Ombres douces en temps réel

Audric Schiltknecht

Automne 2009

Résumé

Ce projet consiste en l'implémentation d'un algorithme d'ombres, dites « douces », avec une vitesse permetant un rendu en temps réel. Cette technique se base sur [BS02], qui permet d'obtenir des pénombres assez réalistes, bien que physiquement incorrectes, en se basant sur la recherche de voisinage autours d'un pixel.

Nous présentons la démarche qui a permis d'aboutir à une implémentation concrète de cet algorithme, les modifications qui ont été apportées, ainsi qu'une étude comparative de résultats, suivi par une discussion de ces résultats et des choix d'implémentation.

1 Introduction

Une ombre est une zone 3D où la lumière est bloquée par un objet inséré entre la source de lumière et la surface sur laquelle elle se réflechit. Les ombres sont utiles à la visualisation d'une image, car elles apportent de l'information sur la forme, la position et les caractéristiques d'un objet, ainsi que sur la position, intensité et taille des lumières. La figure 1 illustre ce propos : sur le rendu 1(a), il est difficile de situer les sphères par rapport au plan, tandis que le rendu 1(b) montre clairement que la sphère du bas est quasiment collée sur le plan.

Il existe deux types d'ombres :

1. Les ombres dites *dures* : « binaires », un pixel est soit illuminé, soit dans l'ombre.
2. Les ombres dites *douces* : présence de pénombre, zone de transition entre une zone éclairée et une zone ombrée

Diverses classes d'algorithmes se proposent de traiter le problème des ombres.

1.1 Les *Shadow volumes*

Proposée par Crow ([Cro77]), l'approche consiste à générer une ombre polygonale à partir des objets occultants de la scène, puis à afficher uniquement certaines parties de ces volumes, classiquement, celles comprises entre les faces avant et arrière des volumes d'ombres, tel qu'illustré sur la figure 2.

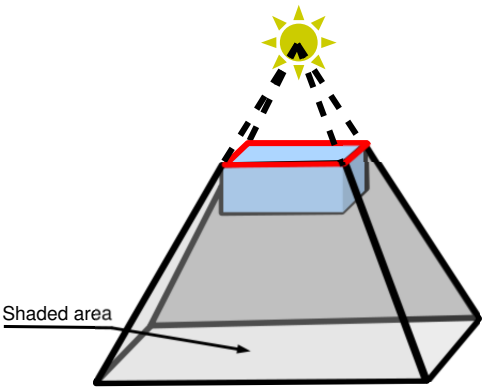


Figure 2 – Principe des Shadow Volumes

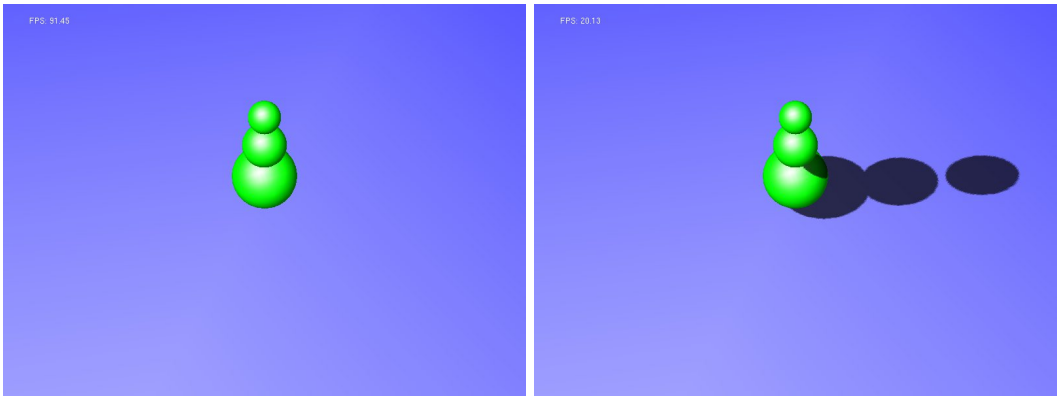
Les avantages d'une telle technique sont :
- Facilement utilisable sur GPU (avec utilisation du *stencil buffer*)
- La source de lumière peut être omni-directionnelle
- Possibilité pour des objets de projetter des ombres sur eux-même (*self-shadowing*)

Parmi les inconvénients, citons :
- Principalement développé pour des géométries polygonales
- Besoin d'informations supplémentaires pour le calcul des silhouettes, lors de la génération des volumes d'ombre
- Pas de gestion des objets transparents

1.2 Les *Shadow Maps*

Décrit par Williams ([Wil78]), le principe consiste en l'utilisation d'un *Z-Buffer* pour déterminer les zones d'ombres. Le déroulement de l'algorithme se fait en deux grandes étapes :

1. On se place du point de vue de la lumière, et l'on réalise le Z-Buffer.



(a) Sans ombre (b) Avec ombres

FIGURE 1 – Une scène rendue sans et avec des ombres

2. On se place ensuite du point de vue de l'œil. Chaque pixel à rendre va être projeté dans le plan de la lumière, et l'on compare la distance du pixel à la lumière avec celle stockée dans le Z-Buffer. Si la distance entre le pixel et la lumière est plus grande que la valeur contenue dans le Z-Buffer, alors le pixel est dans l'ombre ; sinon il est dans la lumière (test de profondeur).

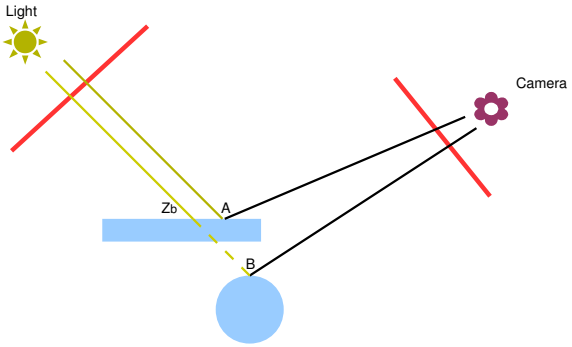


FIGURE 3 – Principe des Shadow Map

Ainsi, sur la figure 3, le point A sera éclairé, tandis que le point B sera dans l'ombre, étant donné que sa profondeur du point de vue de la lumière est plus importante que celle stockée dans la depth map (Z_b).

Cette technique possède des avantages non-négligeables :
- La théorie est simple.
- Tout l'algorithme se base sur le Z-Buffer et le test de profondeur, qui sont implémentables sur hardware, et donc rapides.

Néanmoins, elle souffre des inconvénients suivants :
- Présence d'aliassage dû au Z-Buffer
- Complication avec les lumières sont autres que ponctuelles et directionnelles
- Ne permet que de générer des ombres dures

1.3 Les *Ray-Tracers*

Whitted ([Whi80]) propose une approche extrêmement simple, basée sur les ray-tracers : un rayon est lancé d'un point de la scène vers la lumière. Si ce rayon intercepte un objet avant d'arriver à la lumière, alors le point est dans l'ombre. Cependant, les ray-tracers sont principalement implémentés sur CPU, et sont très couteux, donc inutilisables pour des applications en temps réel.

2 État de l'art

2.1 Choix de l'algorithme

Il existe une grande quantité d'algorithmes traitants des problèmes d'ombres, compilés dans [WPF90] ainsi que plus récement dans [CCOD+09]. Dans une première étape, nous avons décidé de traiter les algorithmes de génération d'ombres douces, avec pour objectif des algorithmes utilisables dans des rendus en temps réel.
Nous nous sommes donc intéressé à l'article de Brabec et Seidel, publié dans [BS02]. Dans leur article, les auteurs modélisent la pénombre comme une fonction de la distance du point considéré à un point ombré. Cette approche, qui paraît assez simple, semble fournir de bons résultats, que ce soit d'un point de vue esthétique ou rapidité de rendu. Nous avons voulu vérifier par nous même ces propos.

2.2 Description

Le principe de cet algorithme est de simuler la décroissance de l'intensité lumineuse en un point comme fonction de la distance de ce point à son plus proche bloqueur, ainsi que de la différence entre la profondeur du bloqueur et du receveur. Pour éviter que la recherche prenne trop de temps (point « loin » d'un bloqueur par exemple), il est possible de fixer une distance maximale à la recherche du plus proche bloqueur, au delà de laquelle on considèrera que le point est pleinement éclairé. Inversement, pour les points situés dans l'ombre, l'algorithme cherche le plus proche point

Pour savoir si un point est dans l'ombre ou est éclairé, on se servira de la définition de base des Shadow Maps, à savoir comparer la profondeur du point à étudier lorsqu'il est projetté dans le plan de la lumière avec celle stockée dans la carte des profondeurs. éclairé.

Cette idée est illustrée sur la figure 4 : le point P est projetté en P' dans le plan de la lumière. On effectue la recherche du plus proche pixel bloqué, qui se trouve au rayon r, $r < r_{max}$.

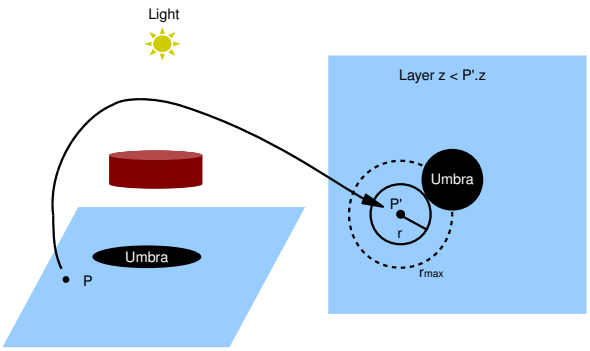


FIGURE 4 – Projection et recherche du plus proche pixel bloqué

L'approche des auteurs ne se veut pas rigoureuse d'un point de vue physique. En effet, de nombreuses approximations se font au niveau du calcul des zones de pénombres. La zone de pénombre d'une ombre douce est principalement fonction des deux distances suivantes :

– la distance entre le bloqueur et le receveur
– la distance entre la lumière et le receveur

Le pseudo code 1 décrit l'algorithme utilisé. Divers paramètres sont utilisés afin de contrôler le rendu :

r_{scale}, r_{biais} contrôlent la zone de recherche des points voisins. En particulier, r_{biais} peut être utilisé pour forcer une certaine pénombre de receveurs proches de la lumière, puisque l'on forcera l'algorithme à effectuer une recherche autours de ces points.

r_{shrink} permet de jouer sur l'effet du facteur construit par la distance entre l'occludeur et la surface, paramètre de la pénombre.

r_{fact} paramètre l'attenuation produite aux zones de jonctions entre les pénombres internes et externes.

Algorithm 1 Soft Shadows Depth Map Algorithm

Pour Tout (x, y) **Faire**
  $P \leftarrow (x, y, z)$
  $P' \leftarrow$ warp_to_light(P)
  $inner \leftarrow$ depth_map$[P'_x, P'_y] < P'_z$
  $f \leftarrow$ MODULATE$(P', inner)$
  modulate pixel by f
Fin Pour

Fonction MODULATE ($P', inner$)
  $r \leftarrow 0$
  $r_{max} \leftarrow r_{scale} * |P'_z| + r_{biais}$
  Tant que $r < r_{max}$ **Faire**
    Si $inner$ **Alors**
      ▷ Cherche le pixel éclairé le plus proche
      $Point \leftarrow$ search_near_lit(P', r)
      Si $Point \neq$ NULL **Alors**
        $r_{max}* \leftarrow r_{shrink} * (Point.z - P'.z)$
        $f \leftarrow r_{fact} * \text{clamp}_{[0,1]}(r/r_{max})$
        Retourner f
      Fin Si
    Sinon
      ▷ Cherche le pixel bloqué le plus proche
      $Point \leftarrow$ search_near_blocker(P', r)
      Si $Point \neq$ NULL **Alors**
        $r_{max}* \leftarrow r_{shrink} * (P'.z - Point.z)$
        $f \leftarrow 1 - r_{fact} * \text{clamp}_{[0,1]}(r/r_{max})$
        Retourner f
      Fin Si
    Fin Si
    r++
  Fin Tant que
  Retourner $inner$? 0.0 : 1.0
Fin Fonction

En pratique les valeurs des différents paramètres ont été fixées à :

$r_{scale}, r_{biais} = 2.0$

$r_{shrink} = 1.0$

$r_{fact} = 0.5$

3 Implémentation

La base de l'algorithme se situe en la réalisation de la shadow map. Pour cela, nous sommes partis du code trouvé sur le tutoriel à l'adresse suivante : http://www.paulsprojects.net/tutorials/smt/smt.html, qui nous a servi de patron pour les grandes lignes de l'algorihtme. Ce code utilise les extensions *ARB* d'OpenGL pour effectuer les tests de profondeur. Pour nous épargner la phase d'initialisation des pointeurs de fonctions des extensions, nous avons utilisé la bibliothèque de chargement *GLee*, de Ben Woodhouse, disponible à cette adresse http://elf-stone.com/glee.php.

Nous avons ensuite adapté le code pour pouvoir utiliser un *Frame Buffer Object* (FBO) pour stocker les valeurs de notre carte de profondeur. En effet, initialement, les valeurs étaient stockées dans une texture simple. Or, si les dimensions de la shadow map, c'est-à-dire de la texture, ne sont pas les mêmes que celle de la fenêtre d'affichage, il se produit un « décalage » des ombres. Pour nous permettre d'utiliser une taille de depth map indépendante de celle de l'affichage, nous utilisé en place d'une texture un *FBO*, qui possède l'avantage de permettre de faire des rendus dits hors-écran.

3.1 Création de la depth-map et projection des points

La création de la carte de profondeur se fait simplement de la façon suivante : nous paramétrons la vue à partir de la lumière, et effectuons un rendu en Z-Buffer de la scène. Ce rendu sera stocké dans le FBO spécifiquement alloué à cet effet.

Pour réaliser cette étape, il nous faut utiliser un *depth offset*. En effet, le Z-buffer est soumis à des problème de *Z-Fighting* : si deux primitives sont trop proches, elles risquent d'avoir des valeurs similaires dans le Z-buffer. Cela entraîne des artéfacts à ces endroits. Pour éviter cela, nous utilisons donc le *Polygon Offset*, afin de « décaler » virtuellement les objets, et de permettre un rendu sans artéfacts.

Ensuite, nous effectuons l'étape de *warping* : nous calculons une matrice de passage de l'espace de la caméra vers l'espace de la lumière. Cette opération se fait par une suite de multiplication matricielle. Nous en profitons pour ajouter une « matrice de biais » : en effet, les coordonées obtenues suite à la projection seront comprises dans l'intervalle $[-1, 1]$. Or, les coordonnées utilisées en OpenGL doivent être comprise dans $[0, 1]$. Nous nous ramenons donc à cet intervalle via la matrice de biais. Cette matrice de projection est stockée sous la forme d'une texture et sera utilisée dans la suite de l'algorithme.

3.2 Calcul des ombres

Brabec et Seidel effectuent certains calculs (warping dans le plan de la lumière, recherche des bloqueurs) sur le CPU de la machine. Nous avons décidé d'adopter une approche différente, et de faire effectuer ces calculs sur le GPU. En effet, nous pensons ainsi gagner sur le goulot d'étranglement que constitue le bus entre le CPU et le GPU, lors de la récupération de la shadow map. Pour pouvoir travailler sur le GPU, nous allons utiliser le langage dédié proposé par OpenGL : GLSL, « OpenGL Shader Language ».

3.2.1 Les *Shaders*

Pour cela, il nous faut définir deux « programmes », plus précisement *shaders*, qui seront exécutés par la puce graphique. Ces programmes sont les *Vertex* et *Fragment Shaders*.

Vertex Shaders Ils sont exécutés lors du traitement de chaque sommet des primitives. Dans notre cas, ils permettront de calculer les coordonnées projetées dans le plan de la lumière, grâce à la matrice de projection précédente, ainsi que les informations de couleur.

Fragment Shader Encore appelés *Pixel Shader*, ils interviennent pour le traitement de chacun des pixels à afficher. C'est dans ces shaders que se concentre le gros de l'algorithme. Nous y effectuerons la recherche de bloqueurs/points éclairés, ainsi que la modulation de la couleur (et donc l'effet d'ombre) calculée dans le Vertex Shader.

3.2.2 Recherche de points

Lors de la recherche du plus proche pixel bloqué (resp. éclairé), nous effectuons une recherche linéaire, alignée sur les axes. Pour éviter de parcourir une quantité astronomique de pixels n'intervenant pas dans le calcul, nous vérifions au préalable que la distance entre le pixel traité et celui qui vient d'être trouvé est approximativement égale au rayon de recherche. Approxima-

tivement, car les coordonnées du pixel traité, considéré comme le centre de notre recherche, sont des nombres flottants. De fait, lorsque nous effectuons la recherche autours de ce pixel, nous échantillonons la depth map tous les $\frac{1}{r}$, où r est la résolution de la depth map.
La recherche se termine lorsque l'on trouve un pixel dont la profondeur dans la depth map est inférieure (resp. supérieure) à celle du point centre la recherche.

4 Résultats

4.1 Quelques rendus

Tous les résultats présentés sont obtenus, sauf mention contraire, pour une résolution de 800×600 pixels, avec des shadow maps de taille 1024×1024 pixels, sur une carte nVidia GeForce 8600M GT.

La figure 5 représente une partie d'une scène. Nous pouvons voir la présence de légères zones de pénombre, entourant les ombres dures. Une constatation s'impose : la différence avec un rendu en ombres dures n'est pas immédiate. Nous avons donc essayé de jouer avec les paramètres r_{scale} et r_{biais}, mais sans aucun effet visible (autre qu'un temps de rendu considérablement allongé).

Sur la figure 6 sont illustrés les divers types d'ombres mis en jeu. La figure 6(a) est obtenue avec des ombres dures. La figure 6(b) représente en blanc la pénombre obtenue en cherchant les pixels exposés, et la figure 6(c) celle obtenue en cherchant les pixels bloqués. Cette figure prouve bien qu'il existe bien des zones de pénombres. Elles ne sont en revanche pas du tout conforme à ce que la théorie et le bon sens nous indiquent.

La figure 7 est obtenue avec une lumière éloignée des objets. Théoriquement, la zone de pénombre devrait être assez importante. Le rendu nous montre bien que ce n'est pas le cas, bien que l'on puisse remarquer que l'ombre projetée sur le tore par la sphère au premier plan est plus importante. La présence d'un (fort) aliasage, sur le plan en particulier, est à noter.

Les 3 rendus présentés figure 8 illustrent l'effet de la résolution de la depth map sur le rendu. Plus la résolution est faible, plus il y a aliasage. Cela est encore une fois dû à l'opération de rasterization effectuée lors du Z-Buffer.

4.2 Vitesse de rendu

L'un des objectifs de cette étude est de réaliser un algorithme d'ombres en temps réel. Tentons de voir si l'algorithme implémenté permet de faire un rendu en un temps acceptable.

La scène utilisée dans les exemples précédents (figure 7) se rend avec un framerate de ≈ 440 FPS en utilisant les extensions ARB (en ombres dures donc). Si l'on utilise notre implémentation, la vitesse de rendu tombe à ≈ 25 FPS. Cette vitesse est tout à fait acceptable, et permet des interactions de l'utilisateur, qu'il s'agisse de déplacement de caméra ou de la lumière.
Bien entendu, la vitesse de rendu va dépendre de la distance de la caméra à la scène. Ainsi, pour une scène plus complexe, mais réalisée d'un point de vue éloigné comme la figure 9(a), nous pouvons obtenir un rendu à ≈ 40 FPS. Si l'on rapproche la caméra d'une zone d'ombre, le framerate tombe immédiatement à ≈ 11 FPS (figure 9(b)).

4.3 Limitations

Notre algorithme ne fonctionne que pour une seule lumière, ponctuelle, de type « projecteur ». Il serait possible de l'adapter pour pouvoir traiter plusieurs lampes. Pour cela, pour chacune des lampes, il faut créer une shadow map, et effectuer le calcul d'ombre. Ensuite, en combinant les résultats, il est possible d'obtenir les ombres pour l'ensemble des lumières.

Pour des lumières omni-directionnelles, il est possible de traiter le problème en utilisant des *Cube Shadow Maps* : la scène est rendue dans les 6 directions de l'espace, et les depth maps sont stockées comme les faces d'un cube.

Enfin, pour le cas des lumières non-ponctuelles (néon, par exemple), le problème peut être ramené au cas de plusieurs lumières par échantillonage.

D'autre part, le principal défaut inhérent au shadow map est l'aliasage dû à l'emploi du Z-Buffer. Pour minimimiser cet artéfact, plusieurs solutions s'offrent à nous. La première est d'augmenter la taille des données stockées dans une depth map. Cette contrainte est cependant matérielle, puisque la profondeur sera codée sur 16, 24 ou 32 bits selon les cartes graphiques. La seconde est de bien positionner les plans de clipping de la lumière (les valeurs *ZNear* et *ZFar*). En effet, en tentant d'attribuer la plus grande valeur possible à ZNear, et la plus petite pour ZFar, on peut réduire l'intervalle des valeurs prises par la profondeur

entre ces deux plans, et ainsi gagner en précision. Cependant, ces valeurs dépendent fortement de la scène (géométrie, position des lumières).

Enfin, il faut noter que lors de l'utilisation de modèle volumineux (scène 3DSMax de quelques milliers de polygones), le comportement du programme tend à devenir assez hasardeux. Des blocages de quelques secondes se font ressentir, allant même jusqu'à la saturation du processeur graphique et à l'arrêt brutal du programme. Nous n'avons pas pu déterminer l'origine de ces comportements.

4.4 Discussions des choix de réalisation

Un avantage de l'utilisation de GLSL en place d'autres langages de shader (*Cg*[1], par exemple) est, qu'en plus d'une portabilité accrue, la compilation est effectuée par le driver. Ainsi, des optimisations peuvent être effectuées automatiquement, en fonction du matériel disponible sur la machine.

Nous avons évoqué précédement que le choix de l'utilisation de Shaders était dû au fait que tous les calculs nécessitant des accès à la mémoire graphique se font sur le processeur graphique. Il n'y a donc pas de transfert prohibitif entre le GPU et le CPU (pour récupérer la shadow map) d'une part, puis entre le CPU et le GPU (pour envoyer la texture stockant les ombres).
Il faut cependant nuancer ce point. En effet, nous économisons des temps d'accès à la mémoire (1 accès en lecture et 1 en écriture par frame). Cependant, lors de la recherche de voisins, nous allons effectuer de nombreux lookups dans la depth map. Comparativement, un algorithme s'exécutant sur le processeur n'aurait à effectuer que des lookups dans une matrice, opération extrêmement rapide à réaliser. Dès lors, il pourrait être intéressant de comparer le coût d'une lecture en mémoire graphique pour récuperer notre depth map à celui d'un lookup réalisé dans les shaders, et voir si l'une des deux méthodes possède un réel avantage de vitesse.

5 Conclusion et améliorations

Nous avons donc implémenté un algorithme d'ombres douces, capable d'effectuer des rendus en temps réel. Des modifications ont été apportées à l'algorithme initial, puisque nous avons souhaité l'implémenter sur le processeur graphique.

Bien que les résultats ne soient pas à la hauteur des attentes concernant l'esthétique, ils sont quand même visibles. Néanmoins, malgré les quelques bogues évoqués, cela constitue une première approche de la réalisation d'un algorithme d'ombrage en temps réel.

Nous n'avons pas implémenté le mécanisme d'échantillonage adaptatif, expliqué par les auteurs. Ce système permet de réduire les recherches de voisins dans certains cas, en faisant l'hypothèse que sur des zones où les pixels sont ombrés par un même objet, et dont le résultat de test de profondeur sont les mêmes, l'ombre ne subit pas de variation brusque. Il est donc possible de l'interpoler dans ces zones spécifiques. Les auteurs annoncent une division par 16 du nombre de recherches effectuées dans des cas favorables.

1. Langage de programmation de shaders créé par nVidia

6 Remerciements

Je tiens à remercier Anthony Pajot, qui s'est toujours montré présent et attentif à mes nombreuses questions en tant que débutant du domaine, ainsi que pour ses nombreuses critiques.

Références

[BS02] Stefan Brabec and Hans-Peter Seidel. Single sample soft shadows using depth maps. In *Graphics Interface*, pages 219–228, 2002.

[CCOD+09] Yiorgos Chrysanthou, Daniel Cohen-Or, Frédo Durand, Claudio T. Silva, Andrew Woo, and Pierre Poulin. Visibility and occlusion : Rendering Acceleration and Shadow Computation. 2009.

[Cro77] F.C. Crow. Shadow algorithms for computer graphics. *ACM SIGGRAPH Computer Graphics*, 11(2) :248, 1977.

[Whi80] T. Whirred. An improved illumination model for shaded display. *Communications of the ACM*, 23(6) :343–349, 1980.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3) :270–274, 1978.

[WPF90] Andrew. Woo, Pierre Poulin, and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6) :13–32, 1990.

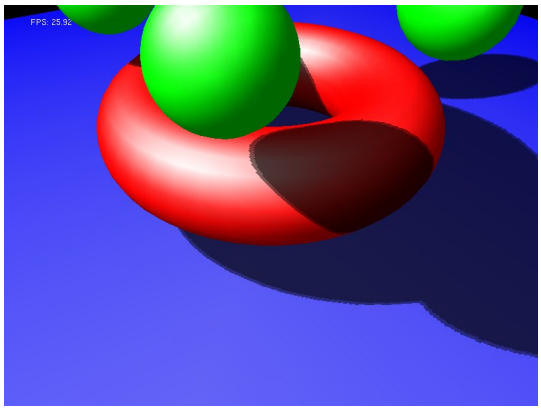
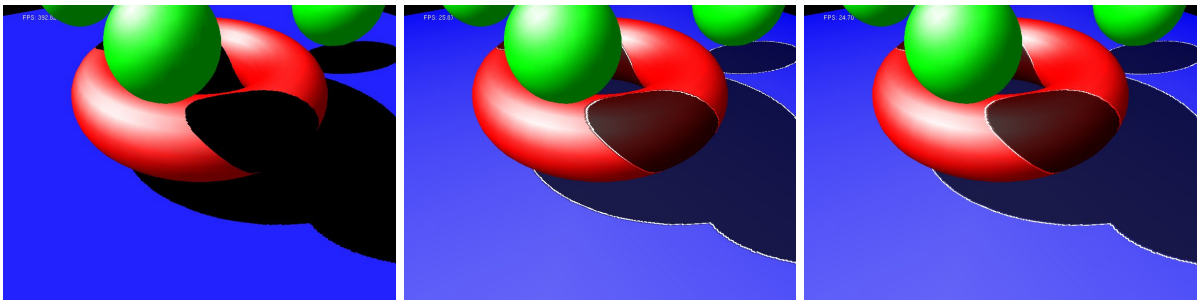


FIGURE 5 – Un exemple de rendu



(a) Hard Shadows (b) Soft Shadows - Internes (c) Soft Shadows - Externes

FIGURE 6 – Les diverses ombres

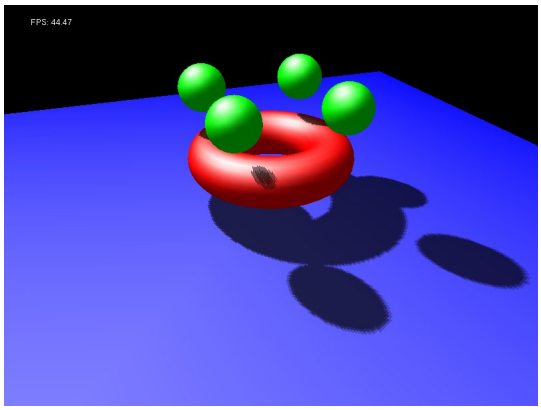
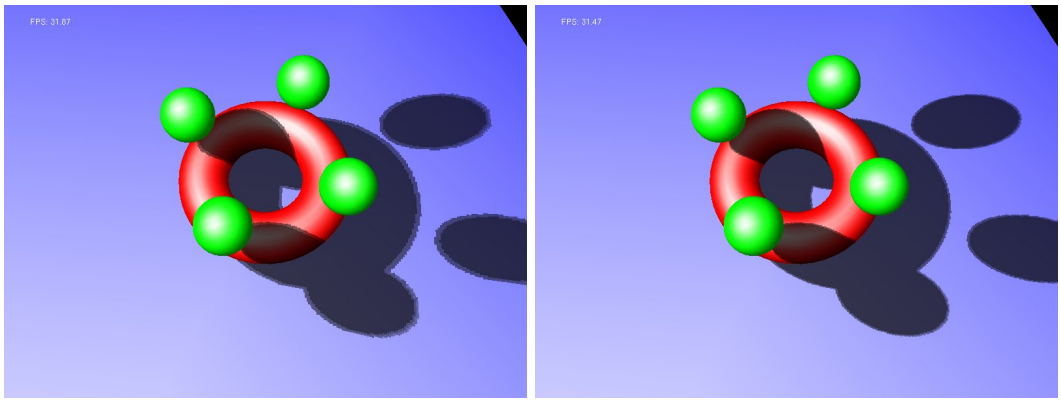
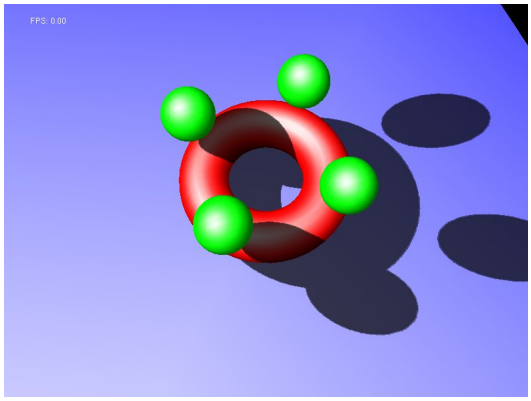


FIGURE 7 – Lumière éloignée

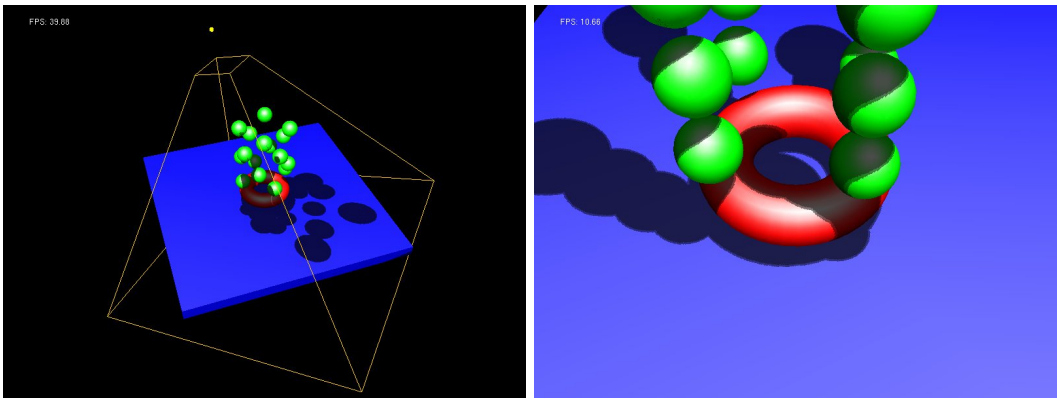


(a) 512 pixels (b) 1024 pixels



(c) 2048 pixels

FIGURE 8 – Diverses résolutions de Shadow Depth Map



(a) Caméra éloignée (b) Caméra proche

FIGURE 9 – Vitesses de rendu