TP Shadow Mapping

Malek.Bengougam@gmail.com

Projection de texture

Le Shadow Mapping est une extension d'une technique appelée projection de texture.
Cette dernière simule la projection d'une texture sur une surface en prenant en compte le point de vue du projecteur, le plus souvent une lumière positionnelle de type spot.

Le principe est le suivant : on créé deux matrices, view et projection, comme pour une caméra.
Ces matrices vont nous permettre de simuler un point de fuite depuis le point de vue du projecteur.

On va ensuite utiliser la concaténation de ces deux matrices dans le shader de la surface pour calculer des coordonnées de texture qui vont nous servir à sampler la texture du projecteur.

Il faut noter que la matrice view-proj produit des valeurs exprimées dans le repère post-projection (NDC) tandis que les valeurs attendues par les fonctions de sampling de texture doivent être normalisées entre [0, 1].

Il suffit de concaténer une autre matrice qui va transformer les valeurs exprimées entre [-1, +1] vers [0, 1]. Cette matrice est appelée matrice d'offset.
Il faudra passer la concaténation de ces matrices en tant que variable uniforme du shader de la surface, ainsi qu'assigner un sampler2D correspondant à la texture.

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & & & \frac{1}{2} \\ & \frac{1}{2} & & \frac{1}{2} \\ & & \frac{1}{2} & \frac{1}{2} \\ & & & 1 \end{bmatrix} \begin{bmatrix} Light \\ Frustum \\ (projection) \\ Matrix \end{bmatrix} \begin{bmatrix} Light \\ View \\ (look\ at) \\ Matrix \end{bmatrix} \begin{bmatrix} \\ Modeling \\ Matrix \\ \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ w_0 \end{bmatrix}$$

On va alors transformer les vertices de la surface par cette matrice. Le résultat va nous permettre de sampler la texture, cependant le résultat est un vecteur homogène (4D) tandis que la fonction texture() ne prend qu'un vec2.

Il faut utiliser une fonction glsl particulière qui est **textureProj**(). Cette fonction effectue la division perspective avant de sampler simulant ainsi le point de fuite.

Projection opposée (back projection)

Un problème arrive souvent : la texture est projetée dans les deux directions, vers l'avant et vers l'arrière. Ceci est dû à la logique des coordonnées homogènes.
Comme les valeurs xyz sont d'abord divisées par w, ceci implique que xyz/w produit la même valeur lorsque xyz sont tous trois négatifs et que w est négatif, que lorsque xyz sont tous trois positifs et que w est positif.
La façon la plus aisée de fixer ce problème est de ne pas sampler la texture projetée lorsque la valeur de w est négative.

Depth Texture

Exemple : création d'une depth texture de type float 32 bits attachée sur un FBO. Notez l'appel à **glDrawBuffer**(GL_NONE) pour indiquer à OpenGL qu'il est normal que le FBO n'ait pas de color buffer.

```
GLuint shadowmapTexture;
GLuint shadowmapFBO;
const int shadowmap_resolution = 1024;
GLenum shadowmap_precision = GL_DEPTH_COMPONENT32; // 24 ou 16
GLenum shadowmap_type = GL_FLOAT; // GL_UNSIGNED_INT

glGenTextures( 1, &shadowmapTexture );
glBindTexture( GL_TEXTURE_2D, shadowmapTexture );
glTexImage2D(GL_TEXTURE_2D, 0, shadowmap_precision, shadowmap_resolution,
shadowmap_resolution, 0, GL_DEPTH_COMPONENT, shadowmap_type, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glGenFramebuffers(1, &shadowmapFBO);
glBindFramebuffer(GL_FRAMEBUFFER, shadowmapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
shadowmapTexture, 0);
glDrawBuffer(GL_NONE); // tres important, autrement FBO non complet !
//glReadBuffer(GL_NONE); // optionnellement

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindTexture( GL_TEXTURE_2D, 0);
```

Au moment de dessiner dans le FBO, il faut penser à redimensionner le viewport pour tenir compte des dimensions de la depth texture. Il est possible de demander à OpenGL de stocker les anciens paramètres de **glViewport**() pour les rétablir par la suite.

```
glBindFramebuffer(GL_FRAMEBUFFER, shadowmapFBO);
glPushAttrib(GL_VIEWPORT_BIT); // optionnellement
glViewport(0,0,shadowmap_resolution, shadowmap_resolution);
```

Ne pas oublier de rétablir les anciennes valeurs du viewport correspondant à la taille du canvas. Si vous avez utilisé **glPushAttrib**() il faudra alors utiliser **glPopAttrib**(). Ce mécanisme n'est pas obligatoire, et, dans le cas où une autre API qu'OpenGL serait utilisé sur une autre plateforme, il serait plus logique de stocker explicitement ces informations.

Shadow Mapping

1. Créer une shadow map consiste à générer une Depth Texture depuis le point de vue de la lumière. On va utiliser une variante particulière de texture qui utilise un sampler « shadow ».
   La particularité est qu'un sampler « shadow » effectue automatiquement la **comparaison** de profondeur. On indique cela en ajoutant deux paramètres supplémentaires à notre texture :

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
```

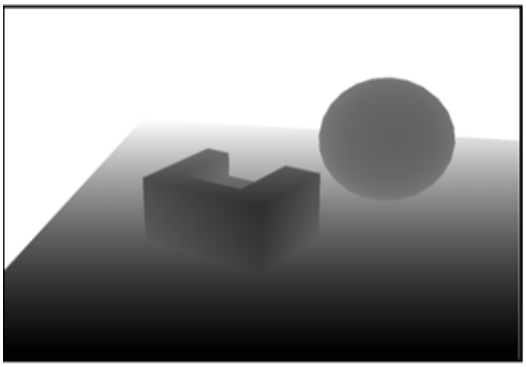
*Note : le type correspondant en GLSL est **sampler2DShadow** (en 2D et OpenGL 2.x).*

La valeur retournée est donc un float et non pas un vec4 rgba !

*Si vous ne souhaitez pas que le hardware effectue la comparaison, utilisez un classique **sampler2D**.*

2. Comme pour la projection de texture on va devoir créer une matrice view et une matrice projection pour la lumière. Le choix de la matrice de projection est important car cela aura un impact sur la façon dont les ombres vont être générées.

Note : la règle de base est qu'il faut utiliser une projection orthographique dans le cas des lumières directionnelles et une projection perspective dans le cas des positionnelles (spots ou omnidirectionnelles çad point light).

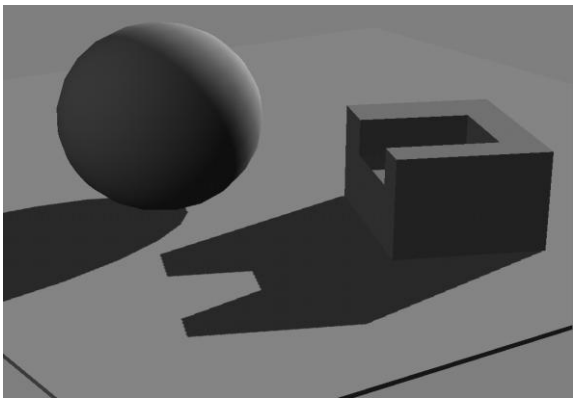


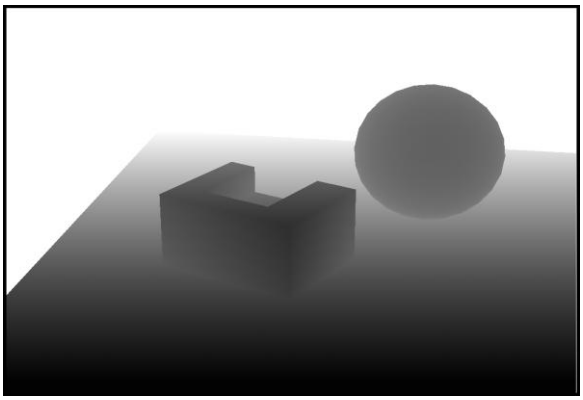
Shadow Map (light's view)

Idéalement, il ne faudrait dessiner dans cette passe que les objets visibles depuis le point de vue de la lumière.

3. Ne pas oublier de remettre le framebuffer dans lequel vous souhaitez effectuer le rendu et réattribuer les bonnes valeurs à **glViewport**() – après avoir switché de buffer sinon **glViewport**() ne sera pas pris en compte.

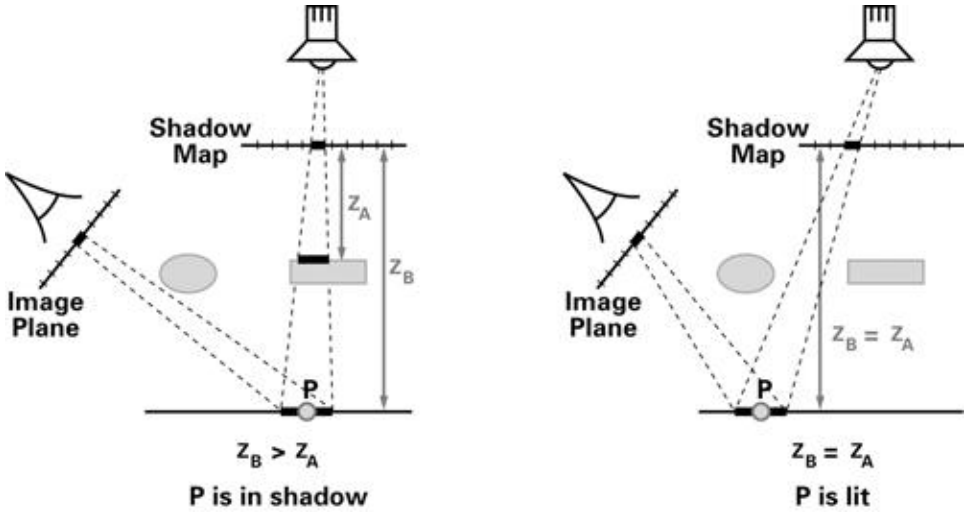
4. On a maintenant une texture contenant des valeurs de profondeur définies dans le repère de la lumière. Or on souhaite déterminer quels fragments sont dans l'ombre mais dans une passe de rendu principale qui s'effectue, elle, du point de vue de la caméra.





On va donc devoir comparer des valeurs qui se trouvent dans deux repères différents. Le plus simple est alors de passer les matrices view et projection de la lumière au shader et de transformer la position monde d'un vertex de l'objet en coordonnées post-projeté (NDC).

Ces coordonnées dans le repère de la light vont alors nous servir de coordonnées de texture afin de sampler la depth map avec **texture2D**() ou **texture2DProj**() (ou **texture**() et **textureProj**() en GL3+).



*Rappel : on utilise **textureProj**() en théorie si la matrice de projection est perspective. La fonction attend un **vec3** homogène xyw, et effectue donc la division de xy par w.*
Cette opération peut également se faire manuellement si la fonction n'existe pas.

Comme dans le cas des textures projetées il ne faut pas oublier que les coordonnées de textures sont normalisées et positives [0, 1] alors que les valeurs en NDC sont entre [-1, +1] sur les 3 axes.

Problématiques du shadow mapping

Problème de moirage (Moire pattern)

Solution classique pour le Z-fighting : introduire un bias. Dans le shader à l'aide d'une valeur constante ou passé en uniform, ou bien via **glPolygonOffset**(). Cependant, on préfère souvent spécifier une valeur de biais dans le shader directement (en dur, ou variable uniform).

Effet « Peter Pan »

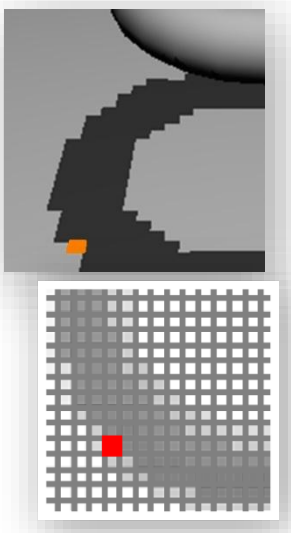
La principale problématique du bias est qu'il risque de produire un effet où l'ombre est détachée de l'objet (d'où la référence à Peter Pan).
Une solution simple à ce problème : n'afficher que les faces arrière (donc *culling* des faces avant lors du rendu dans la shadow map). Cependant ce n'est pas idéal lorsque les polygones sont « fins » (par exemple si un mur n'est pas un volume mais un simple plan ou quad).

Cela dit, le bias est ici fixe pour toute la scène, ou pour tout un objet, or si les surfaces de l'objet ne sont pas plates on a donc un bias qui ne tient pas compte de la perspective et donc des différentiels (en termes de pente) entre les fragments.
Le hardware dispose de fonctionnalité pour corriger cela, mais on peut émuler la fonctionnalité sur d'autres GPU plus ancien en calculant un facteur de correction en fonction de la pente.

Effet de bordure

Le fait que la texture soit limitée en dimension fait que l'on ne peut stocker toute la scène. Du coup certains fragment qui sont en dehors des limites de la zone de la lumière peuvent par inadvertance être considéré dans l'ombre. Ceci arrive particulièrement si le mode de wrapping de la texture est GL_REPEAT. On a légèrement corrigé cela en spécifiant **GL_CLAMP_TO_EDGE** pour notre depth texture.
Cependant cela signifie que toute coordonnée au delà de [0,1] sera clampée à [0,1]. Donc si aux bordures de la texture il y'a des ombres, tout objet ne tenant pas dans la texture sera ombré.
Une solution consiste à forcer la couleur de la bordure à une intensité blanche (donc non ombrée). Cela se fait via le paramètre de texture **GL_TEXTURE_BORDER_COLOR** qui prend un pointeur vers 4 float normalisés. De plus on peut aussi remplacer **GL_CLAMP_TO_EDGE** par **GL_CLAMP_TO_BORDER** pour forcer plus concrètement ce comportement.

Aliasing



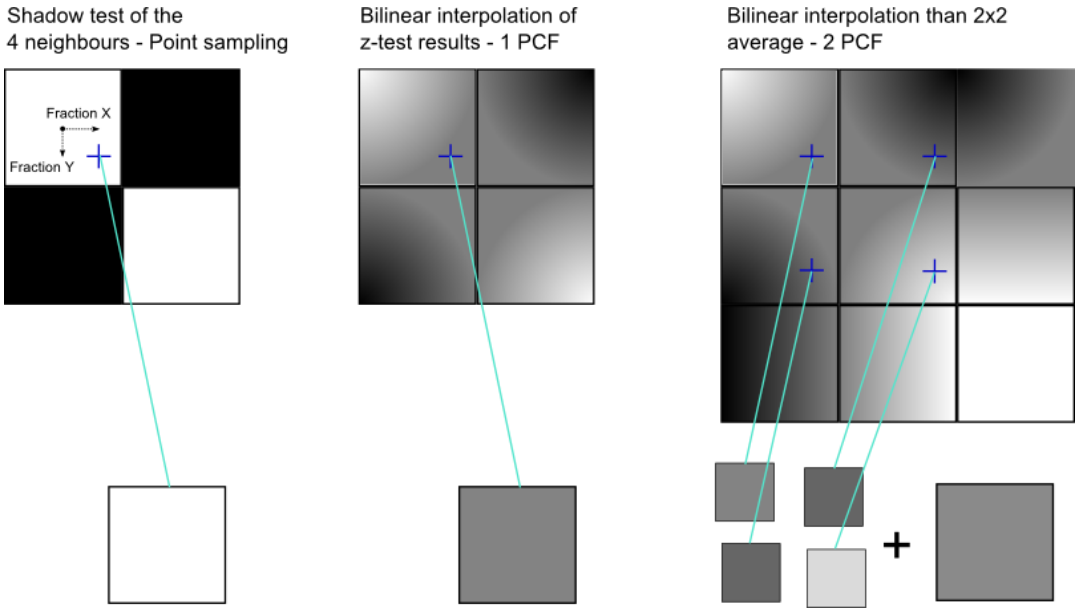
Visuellement c'est surtout la faible résolution, ou densité de pixels dans la shadow map, conjuguée au fait que le test soit binaire (dans ou hors de l'ombre) qui est le plus frappant. En effet, on obtient un effet de crénelage que l'on peut compenser en augmentant la résolution de la depth texture avec le risque d'augmenter la charge sur la bande passante et donc de ralentir les performances.

Plusieurs techniques ont été mis en place pour combattre ce problème, à commencer par la tentative d'adoucir les contours des ombres et tenter de simuler une pénombre (soft shadows) et donc d'avoir des valeurs d'ombrage plus graduelles.

Une option serait de flouter (bilinéaire ou fitre plus complexe) le contenu de la shadow map. Ce qui peut être utile mais mais pour résoudre ce problème.

Solution standard pour contrer l'aliasing : Percentage Close Filter (PCF)

Une depth texture contient des données de profondeurs normalisées ce qui signifie que l'on ne peut pas simplement utiliser un filtre de flou prévu pour les couleurs.



0.49	0.5
0.5	0.51

Une possibilité serait de reprendre l'idée d'un box blur (bilinéaire) mais appliqué à la valeur retournée par texture2D(). Par exemple, pour les 4 pixels à gauche ici, 1 seul est dans l'ombre. Le PCF consiste à pondérer la valeur d'ombre (qui ne sera plus binaire, 0 = ombre, 1 = lumière) par le ratio nombre de pixels ombrés / pixels total (1/4 ici) mais également par le différentiel entre les valeurs des 4 pixels.

Le GPU est capable d'effectuer ce calcul automatiquement sur un **sampler2DShadow** (lorsque la comparaison de depth) simplement en utilisant **GL_LINEAR** comme TexFilter et pas **GL_NEAREST**.
Le noyau de filtrage (box blur/bilinear) est ici limité : 2x2. On peut coder manuellement dans le fragment shader un noyau plus large en combinant plusieurs blocs 2x2. Exemple, d'un noyau 4x4 :

```
float sum = 0.0;
sum += texture2DOffset(u_ShadowMap, ShadowCoord, ivec2(-1,-1));
sum += texture2DOffset(u_ShadowMap, ShadowCoord, ivec2(-1,1));
sum += texture2DOffset(u_ShadowMap, ShadowCoord, ivec2(1,1));
sum += texture2DOffset(u_ShadowMap, ShadowCoord, ivec2(1,-1));
float shadow = sum * 0.25;
```

A noter que l'offset est ici exprimé en coordonnées pixels entières. Remarquez que l'on fait une simple moyenne des appels (1/4), ce qui ne correspond pas forcément au nombre réel d'échantillons. Cette fonction va automatiquement récupérer la taille de la texture -*textureSize*() en GL3+- et normaliser. Un meilleur choix serait d'utiliser une taille de 3x3 centrée sur le pixel courant, donc 9 échantillons, mais on va exploiter le filtre bilinéaire (on est entre deux pixels) et le PCF avec le même code que ci-dessus mais avec un offset entre deux texels. On parle alors de filtre en tente.

Exercice : coder un kernel 3x3 L'offset pour ce tent filter doit être de +/-0.5. On ne peut donc pas utiliser texture2DProjOffset() mais calculer l'offset exact en multipliant par la dimension du texel.

Aller plus loin, filtrage avancé

Plusieurs techniques de flou ont été proposées afin d'améliorer le rendu et donner un côté soft shadow. La plupart du temps il s'agit de noyau basé sur des analyses stochastiques (on échantillonne au hasard un certain nombre de points) en suivant un pattern particulier.
On trouve très souvent un PCF dont les offsets sont tirés d'une distribution suivant la loi de Poisson. Par exemple, pour un PCF 2x2, on utilise les offsets suivants (à diviser par un facteur de distribution) :

```
vec2 poissonDisk[4] = vec2[](
  vec2( -0.94201624, -0.39906216 ),
  vec2( 0.94558609, -0.76890725 ),
  vec2( -0.094184101, -0.92938870 ),
  vec2( 0.34495938, 0.29387760 )
);
```

D'autres techniques ont également exploré le côté statistique, on pourrait par exemple ici « stratifier » les échantillons, ou bien calculer la probabilité d'être dans l'ombre en utilisant le concept statistique de « moment » : Variance Shadow Map (VSM) et Exponential Shadow Map (ESM).

Amélioration de la précision

D'autres recherches ont plutôt essayé d'augmenter la précision de la shadow map. Une partie de l'aliasing est du à la projection perspective. Les techniques comme les Perspective Shadow Map (PSM) ou une variante plus optimale les Light-Space Perspective Shadow Map (LiPSM) essaye de corriger ce problème en essayant d'altérer la projection pour avoir le plus de densité proche de la caméra. Par contre, comme le nom l'indique, elles impliquent une projection perspective et sont par essences complètement statique, d'autant que très souvent elles ne fonctionnent pas suffisamment bien pour des grosses scènes et ont tendance à produire une matrice quasiment orthographique ce que la technique de Logarithmic Perspective Shadow Map (LogPSM) essaye de corriger en générant une shadow map pour chaque face du frustum en effectuant un rendu non linéaire qui tient compte de l'aspect logarithmique de la profondeur stockée dans la depth map (Z/W).

Actuellement, la technique recommandée et qui fourni un très bon rendu est le Cascaded Shadow Map (CSM) aussi appelé Parallel Split Shadow Map (PSSM).
Le CSM ou PSSM repose sur l'utilisation de plusieurs shadow maps de tailles différentes (façon mipmap mais sur 3 ou 4 niveaux seulement). Chacune de ces textures est utilisée en cascade, la texture de plus haute résolution stockera les ombres proches de la lumière tandis que la plus petite stockera les ombres les plus éloignées.
On va donc splitter le point de vue de la lumière en plusieurs zones parallèles (mais pas forcément réparties de manière égale), chacune des zones ayant donc ses propres plans near et far et bien sûr sa propre shadow map.

Références

http://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping
http://http.developer.nvidia.com/GPUGems/gpugems_ch14.html
http://http.developer.nvidia.com/GPUGems/gpugems_ch11.html
http://fabiensanglard.net/shadowmappingVSM/
https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324%28v=vs.85%29.aspx
Voir les séries ShaderX, GPU Pro (ou GPU 360) - chapitres sur les ombres