

# Gestion dynamique de la lumière avec OpenGL - Partie 3 : light mapping dynamique avec texture 3D

par [Michel de VERDELHAN](#)

Date de publication : 06/09/2006

Dernière mise à jour :

Ce tutoriel fait suite au tutoriel sur la mise en place d'un système de light mapping basique. Dans celui-ci, nous verrons comment mettre en place un système de light mapping se basant sur l'utilisation de textures 3D. Contrairement aux tutoriaux précédents, ici, nous ne verrons pas de nouvelle méthode de gestion de la lumière, mais nous allons approfondir la technique précédente. Ce tutoriel est relativement court et simple si on maîtrise bien le concept du light mapping, mais c'est une étape importante pour faciliter la compréhension des tutoriaux suivants.

- I - Pourquoi utiliser une texture 3D ?
- II - Modifications apportées au programme
  - II-1 - Gestion des extensions
  - II-2 - Générer une texture 3D
  - II-3 - Modification du rendu des models
  - II-4 - Le rendu final
- III - Conclusions

## I - Pourquoi utiliser une texture 3D ?

Le light mapping que nous avons vu dans le tutoriel précédent pose quelques problèmes, aussi bien d'un point de vue qualitatif que d'un point de vue de l'optimisation du rendu. En effet, comme nous l'avons vu, le fait d'utiliser deux unités de textures pour simuler notre sphère pose problème du point de vue de la qualité de la sphère. Ce n'est pas une vraie sphère, et, si le rendu pour une petite lumière semble acceptable, il l'est nettement moins pour le rendu des lumières avec un rayon plus important. C'est donc une première raison d'utiliser une texture 3D comme light map : on obtient une sphère nettement plus sphérique, ce qui améliore la qualité d'affichage des grandes lumières.

Le deuxième problème posé est que notre système utilise deux unités de textures, ce qui est important, surtout quand on souhaite utiliser un modèle d'éclairage complexe très demandeur en unités de texture (comme nous le verrons dans le sixième tutoriel.). De plus, cette forte consommation en unités de textures limite l'utilisation de l'éclairage par pixel sur les cartes anciennes (les GeForce 3 et 4 n'ont que 4 unités de textures, les GeForce 2, seulement 2...). Un des avantages de l'utilisation d'une texture 3D est justement de n'utiliser qu'une seule unité de texture pour la light map.

Ceci nous permettra d'utiliser une optimisation du rendu réduisant le nombre de passes nécessaires. Bien que cette optimisation soit utilisable sans texture 3D, le fait de n'avoir qu'une unité de texture pour la light map permet cette optimisation sur les cartes ne possédant que 2 unités de textures, et surtout, nous nous rapprochons plus d'un modèle d'éclairage réaliste.

## II - Modifications apportées au programme

Les modifications apportées au programme dans ce tutoriel sont relativement légères. Dans un premier temps, nous allons voir le chargement des extensions nécessaires, puis nous verrons comment effectuer le calcul de la texture 3D représentant la sphère d'atténuation, puis, pour terminer, nous aborderons le rendu final de la scène.

### II-1 - Gestion des extensions

Ici, pas de changement majeur. Nous allons juste ajouter plusieurs fonctions nécessaires au fonctionnement de notre programme, mais le principe reste le même que pour le tutoriel précédent.

Etant donné que nous allons utiliser une texture 3D, nous avons besoin de pouvoir lui passer des coordonnées de textures 3D. Pour cela, nous devons donc charger les fonctions qui prennent en charge cela pour le multitexturing. Nos déclarations de fonctions deviennent donc :

#### Déclaration des fonctions nécessaires pour le multitexturing : main.cpp

```
// les fonctions utilisees pour gerer le multitexturing
PFNGLMULTITEXCOORD2FARBPROC      glMultiTexCoord2fARB = NULL;
PFNGLMULTITEXCOORD3FARBPROC      glMultiTexCoord3fARB = NULL;
PFNGLMULTITEXCOORD4FARBPROC      glMultiTexCoord4fARB = NULL;
PFNGLOBJECTIDARBPROC              glObjectIDARB = NULL;
PFNGLACTIVE_TEXTUREARBPROC        glActiveTextureARB = NULL;
PFNGLCLIENTACTIVE_TEXTUREARBPROC  glClientActiveTextureARB = NULL;
```

#### Déclaration des fonctions nécessaire pour le multitexturing : Model.cpp

```
extern PFNGLMULTITEXCOORD2FARBPROC      glMultiTexCoord2fARB;
extern PFNGLMULTITEXCOORD3FARBPROC      glMultiTexCoord3fARB;
extern PFNGLMULTITEXCOORD4FARBPROC      glMultiTexCoord4fARB;
```

Maintenant que nous pouvons passer nos coordonnées de textures à nos textures 3D, il nous faut gérer ces textures 3D, nous devons donc charger les fonctions suivantes :

#### Déclaration des fonctions nécessaire pour les textures 3D

```
// les fonctions utilisees pour gerer les textures 3D
PFNGLTEXIMAGE3DEXTPROC            glTexImage3DEXT = NULL;
PFNGLTEXSUBIMAGE3DEXTPROC         glTexSubImage3DEXT = NULL;
```

*Les textures 3D sont une extension EXT et non pas ARB, c'est-à-dire qu'elle n'est pas forcément implémentée sur toutes les cartes 3D.*

Maintenant que nous avons déclaré nos fonctions, il ne faut pas oublier de les charger. Notre fonction de chargement des extensions devient donc :

#### Fonction de chargement des extensions OpenGL

```
void initExtensions()
{
    if (glutExtensionSupported("GL_ARB_multitexture"))
    {
        glMultiTexCoord2fARB =
        (PFNGLMULTITEXCOORD2FARBPROC)wglGetProcAddress("glMultiTexCoord2fARB");
        glMultiTexCoord3fARB =
        (PFNGLMULTITEXCOORD3FARBPROC)wglGetProcAddress("glMultiTexCoord3fARB");
        glMultiTexCoord4fARB =
        (PFNGLMULTITEXCOORD4FARBPROC)wglGetProcAddress("glMultiTexCoord4fARB");
        glActiveTextureARB =
        (PFNGLACTIVE_TEXTUREARBPROC)wglGetProcAddress("glActiveTextureARB");
        glClientActiveTextureARB =
        (PFNGLCLIENTACTIVE_TEXTUREARBPROC)wglGetProcAddress("glClientActiveTextureARB");
        if (!glActiveTextureARB || !glMultiTexCoord2fARB || !glMultiTexCoord3fARB
            || !glMultiTexCoord4fARB || !glClientActiveTextureARB)
        {
            // Gestion des erreurs de chargement des extensions
        }
    }
}
```

## Fonction de chargement des extensions OpenGL

```

    {
        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);
    }
    if (glutExtensionSupported("GL_EXT_texture3D"))
    {
        glTexImage3D = (PFNGLTEXIMAGE3DPROC) wglGetProcAddress("glTexImage3D");
        glTexSubImage3D = (PFNGLTEXSUBIMAGE3DPROC)
        wglGetProcAddress("glTexSubImage3D");
        if (!glTexImage3D || !glTexSubImage3D)
        {
            fprintf(stderr, "impossible d'initialiser l'extension GL_EXT_texture3D\n");
            exit(0);
        }
    }
    else
    {
        fprintf(stderr, "votre carte ne supporte pas l'extension GL_EXT_texture3D\n");
        exit(0);
    }
}

```

Une fois que nous avons chargé les extensions, nous pouvons voir comment calculer la texture 3D qui va nous servir de light map.

## II-2 - Générer une texture 3D

Avant de voir comment calculer notre light map 3D, je vais commencer par faire une rapide explication sur les textures 3D.

Les textures 3D peuvent être vu comme une superposition de plusieurs textures de même taille. Généralement, elles sont générées de manière procédural (comme c'est le cas ici), mais elles peuvent aussi être créées en chargeant plusieurs textures de même taille dans les différentes couches de la texture, pour créer, par exemple, une texture animée simplement (et en profitant de l'interpolation hardware entre les couches de la texture). Il faut aussi bien différencier les textures 3D des cubes maps, qui, bien qu'elles soient elles aussi des représentations 3D d'images, ne représentent pas un volume comme la texture 3D.

Dans notre cas, nous souhaitons créer une texture dont le volume correspond à une sphère de luminosité, c'est-à-dire que plus on s'éloigne du centre de la texture, plus la luminosité diminue. Pour effectuer cette sphère, nous allons utiliser une fonction qui va nous la générer plutôt que de charger plusieurs textures représentant les couches de la texture.

Le concept du calcul de la texture 3D est très simple, il suffit, pour chaque texel de la texture, de calculer sa distance par rapport au "centre" de la texture. Si le texel est proche du centre, il sera blanc, alors que si la distance est supérieure ou égale au rayon de la sphère (la moitié de la taille de la texture), alors le texel sera noir.

Ce calcul s'effectue sur le même principe que le calcul de l'atténuation de l'éclairage par vertex.

La fonction de calcul de la light map est sans doute la partie la plus compliquée de ce tutoriel, mais si vous avez bien compris le principe du calcul de l'atténuation présenté dans le premier tutoriel, celui-ci ne va sans doute pas poser de problèmes. Le code de la fonction donne donc :

## Fonction de création d'une light map 3D

```

void generateSphereMap(int size)
{
    glEnable(GL_TEXTURE_3D_EXT);
    if(size<1)
    {
        fprintf(stderr,"la taille de la texture 3D doit etre superieur à 1\n");
        return;
    }

    float *datas = new float[size*size*size*3];
    //on remplit avec la couleur noir
    for(int i=0; i<size*size*size*3; i++)
    {
        datas[i] = 0.0f;
    }
    float length;
    //la demi largeur de la texture 3d en texels (represente le rayon de la sphere)
    float size2= (float)size / 2.0f;

    // pour chaque texel de la texture
    for(float i=0; i<size; i++)
    {
        for(float j=0; j<size; j++)
        {
            for(float k=0; k<size; k++)
            {
                //la demi-largeur moins la distance du centre au texel courant de la
                //size2-1 pour eviter l'elairage sur le bord de la texture.
                length = (size2-1) - sqrt( (i-size2)*(i-size2) +
                    (j-size2)*(j-size2) + (k-size2)*(k-size2) );
                //normalisation dans la plage[0,1] (intensite de la lumiere)
                length /= size2;

                //on stocke la couleur avec son intensite dans le texel
                datas[(int)i*size*size*3 + (int)j*size*3 + (int)k*3 ] = length;
                datas[(int)i*size*size*3 + (int)j*size*3 + (int)k*3 +1] = length;
                datas[(int)i*size*size*3 + (int)j*size*3 + (int)k*3 +2] = length;
            }
        }
    }

    // on genere une texture openGL
    glGenTextures(1,&lightmap);
    glBindTexture(GL_TEXTURE_3D_EXT,lightmap);
    // on parametre la texture (ici on utilise que du filtrage linéaire, en cas
    // d'utilisation de mipmapping, il faut calculer les different niveaux
    // de mipmap.)
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    // le mode de repetition : CLAMP_TO_EDGE pour ne pas voir la texture se repeter.
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
    // la texture doit moduler la couleur precedente.
    glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    // on envoi la texture à openGL
    glTexImage3D(GL_TEXTURE_3D_EXT, 0, GL_RGB, size, size, size, 0, GL_RGB, GL_FLOAT, datas);

    glDisable(GL_TEXTURE_3D_EXT);
    delete [] datas;
}

```

Ici, nous n'utilisons qu'un filtrage linéaire de la texture. Pour utiliser du filtrage bilinéaire ou trilineaire (`GL_LINEAR_MIPMAP_NEAREST` et `GL_LINEAR_MIPMAP_LINEAR`), il faut générer tous les niveaux de mipmap de la texture 3D.

Maintenant que nous avons une light map, nous pouvons voir les modifications à apporter au rendu de modèles pour qu'ils prennent en compte l'utilisation d'une texture 3D.

## II-3 - Modification du rendu des modèles

C'est dans cette partie que nous allons voir l'optimisation présentée au début du tutoriel permettant de réduire le nombre de passes pour rendre la scène.

Dans le tutoriel précédent, le rendu s'effectuait en 3 parties : le rendu des light maps statiques, le rendu des light maps dynamiques en niveau de couleur, puis la multiplication du résultat. La scène était rendue dans cet ordre pour que le programme puisse tourner sur toutes les cartes supportant le multitexturing (donc avec deux unités de textures).



*Le mode de rendu présenté dans le tutoriel précédent.*

L'optimisation présentée ici peut être appliquée au tutoriel précédent si la carte supporte plus de deux unités de textures.

Cette optimisation consiste à supprimer la dernière étape de rendu de la scène. En effet, si la scène est rendu à chaque étape avec à la fois la light map et la texture du modèle, on a donc à chaque étape à la fois la luminosité et la couleur de la texture qui sont multipliées par le multitexturing, puis le résultat est ajouté à la scène.

Le dessin suivant explique mieux de concept.



*Rendu de la scène en deux étapes*

La première étape de ce dessin représente le résultat de l'éclairage dynamique sur une surface avec multiplication de la texture par la light map dynamique. La deuxième étape représente l'affichage de la light map statique avec la texture de la surface. Le tout est additionné pour donner le résultat final. On a donc ici uniquement deux étapes de rendu : une étape de rendu des light maps, et une étape de rendu des lumières dynamiques. Un autre avantage de cette méthode est qu'elle nous redonne la possibilité d'organiser ses deux étapes dans l'ordre que l'on souhaite.

Etant donné que nous utilisons désormais une seule unité de texture pour notre light map, un tel système est simple à mettre en oeuvre, il nous suffit d'attribuer la light map à une unité de texture, et la texture de la surface à une autre unité de texture.

Nous allons donc modifier le comportement de notre programme en conséquence.

Comme pour le tutoriel précédent, ici, nous n'avons pas de light maps statiques, nous allons donc juste rendre la scène avec une couleur ambiante pour simuler l'utilisation de light maps. Il faut donc que notre méthode `initLighting` affiche l'ensemble du niveau avec ses textures multipliées par la couleur ambiante. Le code de la méthode devient donc :

## Code de la méthode initLighting

```
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++)
        {
            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();
}
```

De la même façon, le code de la méthode addLight doit simplement afficher le modèle en lui attribuant les coordonnées de texture 3D pour la light map dynamique. Ici, nous supposons qu'OpenGL est dans un état cohérent, c'est-à-dire que l'unité de texture 0 contient bien la texture du modèle, et que l'unité de texture 1 contient la light map dynamique en 3D. Le code de la méthode addLight devient donc :

## Code de la méthode addLight

```
void Model::addLight(Light& light)
{
    glColor3f(light.getColor().r, light.getColor().g, light.getColor().b);
    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]];
            Vecteur lightTexCoord = light.computeLighting(v);
            glMultiTexCoord2fARB(GL_TEXTURE0_ARB, tc.u, tc.v);
            glMultiTexCoord3fARB(GL_TEXTURE1_ARB, lightTexCoord.x, lightTexCoord.y, lightTexCoord.z);
            glVertex3f(v.x, v.y, v.z);
        }
    }
    glEnd();
}
```

La méthode computeLighting n'a pas changé depuis le dernier tutoriel.

Maintenant que nous avons tous les outils pour effectuer le rendu de notre scène, nous allons voir comment effectuer ce rendu.

## II-4 - Le rendu final

Comme nous l'avons vu, le rendu final de la scène ne s'effectue plus en trois étapes mais en deux. La première consiste à rendre les light maps statiques (simplement la lumière ambiante dans notre cas), et s'effectue tout simplement comme ça :

## Rendu de la lumière ambiante

```
// on configure la première unité de texture : elle contient la texture de la scène
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textureId);

// on effectue le rendu de la lumière ambiante
model->initLighting(ambient);
```

La deuxième étape consiste à ajouter chaque lumière à la scène. Le code donne donc :

### Rendu des lumières dynamiques

```
// on configure la deuxieme unite de texture : elle contient la lightmap
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_3D_EXT);
glBindTexture(GL_TEXTURE_3D_EXT,lightmap );

// on doit activer le blending additif pour pouvoir utiliser ce mode d'eclairage
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);
glDepthMask(GL_FALSE);

// on ajoute nos lumieres
model->addLight(light1);
model->addLight(light2);

// on desactive la seconde unite de texture
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_3D_EXT);
// on desactive la premiere unite de texture
glActiveTextureARB(GL_TEXTURE0_ARB);
glDisable(GL_TEXTURE_2D);

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

Voilà c'est tout. Nous avons maintenant une gestion dynamique des lumières par light mapping avec texture 3D.

Le rendu final de la scène donne ça :



*Le rendu final de la scène.*



### III - Conclusions

Ca y est, nous en avons fini avec le light mapping. Nous avons maintenant un système fonctionnel, mais il faut néanmoins faire attention à bien l'utiliser. En effet, dans notre cas, la scène affichée est simple et, surtout, elle n'est composée que d'une seule texture. Dans une vraie scène, on utilise généralement de nombreuses textures, ce qui peut poser problème avec notre réduction du nombre de passes. En effet, si la gestion des textures est faite de façon anarchique, il risque d'y avoir multiplication des bind de textures (étant donné qu'on a toujours besoin d'au moins deux unités de textures par rendu), ce qui risque de faire perdre le bénéfice de l'optimisation proposée. Donc si vous souhaitez implémenter un tel système, prenez bien garde à limiter autant que possible les changements de textures.

Contrairement aux tutoriaux précédents et à ceux qui vont suivre, celui-ci est plutôt simple et ne présente pas de nouveau concepts de gestion de la lumière, mais il est néanmoins indispensable de bien l'appréhender pour pouvoir comprendre la suite.

Autant les tutoriaux entre l'éclairage par vertex et l'éclairage par light mapping sont relativement déconnectés, autant le lien entre ce tutoriel et les suivants sur l'éclairage par pixel est important. En effet, l'éclairage par pixel utilise les principes du light mapping pour gérer l'atténuation de la lumière.

Pour récapituler, nous avons vu à l'heure actuelle comment mettre en oeuvre un système de gestion de l'éclairage dynamique par vertex et par light mapping, et il nous reste à voir comment mettre en oeuvre l'éclairage par pixel. Ceci fera l'objet des trois prochains tutoriaux.

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

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