

Gestion dynamique de la lumière avec OpenGL - Partie 1 : l'éclairage par vertex

par [Michel de VERDELHAN](#)

Date de publication : 06/09/2006

Dernière mise à jour :

Ce tutoriel a pour but la mise en place d'un système de gestion des lumières dynamiques par vertex lighting avec OpenGL. nous présenterons aussi une vue d'ensemble des méthodes de gestion des lumières dynamiques dans les moteurs 3D, qui seront traitées pour la plupart dans les prochains tutoriaux.

- I - But de ce tutoriel.
- II - Pourquoi mettre en place son propre système de gestion de la lumière.
- III - Les différents modèles d'éclairage.
 - III-1 - L'éclairage par vertex
 - III-2 - Le light mapping
 - III-3 - L'éclairage par pixel
 - III-4 - Autres modèles
- IV - Un premier système de gestion de la lumière : l'éclairage par vertex.
 - IV-1 - Principe de l'éclairage par vertex
 - IV-2 - Structure de données.
 - IV-3 - Gestion de l'atténuation.
 - IV-4 - Gestion de l'ombrage.
 - IV-5 - Le rendu final
- V - Conclusions.

I - But de ce tutoriel.

Cette série de tutoriaux a pour but d'étudier la mise en oeuvre des systèmes d'éclairage dans un moteur 3D.

Il existe de très nombreuses façons de gérer son éclairage dans un moteur 3D. Dans cette série de tutoriaux, nous aborderons les techniques d'éclairages les plus connus. Néanmoins, nous ne traiterons que les techniques permettant de gérer un éclairage dynamique de nos scènes, et nous ne parlerons que des lumières ponctuelles omnidirectionnelles à rayon fini. Nous n'aborderons donc pas les sources de lumières infini, ou les spots de lumière.

- 1 L'éclairage par vertex.
- 2 Le light mapping dynamique simple.
- 3 Le light mapping dynamique avec texture 3D.
- 4 L'éclairage par pixel simple.
- 5 L'éclairage par pixel standard.
- 6 La création d'un système d'éclairage complexe basé sur les shaders.

Pour la mise en oeuvre, nous utiliserons les bibliothèques OpenGL et GLUT pour gérer l'affichage. Néanmoins, l'ensemble des principes étudié ici est facilement portable sous Direct3D.

II - Pourquoi mettre en place son propre système de gestion de la lumière.

Une question souvent posée sur les forums est de savoir comment gérer les lumières dans un moteur 3D. En effet, même si OpenGL propose une gestion des lumières standard en interne, celle-ci n'est pas réellement utilisable dans un moteur complexe. Bien qu'accéléérées par le matériel, les lumières OpenGL posent plusieurs problèmes lorsqu'on souhaite les utiliser dans un moteur complexe :

- L'éclairage est calculé par vertex.

Cette façon de gérer l'éclairage oblige à avoir un nombre de polygones élevé pour obtenir une qualité d'éclairage intéressante. En effet, l'éclairage étant interpolé entre les sommets d'un polygone, si les polygones sont trop étendus, l'éclairage ne sera pas bon au centre du polygone. La seule solution pour résoudre ce problème est de découper les grands polygones en plusieurs petits polygones (on parle d'augmentation de la tessellation). Cette solution résout relativement bien le problème de l'interpolation de la lumière, mais elle a aussi comme conséquence d'augmenter fortement la géométrie ce qui a un impact important sur les performances, notamment au niveau de :

- l'envoi de la géométrie à la carte 3D

- l'utilisation de certains algorithmes (par exemple : les collisions) qui seront plus coûteux, ou encore pour le calcul des ombres volumétriques qui dépendent fortement de la géométrie.

- L'éclairage étant calculé par la carte graphique, le nombre de lumières utilisables simultanément est limité par le matériel. C'est principalement cette limitation qui impose la mise en place de systèmes de gestion de la lumière dans les moteurs 3D. En effet, on imagine mal un jeu moderne qui ne permet d'avoir que 8 lumières dans un niveau. Il existe bien entendu des méthodes qui permettent d'activer/désactiver des lumières quand elles ne sont plus visibles. Ceci permet de repousser la limite du nombre de lumières, mais cette limite existe toujours. Imaginez un très grand hangar avec seulement 8 lumières là où on en aurait mis plusieurs dizaines dans la réalité, le résultat risque de ne pas être très crédible.

Tous ces problèmes ont contraint les créateurs de moteurs 3D à mettre en place des systèmes de gestion de la lumière plus ou moins complexes.

III - Les différents modèles d'éclairage.

Il existe de nombreuses manières d'appréhender la lumière. La plupart des systèmes de gestion de la lumière sont basés sur les modèles de Phong et de Blinn. Néanmoins, de nombreux moteurs 3D n'implémentent pas ces modèles dans leurs totalités.

Je ne détail pas ici les modèles de Phong et Blinn, mais le net regorge de sites qui les traitent très bien.

Le principe de base d'un système de gestion de la lumière est de pouvoir noircir plus ou moins les éléments d'une scène en fonction de la position et de l'orientation du pixel par rapport à la lumière.

Généralement, les modèles d'éclairage considèrent que la lumière est additive. Ce qui signifie que si une surface est éclairée par deux lumières, le résultat final sera l'addition des deux éclairages. Ainsi une surface éclairée par une lumière rouge $\{1,0,0\}$ et par une lumière bleu $\{0,0,1\}$ aura la même couleur que si elle avait été éclairée par une lumière violette $\{1,0,1\}$. Ceci est particulièrement important pour déterminer la façon dont seront affichées les lumières par la suite.

Maintenant que nous avons vue cette introduction basique des modèles d'éclairages, nous allons voir des systèmes classiques pour les mettre en oeuvre.

Les types de systèmes de gestion de l'éclairage dynamiques les plus connus sont :

III-1 - L'éclairage par vertex

C'est, historiquement, le premier modèle d'éclairage en temps réel utilisé. En effet, l'éclairage en temps réel étant coûteux, lorsque les ordinateurs n'avaient pas de carte accélératrice, il fallait utiliser un système très rapide, même si il présente les désavantages présentés précédemment.

C'est ce modèle dont nous allons étudier la mise en oeuvre dans ce premier tutoriel.

III-2 - Le light mapping

Comme son nom l'indique (si vous n'êtes pas anglophobe), le light mapping consiste à stocker l'éclairage dans une texture. Cet algorithme se décompose en 2 étapes :

- 1 Le rendu de la scène en utilisant la texture d'éclairage. Les parties dans l'ombre sont affichées en noir alors que le reste est affiché dans la couleur de l'éclairage local.
- 2 Le rendu de la scène avec la texture normale, en multipliant la couleur précédente. Ainsi, si la couleur précédente était du noir, le résultat final sera du noir, donc une zone ombrée. Si la couleur précédente était du blanc, la couleur final obtenu est celle de la texture, donc une zone éclairée.

Cette technique a comme gros avantage de permettre de pré calculer la texture d'éclairage pour les lumières statiques, et, étant basé sur l'utilisation de textures, d'utiliser principalement les ressources de la carte graphique.

Le light mapping est à la base un système de gestion des lumières statiques, mais il est facile de lui ajouter une gestion de l'éclairage dynamique.

Cette méthode sera vue dans les tutoriaux 2 et 3.

III-3 - L'éclairage par pixel

Basé sur des extensions comme le DOT3 bump mapping ou les shaders, l'éclairage par pixel est possible de nos jours car les cartes graphiques ayant évolué, le développeur peut maintenant intervenir sur la façon dont sont calculé les pixels avant qu'ils ne soient ajouté à la scène. Cette méthode de gestion de l'éclairage permet un très bon rendu graphique, tout en étant très flexible. Le choix peut donc être fait entre la qualité graphique ou la performance, en implémentant seulement une partie d'un modèle d'éclairage par exemple.

Cette méthode permettant de nombreuses implémentations, nous l'étudierons durant 3 tutoriaux.

III-4 - Autres modèles

Il existe d'autres méthodes (qui sont généralement calculées par pixel) comme les spherical harmonics par exemple, mais ces techniques étant plus anecdotiques dans l'industrie 3D, nous ne les étudierons pas ici.

IV - Un premier système de gestion de la lumière : l'éclairage par vertex.

Nous allons maintenant voir comment mettre en oeuvre un système d'éclairage par vertex, mais avant de voir sa mise en oeuvre précise, je vais essayer d'en expliquer rapidement le principe.

IV-1 - Principe de l'éclairage par vertex

L'éclairage par vertex consiste simplement à effectuer l'ensemble des calculs d'éclairage au niveau des vertex. Ce système est très simple et peut se résumer à l'algorithme en pseudo code suivant :

```
initialiser la luminosite de tout les vertex a la lumiere ambiante
pour chaque lumiere
pour chaque face
pour chaque vertex de la face
vertex.couleur = lumiere.couleur * (NdotL) * atténuation
fin pour
fin pour
fin pour
rendre la scène
```

L'algorithme peut être simplifié en ne prenant pas en compte les vertex des faces, mais directement la géométrie, mais cela a pour conséquence de ne pas pouvoir avoir de vertex à normales multiples.

Un vertex à normales multiples est un point de l'espace qui est partagé entre plusieurs faces ayant des orientations très différentes. Par exemple un cube a 8 vertex, mais ses faces ont des orientations suffisamment différentes pour qu'on ne souhaite pas lisser les normales de ces vertex (on parle de normal smoothing), le vertex aura alors plusieurs normales (dans le cas du cube, une par face qui utilise le vertex). Généralement la gestion des normales lissées est laissée à la discrétion de l'artiste qui crée la scène via un outil de modélisation. Ici, nous utiliserons donc les normales contenues dans le fichier.

Pour commencer l'étude de la mise en oeuvre, nous allons voir la structure de données utilisée par nos calculs, ensuite nous aborderons le problème du calcul de l'atténuation de la luminosité, et pour finir, nous parlerons de l'ombrage.

Nous ne traiterons pas des composantes spéculaire et émissive de la lumière, mais leur implémentation est relativement facile à mettre en place.

IV-2 - Structure de données.

Ici, nous allons parler des structures de donnée utilisé pour le calcul de la luminosité de la scène. J'utilise dans ce tutoriel des classes basiques pour gérer mes vecteurs, couleurs et coordonnées de textures. Ces classes ne seront pas détaillées ici mais elles sont disponibles avec le code.

Pour le tutoriel nous utilisons une scène stockée dans un fichier .obj. Le chargement de ce fichier n'entre pas en compte dans ce tutoriel, mais il nous permet de récupérer les informations de géométrie sous la forme de 4 tableaux qui contiennent :

- 1 Les positions des vertex.
- 2 Les normales.
- 3 Les coordonnées de textures.
- 4 Les faces.

Les faces sont des structures qui contiennent les indices des vertex/normales/coordonnées de texture de chacun des points de la face. Ici nous ne traitons que des faces triangulaires.

Nous avons donc besoin d'une structure pour stocker ces faces

La structure gerant une face.

```
struct Face
{
    unsigned int vertexIndex[3];
    unsigned int texCoordIndex[3];
    unsigned int normalIndex[3];
    Color color[3];
};
```

Ici, la face contient aussi une couleur pour chacun de ses vertex, c'est la couleur qui sera utilisée pour déterminer l'éclairage final du vertex. Cette information est propre à ce tutoriel et disparaîtra dans les tutoriaux suivants.

Nous avons besoin d'une classe qui va gérer notre géométrie. Cette classe doit pouvoir stocker les informations sur les vertex, normales, coordonnées de textures et sur les faces. Elle doit aussi permettre de charger un fichier de géométrie, gérer l'éclairage et le rendu de la scène.

La classe gerant un modèle (une scène)

```
class Model
{
private:
    Vecteur * vertex;
    Vecteur * normals;
    TexCoord * texCoords;
    Face * faces;
    int nbVertex;
    int nbNormals;
    int nbTexCoord;
    int nbFaces;
public:
    Model(void);
    virtual ~Model(void);
    bool load(const std::string & filePath);
    void initLighting(const Color& ambient);
    void addLight(Light& light);
    void render();
};
```

La méthode load() sert à charger la géométrie depuis un fichier, elle n'est pas particulièrement intéressante ici, la seule information qui nous intéresse est que le chargement prend en compte les normales partagées, nous avons donc un modèle dont certains vertex ont plusieurs normales et donc un rendu cohérent des cubes et autres formes avec des angles important entre les faces.

La méthode initLighting prend en paramètre la couleur ambiante et l'assigne à tout les vertex de la scène. Cette méthode donne donc :

La méthode d'initialisation de la lumière

```
void Model::initLighting(const Color& ambient)
{
    for (int i = 0; i < nbFaces; i++)
    {
        faces[i].color[0] = ambient;
        faces[i].color[1] = ambient;
        faces[i].color[2] = ambient;
    }
}
```

La méthode addLight prend en paramètre une lumière, effectue les calculs d'éclairage et les applique aux vertex.

Le code donne donc :

La méthode permettant d'ajouter une lumière au modèle

```
void Model::addLight(Light& light)
{
    // pour chaque face
    for (int i = 0; i < nbFaces; i++)
    {
        // pour chaque vertex de la face
        for (int j = 0; j < 3; j++)
        {
            // on calcul la lumiere
            Color c =
light.computeLighting(vertex[faces[i].vertexIndex[j]], normals[faces[i].normalIndex[j]]);
            // et on l'addition avec la lumiere precedente
            faces[i].color[j].r += c.r;
            faces[i].color[j].g += c.g;
            faces[i].color[j].b += c.b;
        }
    }
}
```

Avec computeLighting, la fonction de calcul de la luminosité du vertex. Cette méthode sera vue plus loin.

Ici la luminosité calculée est additionnée à celle calculée précédemment, car, comme nous l'avons vue, la lumière est additive.

Il ne nous reste plus que la méthode permettant le rendu de la scène. Elle est relativement simple, et ici, je n'utilise aucune optimisation à base de tableaux de vertex ou autre, j'effectue un simple rendu en mode immédiat.

Méthode de rendu du modèle

```
void Model::render()
{
    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 recupere la couleur, les coordonnees de texture et la position
            Color &c = faces[i].color[j];
            TexCoord &tc = texCoords[faces[i].texCoordIndex[j]];
            Vecteur &v = vertex[faces[i].vertexIndex[j]];
            // et on les envoient à OpenGL
            glColor3f(c.r,c.g,c.b);
            glTexCoord2f(tc.u,tc.v);
            glVertex3f(v.x,v.y,v.z);
        }
    }
    glEnd();
}
```

Maintenant que nous avons notre classe de gestion de la géométrie, nous avons besoins d'une classe qui gère une lumière.

Pour nous simplifier la gestion des calculs, c'est cette classe qui va calculer l'éclairage des vertex en fonction de leur position et de leur normale.

La classe de gestion des lumières

```
class Light
{
private:
    float radius;
    Color color;
    Vecteur position;
```

La classe de gestion des lumières

```

public:
    Light(void);
    virtual ~Light(void);
    void setRadius(float r);
    float getRadius();
    void setColor(float r, float g, float b);
    Color getColor();
    void setPosition(float x, float y, float z);
    Vecteur getPosition();
    Color computeLighting(const Vecteur & position, const Vecteur & normal);
};

```

Notre lumière contient simplement une position, un rayon et une couleur. Mis à part les accesseur et modificateur, cette classe ne contient qu'une méthode intéressante : `computeLighting` qui prend en paramètre la position d'un vertex et sa normal et retourne sa couleur d'éclairage. C'est cette méthode qui permet de calculer l'éclairage du vertex comme nous allons le voir.

IV-3 - Gestion de l'atténuation.

La première chose à prendre en compte dans notre calcul de la luminosité est l'atténuation de la lumière. En effet plus notre vertex est loin de la lumière, moins il est éclairé, et si il dépasse le rayon d'action de la lumière, il n'est plus du tout éclairé.

Il existe plusieurs équations pour calculer l'atténuation (linéaire, quadratique et autre). Ici, nous utiliserons une équation linéaire. C'est à dire qu'un vertex situé à mi-distance du rayon de la lumière recevra la moitié de sa luminosité.

L'équation donne donc

$$\text{atténuation} = \max(0, 1 - (\text{distance} / \text{rayon}))$$

ou *atténuation* est le facteur d'atténuation de la lumière en fonction de la distance, *distance* est la distance du vertex par rapport à la position de la lumière et *rayon* est le rayon maximum de la lumière.

Le code du calcul de l'atténuation donne donc :

Code du calcul de l'atténuation

```

// on calcul le vecteur allant du vertex à la lumière.
Vecteur vertexToLight = this->position - position;
float attenuation = std::max<float>(0.0f, 1.0f - (vertexToLight.getLength() / radius));

```

Nous calculons le vecteur allant du vertex à la lumière. Ici, le sens du vecteur n'a pas d'importance étant donné que ce qui nous intéresse est sa longueur, mais il est néanmoins important de le calculer dans le bon sens car il est réutilisé par la suite dans la gestion de l'ombrage.

IV-4 - Gestion de l'ombrage.

Ce qui est appelé ombrage en infographie est à différencier des ombres portées. L'ombre portée correspond à l'ombre générée par un objet sur un autre objet. L'ombrage lui est juste le fait qu'une face est moins éclairé si elle ne fait pas face à la lumière et est complètement ombrée si elle tourne le dos à la lumière.

Nous voulons donc obtenir ce comportement pour nos vertex. Pour cela, nous allons justement utiliser le fameux calcul `NdotL`. Mais d'abord un petit rappel de maths sur le produit scalaire s'impose.

Le produit scalaire entre deux vecteurs normalisé (de longueur 1) a une propriété intéressante :

Si les 2 vecteurs sont identiques, le produit scalaire vaut 1. Si les 2 vecteurs ont la même direction, le produit scalaire est compris dans]0..1]. Si les 2 vecteurs sont perpendiculaires, le produit scalaire vaut 0, si ils sont de sens opposé, le produit scalaire est compris dans [-1..0[, et si ils sont complètement opposé, le produit scalaire vaut -1.

Appliqué à notre problème, si le produit scalaire entre la normale et le vecteur vertexToLight est supérieur à 0, c'est que la face est éclairé, sinon, elle est dans l'ombre. Et, encore plus intéressant, si la normal est exactement identique au vecteur vertexToLight (c'est à dire si le polygone fait exactement face à la lumière), le produit scalaire vaut 1. Nous avons donc, à moindre coût, le calcul exacte de l'ombrage de nos faces avec un simple produit scalaire (et une normalisation de vecteur car il faut que le vecteur vertexToLight soit normalisé pour que ce calcul fonctionne). Nous avons donc retrouvé notre NdotL, mais nous l'appliquons par vertex et non par faces.

Notre fonction de calcul de la luminosité devient donc :

Calcul de l'atténuation en prenant en compte l'ombrage

```
// on calcul le vecteur allant du vertex a la lumiere.
Vecteur vertexToLight = this->position - position;
float attenuation = 1.0f - (vertexToLight.getLength() / radius);
// on normalise le vecteur
vertexToLight.normalise();
// on calcul le produit scalaire NdotL avec N = normal et L = vertexToLight
attenuation *= vertexToLight*normal;
attenuation = std::max<float>(0.0f,attenuation);
```

Maintenant que nous avons calculé la luminosité de notre vertex, nous devons calculer sa couleur. Ceci se fait simplement en multipliant la couleur de la lumière par la luminosité du vertex. Nous obtenons donc le code final de la méthode computeLighting :

La methode de calcul de la couleur d'un vertex

```
Color Light::computeLighting(const Vecteur & position, const Vecteur & normal)
{
    // on calcul le vecteur allant du vertex à la lumiere.
    Vecteur vertexToLight = this->position - position;
    float attenuation = 1.0f - (vertexToLight.getLength() / radius);
    // on normalise le vecteur
    vertexToLight.normalise();
    // on calcul le produit scalaire
    attenuation *= vertexToLight*normal;
    attenuation = std::max<float>(0.0f,attenuation);
    Color ret = this->color;
    ret.r *= attenuation;
    ret.g *= attenuation;
    ret.b *= attenuation;
    return ret;
}
```

IV-5 - Le rendu final

Maintenant que nous avons tout ce qu'il faut pour calculer notre éclairage, il nous faut l'afficher. Ceci se fait très simplement en respectant la séquence suivante :

La séquence de code à suivre pour rendre la scène

```
scene->initLighting(ambient);
// on ajoute nos lumieres
scene->addLight(light1);
scene->addLight(light2);
// on active la texture
glEnable(GL_TEXTURE_2D);
// on affiche la scene
```

La séquence de code à suivre pour rendre la scène

```
scene->render();
```

Ce qui donne le resultat suivant :

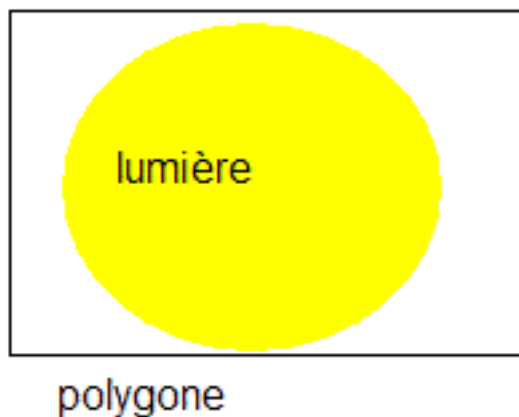


Scène éclairée par 3 lumières : 1 rouge, 1 bleu et 1 blanche

V - Conclusions.

La méthode d'éclairage par vertex est simple à mettre en oeuvre et présente comme avantage important par rapport aux techniques que nous étudierons plus tard de dessiner la scène qu'une seule fois. Nous économisons donc du temps sur l'envoi de la géométrie, mais c'est du coup le processeur qui prend en charge la majorité des calculs.

Un autre défaut vu précédemment est le problème des artefacts d'éclairage sur les polygones trop étendu. Cette artefact se produit principalement au niveau du calcul de l'atténuation. En effet, lorsqu'on place une lumière sur un polygone trop étendu, on peut tomber sur la situation où le rayon de la lumière n'est pas suffisant pour éclairer les vertex du polygone, il n'est donc pas éclairé alors que la lumière le touche.



Cas où l'atténuation de l'éclairage par vertex pose problème

Pour résoudre ces problèmes, nous allons devoir changer de méthode d'éclairage et passer au light mapping que nous verrons dans le prochain tutoriel.

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

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