *Shader X².* 2003, pp. 251-269.

2. **Shishkovtsov, Oles.** Deferred Shading in S.T.A.L.K.E.R. [auteur du livre] Matt Pharr and Randima Fernando. s.l. : Addison-Wesley Professional, 2005. Aussi disponible en ligne : http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html.

3. **Koonce, Rusty.** Deferred Shading in Tabula Rasa. [auteur du livre] Hubert Nguyen. *GPU Gems 3.* s.l. : Addison-Wesley Professional, 2007. Aussi disponible en ligne : http://http.developer.nvidia.com/GPUGems3/gpugems3_ch19.html.

4. **Claver, Dean.** Deferred Lighting on PS 3.0 with Hight Dynamic Range. [auteur du livre] Charles River. *Shader X³.* s.l. : Charles River Media, 2004, pp. 97-107.

5. **Calver, Dean.** Photo-realistic Deferred Lighting. *Beyond3D.* [En ligne] 2003. http://www.beyond3d.com/content/articles/19.

6. *6800 Leagues Deferred Shading.* **Harris, Mark.** 2002. http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf.

7. *Deferred Rendering in Killzone 2.* **Valient, Michal.** Brighton : s.n., 2007. http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf.

8. **Trebilco, Damian.** *Light Indexed Deferred Lighting.* 2008. http://lightindexed-deferredrender.googlecode.com/files/LightIndexedDeferredLighting1.1.pdf.

9. **Hurley, Kenneth.** Rendering Techniques - Overcoming Deferred Shading Drawbacks by Frank Puig Placeres. [auteur du livre] Wolfgang Engel. *ShaderX5 - Advanced Rendering Techniques.* 2006.

10. **Bavoil, Kevin Myers et Louis.** *Stencil Routed A-Buffer.* 2007. http://www.sci.utah.edu/~bavoil/research/kbuffer/StencilRoutedABuffer_Sigg07.pdf.

11. *Prelighting.* **Lee, Mark.** San Francisco : s.n., 2009. http://www.insomniacgames.com/tech/articles/0409/files/GDC09_Lee_Prelighting.pdf.

12. **Quilez, Iñigo.** Screen Space Ambient Occlusion. *iquilezles.org.* [En ligne] 2008. http://iquilezles.org/www/articles/ssao/ssao.htm.

13. **Mittring, Martin.** Finding Next Gen – CryEngine 2. [auteur du livre] SIGGRAPH. *Advanced Real-Time Rendering in 3D Graphics and Games Course.* 2007. http://delivery.acm.org/10.1145/1290000/1281671/p97-mittring.pdf?key1=1281671&key2=9942678811&coll=ACM&dl=ACM&CFID=15151515&CFTOKEN=6184618.

14. **Engel, Wolfgang.** Light Pre-Pass Renderer. *Diary of a Graphics Programmer.* [En ligne] 2008. http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html.