

Gestion dynamique de la lumière avec OpenGL - Partie 2 : le light mapping simple

par [Michel de VERDELHAN](#)

Date de publication : 06/09/2006

Dernière mise à jour :

Ce tutoriel ainsi que celui qui le suit montre la mise en oeuvre d'un système de gestion de lumières dynamiques par light mapping. Ce tutoriel détaille le principe du light mapping et utilise une mise en oeuvre simple à base de textures 2D.

- I - Pourquoi utiliser le light mapping.
- II - Le light mapping : un système simple et puissant.
 - II-1 - Rappels sur le blending.
 - II-2 - Principe du light mapping.
- III - Modifications du programme.
 - III-1 - Gérer les extensions.
 - III-2 - Modifications de la classe de model.
 - III-3 - Modification du calcul de la luminosité.
 - III-4 - Le rendu final
- IV - Conclusions.

I - Pourquoi utiliser le light mapping.

Nous avons vu dans le tutoriel précédent qu'utiliser les vertex pour calculer la luminosité est simple à mettre en oeuvre mais pose des problèmes au niveau de l'atténuation de la lumière sur les polygones étendus. Ce problème peut être contourné en augmentant la tessellation de la scène, mais dans un moteur de jeu cette solution n'est généralement pas utilisable. En effet, augmenter la tessellation va augmenter la géométrie, ce qui va accentuer la charge de travail des algorithmes basé sur cette géométrie (collisions, ombres volumétriques et autres.)

Une solution à ce problème a été trouvée par John Carmack pour le jeu Quake 1. Cette solution se base sur l'utilisation de textures pour stocker la luminosité des surfaces. On utilise donc 2 textures par polygone : une texture standard contenant l'image qu'on souhaite afficher sur notre polygone, et une texture de luminosité appelée light map, d'où le nom de la méthode : le light mapping.

Cette méthode étant basée sur des textures et non plus directement sur la géométrie, les problèmes lors du calcul de l'atténuation sont donc supprimés.

Nous allons voir dans les deux tutoriaux à venir comment mettre en place un système de light mapping.

II - Le light mapping : un système simple et puissant.

Le light mapping est basé sur la modulation de textures. Je vais donc faire un bref rappel sur le blending qui va nous permettre d'utiliser cette modulation de textures. Ensuite je détaillerai le light mapping, puis je vais expliquer les modifications apportées au tutoriel précédent pour obtenir l'effet souhaité.

II-1 - Rappels sur le blending.

L'API OpenGL propose en standard une gestion du blending. Le blending consiste à effectuer plusieurs rendu de la scène et à mélanger les couleurs de la scène rendue précédemment avec celles qu'on est en train de rendre (to blend : mélanger en anglais).

Il existe plusieurs familles de blending. Parmi elles, deux nous intéressent plus particulièrement :

Ces deux modes de blending peuvent s'obtenir en passant à la fonction `glBlendFunc()` les paramètres (`GL_ONE, GL_ONE`) pour le blending additif et (`GL_DST_COLOR, GL_ZERO`) pour le blending multiplicatif.

On peut aussi passer les paramètres (`GL_DST_COLOR, GL_SRC_COLOR`) pour obtenir un blending multiplicatif + additif. Ce blending va multiplier la couleur précédente par la couleur courante et va additionner la couleur courante au résultat. Ce blending peut être utilisé par le light mapping pour obtenir un effet de saturation des couleurs.

Nous ne traiterons pas d'autres modes de blending ici car ils ne sont pas utiles pour le light mapping.

II-2 - Principe du light mapping.

Maintenant que nous savons comment fonctionne le blending (au moins la partie du blending qui nous intéresse), nous allons reparler du light mapping.

Le light mapping est un algorithme de rendu de la lumière multi-passes. La première passe consiste à afficher le niveau en affichant uniquement les textures de luminosité (light map). Une fois cet affichage effectué, nous avons donc notre scène rendue en niveaux de gris (ou en niveaux de couleur si on souhaite avoir des lumières colorées). Ensuite, il ne reste plus qu'à activer le blending multiplicatif et rendre la seconde passe en affichant le niveau texturé. Le résultat final sera le niveau texturé mais avec des zones d'ombres dans les textures correspondantes aux zones non éclairées.



Les 2 passes du light mapping et le rendu final.

À la base, le light mapping est un système créé pour gérer des lumières statiques. Le calcul des light maps peut être très coûteux si on utilise des algorithmes complexes (radiosité ou autre) pour calculer la luminosité des texels. Toutefois, on peut facilement utiliser une simplification pour prendre en compte des lumières dynamiques sphériques.

Pour cela, nous allons simuler une texture en 3 dimensions représentant une sphère de lumière à l'aide d'une texture en 2D particulière et du multitexturing. Cette texture, appelée light map dynamique, est en fait un cercle de luminosité dont les bords doivent impérativement être noir.



La light map dynamique

Utilisée en mode non répétitif (GL_CLAMP_TO_EDGE) et avec une utilisation judicieuse du multitexturing, on peut obtenir une approximation d'une texture 3D efficace. Le concept consiste à utiliser la première unité de texture comme un tube lumineux (on calcule les coordonnées de texture sur le plan XY par exemple), puis à n'utiliser que la ligne centrale de la deuxième unité de texture (donc une bande de luminosité allant du noir au noir en passant par le blanc) comme facteur d'atténuation sur le troisième axe (dans notre exemple : Z).

Le calcul de nos coordonnées de texture pour nos deux unités de textures donne donc :

texture1.texcoord : { $u = \text{vertexToLight}.x / \text{rayon} + 0,5$; $v = \text{vertexToLight}.y / \text{rayon} + 0,5$ }

texture2.texcoord : { $u = \text{vertexToLight}.z / \text{rayon} + 0,5$; $0,5$ }

Nous ajoutons 0,5 aux coordonnées de texture car on considère que le centre de la lumière est au centre de la texture et non pas à son origine. La dernière coordonnée de texture est positionnée à 0,5 car on souhaite utiliser la ligne centrale de la texture.

Ce calcul n'effectue pas réellement une sphère, mais un cube. Cependant, étant donné que notre texture représente un cercle, le résultat visuel s'approche d'une sphère. (Essayer de changer la light map par une texture blanche avec juste les bords noirs pour voir la différence).

Maintenant que nous savons comment calculer les coordonnées de textures, il faut afficher les lumières. Comme nous l'avons vu plus haut, le light mapping s'effectue en 2 passes, et, étant donné que le blending utilisé est multiplicatif, l'ordre de ces 2 passes n'est pas important. On peut rendre d'abord les light maps puis les textures ou l'inverse car l'équation d'éclairage est :

*couleur finale = couleur light map * couleur texture.*

Néanmoins pour la mise en place du light mapping dynamique, il faut changer la façon dont est rendu le niveau. Nous allons d'abord rendre le niveau avec les light maps, puis nous afficherons les lumières (avec du blending additif), pour finalement multiplier le résultat par les textures du niveau. Ceci donne donc l'équation suivante :

*couleur finale = (couleur light map + couleur lumières dynamiques) * couleur texture.*

Qui ne donne pas le même résultat que si nous avons d'abord rendu les textures, puis les light maps et finalement les lumières dynamiques dont l'équation est :

*couleur finale = (couleur texture * couleur light map) + couleur lumières dynamiques*

Nous avons donc besoin de modifier notre programme pour prendre en compte ces changements de calculs, mais aussi pour gérer le multitexturing.

III - Modifications du programme.

Dans cette section, nous allons voir les modifications apportées au tutoriel précédent pour pouvoir utiliser le light mapping dynamique. Nous verrons d'abord comment charger les fonctions de l'extension de multitexturing, puis nous détaillerons les changements dans les méthodes de gestion de la lumière, pour finalement voir la séquence de commandes OpenGL à utiliser pour obtenir le rendu voulu.

III-1 - Gérer les extensions.

Pour gérer le multitexturing dans notre application nous avons besoin de deux fonctions que nous déclarons dans le fichier Main.cpp comme variables globales :

```
PFGLMULTITEXCOORD2FARBPROC      glMultiTexCoord2fARB = NULL;
PFGLACTIVETEXTUREARBPROC       glActiveTextureARB = NULL;
```

La fonction `glMultiTexCoord2fARB` sert à envoyer les coordonnées de texture 2D pour une unité de texture donnée et la fonction `glActiveTextureARB` permet de définir quelle unité de texture est en cours d'utilisation.

Pour avoir accès aux fonctions des extension, il ne faut pas oublier d'inclure le fichier `glext.h`

Comme nous aurons aussi besoins de la fonction d'envoi des coordonnées de textures dans notre classe de Model, nous la déclarons en externe dans le fichier Model.cpp

```
extern PFGLMULTITEXCOORD2FARBPROC      glMultiTexCoord2fARB;
```

Maintenant que nous avons déclaré nos fonctions, il faut les charger. Charger une fonction consiste à récupérer son adresse auprès du pilote de la carte graphique. Pour cela, nous ajoutons une fonction dans Main.cpp qui va s'occuper de charger les extensions pour ce tutoriel et ceux à venir. Cette fonction donne donc :

La fonction de chargement des extensions

```
void initExtensions()
{
    if (glutExtensionSupported("GL_ARB_multitexture"))
    {
        glMultiTexCoord2fARB =
        (PFGLMULTITEXCOORD2FARBPROC)wglGetProcAddress("glMultiTexCoord2fARB");
        glActiveTextureARB =
        (PFGLACTIVETEXTUREARBPROC)wglGetProcAddress("glActiveTextureARB");
        if (!glActiveTextureARB || !glMultiTexCoord2fARB )
        {
            fprintf(stderr, "impossible d'initialiser l'extension
            GL_ARB_multitexture\n");
            exit(0);
        }
    }
    else
    {
        fprintf(stderr, "votre carte ne supporte pas l'extension GL_ARB_multitexture\n");
        exit(0);
    }
}
```

Ici, nous utilisons la fonction `wglGetProcAddress` pour récupérer l'adresse des fonctions de multitexturing. Cette fonction étant une fonction `wgl`, elle est propre à la plateforme Windows et n'est pas portable. Il existe une fonction `glxGetProcAddress` pour Linux, ou encore une fonction portable dans la SDL.

Maintenant que nous avons chargé les fonctions, nous pouvons les utiliser comme n'importe quelle fonction

OpenGL.

III-2 - Modifications de la classe de model.

Pas de changement dans l'interface de la classe Model, par contre, les méthodes `initLighting`, `addLight` et `render` ne vont plus effectuer la même chose.

La méthode `initLighting` sert toujours à paramétrer l'éclairage ambiant. Dans un système avec light maps statiques, cette méthode consisterait à afficher la scène avec les light maps statiques, mais comme nous n'en avons pas, ici, nous nous contenterons d'afficher la lumière ambiante. Le code donne donc :

La méthode d'affichage de la couleur ambiante

```
void Model::initLighting(const Color& ambient)
{
    glColor3f(ambient.r,ambient.g,ambient.b);
    glBegin(GL_TRIANGLES);
    for(int i = 0;i < nbFaces; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            Vecteur &v = vertex[faces[i].vertexIndex[j]];
            glVertex3f(v.x,v.y,v.z);
        }
    }
    glEnd();
}
```

La méthode `addLight` permet d'ajouter une lumière à la scène. Comme nous l'avons vu, elle doit dessiner la scène en calculant l'éclairage des polygones. On suppose que la méthode est toujours appelée au bon endroit, c'est-à-dire que le blending additif est activé et que la couleur ambiante a déjà été rendue. Il faut aussi que les deux premières unités de textures contiennent la light map dynamique. Ici, la méthode `computeLighting` de la classe `Light` ne calcule plus une couleur, mais des coordonnées de texture en 3D qui sont retournées sous la forme d'un vecteur 3D. Comme nous utilisons du blending additif, l'ordre dans lequel sont ajouté les lumières n'a pas d'importance. Le code de la méthode donne donc :

La méthode permettant d'ajouter une lumière à la scène

```
void Model::addLight(Light& light)
{
    glColor3f(light.getColor().r,light.getColor().g,light.getColor().b);
    glBegin(GL_TRIANGLES);
    // pour chaque face
    for (int i = 0; i < nbFaces; i++)
    {
        // pour chaque vertex de la face
        for (int j = 0; j < 3; j++){
            // on calcule la lumiere
            Vecteur lightTexCoord = light.computeLighting
                (vertex[faces[i].vertexIndex[j]]);
            glMultiTexCoord2fARB(GL_TEXTURE0_ARB,
                lightTexCoord.x,lightTexCoord.y);
            glMultiTexCoord2fARB(GL_TEXTURE1_ARB,lightTexCoord.z,0.5f);
            Vecteur &v = vertex[faces[i].vertexIndex[j]];
            glVertex3f(v.x,v.y,v.z);
        }
    }
    glEnd();
}
```

La méthode `render` sert à effectuer la dernière passe de l'algorithme qui consiste à multiplier le résultat de l'éclairage par la texture de la scène. Ici aussi, nous supposons qu'OpenGL est bien paramétré pour que le rendu fonctionne, c'est-à-dire que l'équation de blending est placée en mode multiplicatif. Le code de la méthode donne donc :

La méthode permettant de rendre la dernière passe de l'algorithme

```

void Model::render()
{
    glColor3f(1,1,1);
    glBegin(GL_TRIANGLES);
    for(int i = 0; i < nbFaces; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            TexCoord &tc = texCoords[faces[i].texCoordIndex[j]];
            Vecteur &v = vertex[faces[i].vertexIndex[j]];
            glVertex3f(v.x,v.y,v.z);
            glTexCoord2f(tc.u,tc.v);
        }
    }
    glEnd();
}

```

Ne pas oublier de positionner la couleur courante sur du blanc, sinon le résultat final sera déformé par la couleur.

III-3 - Modification du calcul de la luminosité.

Comme nous l'avons vu dans la méthode addLight, la méthode computeLighting de la classe Light est changée. Elle doit calculer les coordonnées de textures à appliquer à nos deux unités de textures. Le calcul correspond à celui décrit dans la partie d'explication du light mapping. Il est donc très simple et se résume à :

La méthode de calcul de la lumière

```

Vecteur Light::computeLighting(const Vecteur & position)
{
    Vecteur ret;
    ret.x = (position.x - this->position.x) / (radius) + 0.5f;
    ret.y = (position.y - this->position.y) / (radius) + 0.5f;
    ret.z = (position.z - this->position.z) / (radius) + 0.5f;
    return ret;
}

```

Ne pas oublier que la méthode a changé de type de retour depuis le tutoriel précédent, elle ne retourne plus une couleur mais des coordonnées de textures 3D dans un vecteur.

III-4 - Le rendu final

Contrairement au tutoriel précédent, la mise en place du rendu est plus compliquée. Le rendu s'effectue en trois étapes :

- rendre la scène avec les light maps statiques (ici, la couleur ambiante).
- ajouter toutes les lumières à la scène.
- multiplier le résultat par la texture de la scène.

Ce qui nous donne donc :

Rendu de la couleur ambiante

```

model->initLighting(ambient);

```

Ajout des lumières à la scène

```

// On configure les unités de texture pour pouvoir effectuer le multitexturing.
// En fait les deux unités de texture utilisent la même texture.
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);

```

Ajout des lumières à la scène

```
glBindTexture(GL_TEXTURE_2D, lightmap);

glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, lightmap);

// On doit activer le blending additif pour pouvoir utiliser ce mode d'eclairage
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
// Inutile d'ecrire dans le depth buffer, la passe d'ambiante l'a deja fait
glDepthMask(GL_FALSE);
// On ajoute nos lumieres
model->addLight(light1);
model->addLight(light2);
// On desactive la seconde unite de texture qui ne nous sert plus par la suite
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
```

Multiplier le résultat

```
// Et on bind la premiere sur la texture du model
glActiveTextureARB(GL_TEXTURE0_ARB);
glBindTexture(GL_TEXTURE_2D, textureId);

// On doit changer le mode de blending : on passe en mode multiplicatif
glBlendFunc(GL_DST_COLOR, GL_ZERO);

model->render();

glDisable(GL_TEXTURE_2D);

glDepthMask(GL_TRUE);
glDisable(GL_BLEND);
```

Ici, au lieu d'utiliser une fonction de blending multiplicatif simple, on peut utiliser du blending multiplicatif + additif (GL_DST_COLOR, GL_SRC_COLOR) qui va donner un résultat avec des couleurs nettement plus saturées.



Le rendu final en mode multiplicatif normal.



Le rendu final en mode multiplicatif + additif.

IV - Conclusions.

Contrairement à ce que certains pensent, mettre en place un système de light mapping dynamique demande très peu de code pour un résultat de bonne qualité. Par contre, contrairement à la méthode par vertex, le light mapping demande de nombreux rendus de la scène (un rendu pour les light maps statiques, un par lumière dynamique et un dernier pour multiplier la lumière par la texture de la scène.) mais ceci permet d'avoir un bon rendu de la lumière quelque soit la géométrie contrairement à l'éclairage par vertex.

La solution présentée ici a quand même des désavantages : elle ne prend pas en compte l'orientation des faces, c'est-à-dire qu'une face qui tourne le dos à la lumière sera éclairée de la même façon que si elle lui faisait face. Nous n'avons donc pas de gestion de l'ombrage.

Un autre problème est que cette solution n'est pas facilement extensible pour prendre en compte des lumières de type spot. Pour cela, il faudrait mettre en place un système de projection de textures nettement plus complexe.

Nous verrons dans le prochain tutoriel comment mettre en oeuvre un système de light mapping utilisant une texture 3D comme light map dynamique à la place de deux textures 2D.

Vous pouvez télécharger les sources de ce tutoriel [ici](#) ou [ici \[http\]](#)

Une version PDF de ce tutoriel est disponible [ici](#) ou [ici \[http\]](#)