Apprendre OpenGL moderne

Deuxième partie : éclairage

Par Joey de Vries - Jean-Michel Fray (traducteur)

Date de publication : 2 juin 2018

TOUT PUBLIC

Developpez.com a la joie d'accueillir une traduction en français du célèbre cours de grande qualité **Learn OpenGL**. Au cours de celui-ci, vous apprendrez à programmer des applications graphiques 3D grâce à la bibliothèque OpenGL.

Ces tutoriels sont accessibles aux débutants, mais les connaisseurs ne s'ennuieront pas non plus grâce aux chapitres avancés.

Cette page vous amène à la deuxième partie du tutoriel, c'est-à-dire : l'éclairage (modèle Phong), les matériaux, les textures de lumière et la gestion de plusieurs lumières.

Vous pouvez retrouver les autres parties ci-dessous :

- **introduction** ;
- **éclairage** ;
- **chargement de modèle** ;
- **OpenGL avancé** ;
- éclairage avancé ;
- PBR ;
- **mise en pratique**.

Commentez

I - Les couleurs....3
I-A - Une scène éclairée....4
I-B - Remerciements....6
II - Éclairage simple....7
II-A - Éclairage ambiant....7
II-B - Éclairage diffus....8
II-B-1 - Vecteur normal (ou normale)....9
II-B-2 - Calcul de la composante diffuse....10
II-B-3 - Une dernière chose....11
II-C - Éclairage spéculaire....13
II-D - Exercices....16
II-E - Remerciements....16
III - Matériaux....16
III-A - Mise en place des matériaux....17
III-B - Propriétés des sources lumineuses....18
III-C - Différentes couleurs de sources lumineuses....19
III-D - Exercices....20
III-E - Remerciements....20
IV - Textures de lumière (lighting maps)....20
IV-A - Textures de lumière diffuse....20
IV-B - Textures de lumière spéculaire....23
IV-C - Échantillonnage de texture spéculaire....24
IV-D - Exercices....26
IV-E - Remerciements....26
V - Sources de lumières....26
V-A - Sources directionnelles....26
V-B - Sources ponctuelles....29
V-B-1 - Atténuation....29
V-B-2 - Choisir les bonnes valeurs....30
V-B-3 - Implémenter l'atténuation....31
V-C - Spot lumineux....32
V-D - Lampe frontale....33
V-E - Atténuation des bords....35
V-F - Exercices....37
V-G - Remerciements....37
VI - Plus de lumières....37
VI-A - Éclairage directionnel....38
VI-B - Sources ponctuelles....39
VI-C - Tout rassembler....40
VI-D - Exercices....42
VI-E - Remerciements....42
VII - Résumé....42
VII-A - Glossaire....43
VII-B - Remerciements....43

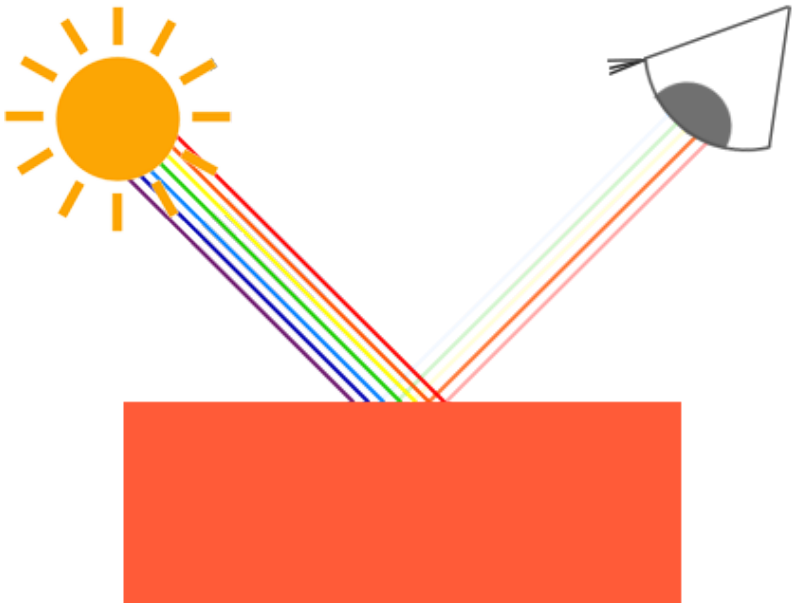
I - Les couleurs

Dans les premiers chapitres, nous avons rapidement montré comment travailler avec les couleurs dans OpenGL, mais nous n'avons fait qu'effleurer le sujet. Dans ce chapitre nous allons approfondir nos connaissances et concevoir une scène de base pour les tutoriels sur l'éclairage.

Dans le monde réel, les couleurs sont innombrables : chaque objet ayant sa propre couleur. Dans le monde numérique, il nous faut représenter cette infinité de couleurs avec un nombre limité de valeurs numériques, on ne pourra donc pas représenter toutes les nuances possibles. On peut cependant représenter un très grand nombre de teintes ce qui sera largement suffisant. Les couleurs sont recomposées en utilisant le rouge, le vert et le bleu, ce qu'on nomme le modèle RVB (RGB en anglais). En utilisant différentes combinaisons de ces trois couleurs fondamentales, on pourra recomposer toutes les couleurs voulues (ou presque). Par exemple pour obtenir la couleur d'un corail, on définira un vecteur couleur avec ces composantes :

```
glm::vec3 coral(1.0f, 0.5f, 0.31f);
```

Les couleurs que nous voyons ne sont pas les couleurs réelles des objets mais sont celles qui émanent de l'objet : les couleurs qui ne sont pas absorbées par l'objet sont réfléchies par l'objet et ce sont celles que nous percevons. Par exemple, la lumière du soleil est perçue comme une lumière blanche, qui est en réalité la combinaison de beaucoup de couleurs différentes (comme schématisé dans la figure suivante). Si l'on éclaire un objet bleu avec une lumière blanche, les couleurs autres que le bleu sont absorbées. La couleur bleue est au contraire réfléchie par l'objet et c'est elle que notre œil perçoit, l'objet nous apparaît donc bleu. La figure suivante montre ce phénomène sur un objet de couleur corail, qui réfléchit certaines couleurs avec des intensités différentes :



On voit que la lumière solaire est une composition de toutes les couleurs visibles et que l'objet absorbe une grande partie d'entre elles. Seules les couleurs représentant la couleur de l'objet sont réfléchies et ce sont celles-ci que l'on perçoit.

Ces règles de réflexion des couleurs sont reproduites dans le monde graphique. Lorsque l'on crée une source de lumière avec OpenGL, on lui donne une couleur. Comme dans l'exemple précédent, on peut lui donner une couleur blanche. Si l'on multiplie alors la couleur de la source lumineuse par la couleur d'un objet, la couleur résultante de l'objet est celle qui serait réfléchie par cet objet (et donc la couleur perçue par l'œil). Reprenons notre objet de couleur corail, et voyons comment nous calculerions sa couleur perçue dans le monde graphique. Nous retrouvons cette couleur perçue en effectuant une multiplication des composantes de chacune de ces deux couleurs :

```
glm::vec3 lightColor(1.0f, 1.0f, 1.0f);
```

```
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (1.0f, 0.5f, 0.31f);
```

On voit que l'objet absorbe une grande partie des couleurs de la source blanche, mais reflète une part de rouge, de vert et de bleu, en fonction de sa propre couleur. On peut donc définir la couleur d'un objet comme la quantité de chaque composante reflétée à partir d'une source de lumière blanche. Mais que se passerait-il avec une source de lumière verte ?

```
glm::vec3 lightColor(0.0f, 1.0f, 0.0f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (0.0f, 0.5f, 0.0f);
```

Comme on peut le voir, les composantes bleue et rouge sont nulles. La composante verte est absorbée pour moitié, mais l'autre moitié est reflétée par l'objet. La couleur perçue serait donc un vert assez foncé. La source ne comprenant que du vert, seule la composante verte peut être reflétée, le rouge et le bleu sont inexistants. L'objet corail apparaîtrait donc d'un vert très foncé. Voyons un autre exemple avec une source de couleur vert olive :

```
glm::vec3 lightColor(0.33f, 0.42f, 0.18f);
glm::vec3 toyColor(1.0f, 0.5f, 0.31f);
glm::vec3 result = lightColor * toyColor; // = (0.33f, 0.21f, 0.06f);
```

On constate que l'on obtient des couleurs inattendues pour un objet suivant la source de lumière qui l'éclaire. Ce n'est pas difficile d'être créatif avec les couleurs.

Essayons maintenant de créer une scène où nous pourrons tester ces différents effets.

I-A - Une scène éclairée

Dans les prochains chapitres, nous allons créer d'intéressants effets visuels en simulant un éclairage du monde réel par l'utilisation des couleurs. Puisque nous allons utiliser des sources lumineuses, nous souhaitons les afficher en tant qu'objets visuels dans la scène et ajouter au moins un objet pour simuler ces éclairages.

La première chose qu'il nous faut est un objet pour y projeter la lumière, nous utiliserons ce pauvre conteneur des tutoriels précédents. Il nous faut aussi un objet lumineux pour montrer d'où vient la lumière de la scène. Pour simplifier, nous représenterons aussi la source de lumière comme un cube (nous avons déjà les sommets pour un cube).

Les opérations d'initialisation d'un VBO, de définition des attributs de sommets vous sont maintenant familières, on ne s'attardera pas sur ce point. En cas de problème, revoyez les précédents tutoriels, y compris les exercices, avant de continuer.

Nous avons besoin d'un vertex shader pour afficher le conteneur. Les positions des sommets du conteneur restent les mêmes (nous n'utiliserons pas les coordonnées de textures pour cette fois), rien de nouveau donc dans le code. Nous utilisons une version allégée du vertex shader des chapitres précédents :

```
#version 330 core
layout (location = 0) in vec3 aPos;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

Assurez-vous de mettre à jour vos données de sommets et les pointeurs d'attributs de sommets pour coller à ce nouveau vertex shader (vous pouvez conserver les textures si vous le souhaitez, nous ne les utiliserons pas. Aussi cela peut être une bonne idée de repartir avec un nouveau projet).

Puisque nous voulons créer une lampe cube, nous allons créer un nouveau VAO pour cette lampe. Nous pourrions aussi utiliser le même VAO et simplement effectuer une transformation de la matrice de modèle, mais dans les prochains chapitres nous modifierons souvent les données de sommets et les pointeurs d'attributs de sommets de l'objet conteneur et nous ne voulons pas que l'objet lampe en soit affecté, nous créons donc un nouveau VAO :

```
unsigned int lightVAO;
glGenVertexArrays(1, &lightVAO);
glBindVertexArray(lightVAO);
// nous avons juste besoin de lier le VBO, les données du container sont déjà prêtes.
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// définit les attributs de sommets (seulement la position)
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

Le code doit vous paraître assez direct. Après avoir créé ces deux objets, le conteneur et la lampe, il nous faut coder le fragment shader :

```
#version 330 core
out vec4 FragColor;
uniform vec3 objectColor;
uniform vec3 lightColor;
void main()
{
    FragColor = vec4(lightColor * objectColor, 1.0);
}
```

Le fragment shader utilise une couleur objet et une couleur de lumière, définies avec des variables uniformes. Ensuite, on multiplie la couleur de la source avec la couleur de l'objet comme expliqué au début de ce chapitre. Le shader est assez simple à comprendre. Définissons la couleur de l'objet corail et la source blanche :

```
// ne pas oublier de lier le shader program avant d'affecter la variable uniforme
lightingShader.use();
lightingShader.setVec3("objectColor", 1.0f, 0.5f, 0.31f);
lightingShader.setVec3("lightColor",  1.0f, 1.0f, 1.0f);
```

À noter que si nous modifions le vertex shader ou le fragment shader, la lampe sera aussi modifiée ; au contraire, nous ne voulons pas que la couleur de la source lumineuse soit affectée par les calculs d'éclairage, mais plutôt qu'elle soit isolée du reste. Nous voulons maintenir un éclairage constant, et fonctionner réellement comme une source de lumière.

Dans ce but, nous allons créer un second ensemble de shaders que nous utiliserons pour dessiner la lampe, indépendants des shaders de l'éclairage. Ce vertex shader est le même que le vertex shader actuel, on se contentera de recopier le code source pour ce nouveau shader. Le fragment shader de la lampe assurera que la couleur de la lampe reste constante et de couleur blanche :

```
#version 330 core
out vec4 FragColor;
void main()
{
    FragColor = vec4(1.0); // définit les quatre composantes du vecteur à 1.0
}
```

Pour dessiner des objets, nous utiliserons le shader d'éclairage que nous venons de définir, alors que pour afficher la lampe, nous utiliserons le shader de la lampe. Au cours des prochains tutoriels, nous ferons évoluer les shaders d'éclairage pour parvenir à des résultats plus réalistes.

Le but principal de la lampe cube est de modéliser l'endroit d'où provient la lumière. Nous définirons la position de la source lumineuse dans la scène, mais ce n'est qu'une position sans réalité visuelle. Pour montrer la lampe réelle, nous dessinerons la lampe cube au même endroit que la source de lumière. Cela est réalisé en affichant l'objet lampe avec le shader de la lampe, cette lampe restera toujours blanche, quelles que soient les conditions d'éclairage de la scène.

Déclarons une variable globale vec3 qui représente l'emplacement de la source lumineuse dans l'espace monde :

```
glm::vec3 lightPos(1.2f, 1.0f, 2.0f);
```

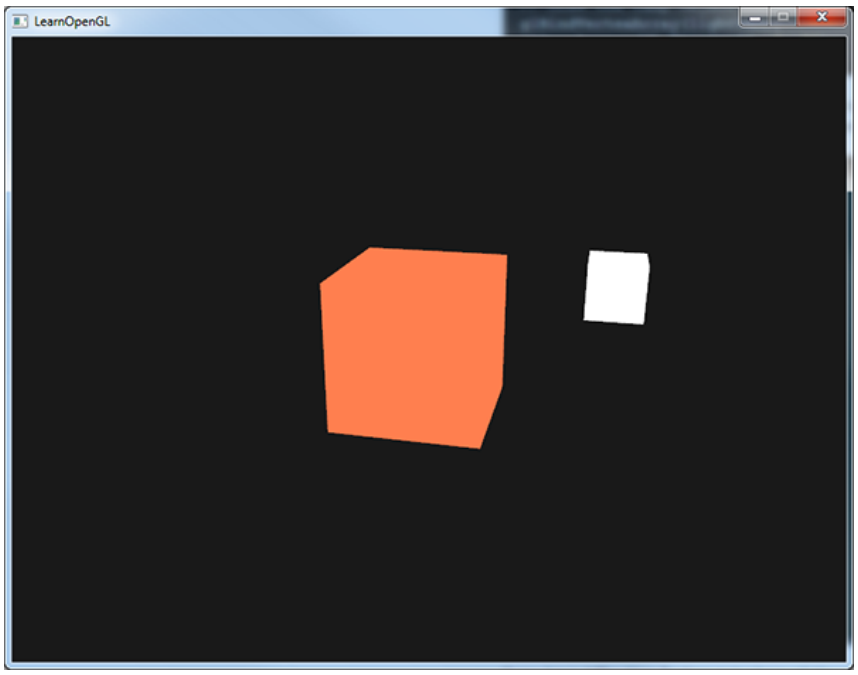
Ensuite, nous déplacerons le cube de la lampe à la position de la source lumineuse avant de l'afficher, et nous la réduirons de façon à la rendre plus discrète :

```
model = glm::mat4();
model = glm::translate(model, lightPos);
model = glm::scale(model, glm::vec3(0.2f));
```

Le code pour afficher la lampe ressemblera à cela :

```
lampShader.use();
// configurer les variables uniformes pour les matrices de modèle, vue et projection
...
// dessin de la lampe
glBindVertexArray(lightVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);
```

En replaçant ces lignes de code au bon endroit, nous aurons une application OpenGL proprement configurée pour nos tests d'éclairages. Si tout se passe bien, on obtient ce résultat :



Pas encore très spectaculaire, mais je vous promets des résultats plus intéressants par la suite.

Si vous rencontrez quelques difficultés pour assembler tous ces bouts de code, le code source est **ici**, inspectez l'ensemble attentivement.

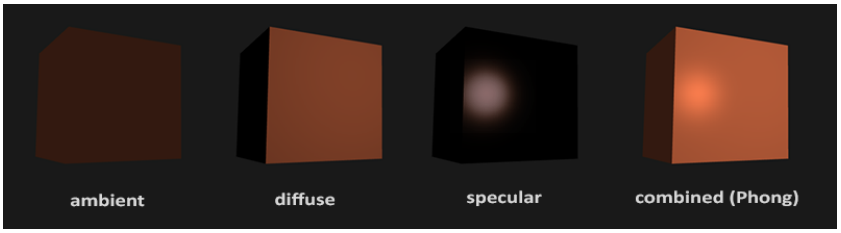
Après ces quelques éléments sur les couleurs dans une scène simple, le **chapitre suivant** commencera à produire des effets plus magiques.

I-B - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

II - Éclairage simple

La lumière dans le monde réel est extrêmement complexe et dépend de beaucoup de facteurs, lesquels ne peuvent pas être tous pris en compte avec la puissance de calcul limitée dont on dispose. Les éclairages dans OpenGL sont par conséquent fondés sur certaines approximations de la réalité en utilisant des modèles simplifiés qui sont bien plus faciles à gérer, mais qui donnent pourtant un résultat assez réaliste. Ces modèles d'éclairage sont basés sur des modèles physiques de la lumière. Un de ces modèles est appelé le modèle de Phong. Les éléments constitutifs de ce modèle sont les trois composantes suivantes : ambiante, diffuse et spéculaire. Vous pouvez voir ci-dessous à quoi ressemblent ces composantes :



- **composante ambiante** : même avec très peu de lumière, il en reste en général un peu (la lune, une lampe lointaine), et les objets ne sont presque jamais complètement noirs. Pour simuler cela, on utilise un éclairage ambiant qui donnera toujours une couleur aux objets.
- **composante diffuse** : on simule l'effet directionnel qu'une source lumineuse produit sur un objet. C'est la composante la plus visible d'un modèle d'éclairage. Plus un objet est face à la source de lumière, plus il apparaîtra clair.
- **composante spéculaire** : simule la réflexion d'une source sur tout ou partie d'un objet brillant. Les éclairages spéculaires donnent une couleur plus proche de celle de la source que de l'objet lui-même.

Pour créer des scènes visuellement intéressantes, il nous faudra utiliser ces trois composantes, commençons par la plus simple : l'éclairage ambiant.

II-A - Éclairage ambiant

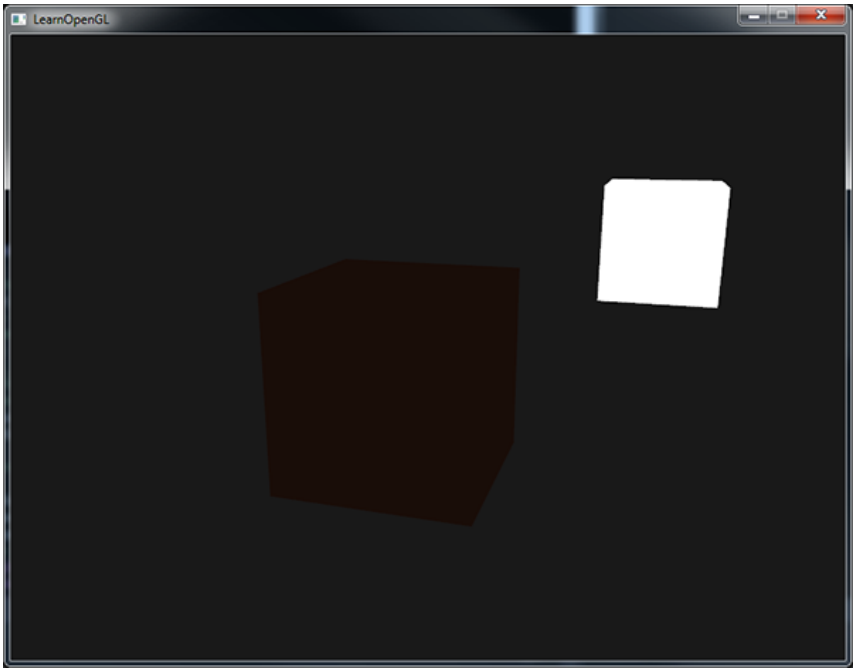
Le plus souvent, la lumière ne provient pas d'une seule source mais de plusieurs, réparties autour de nous, même si elles ne sont pas immédiatement perceptibles. Une des propriétés de la lumière est de se refléter et de se diffuser dans toutes les directions, et pas seulement dans le voisinage direct de la source. La lumière se réfléchit sur les surfaces des objets et a un impact sur l'éclairage des autres objets. Les algorithmes qui traitent cet effet sont appelés algorithmes d'illumination globale, mais ils se révèlent complexes et gourmands.

Puisque nous ne sommes pas très enclins à ces algorithmes complexes, nous utiliserons un modèle simple d'illumination globale : l'éclairage ambiant. Comme nous l'avons vu dans le chapitre précédent, on utilisera une petite source lumineuse de couleur constante que nous ajouterons à la couleur finale des fragments de l'objet, donnant ainsi l'impression qu'il y a toujours un peu de lumière, même en l'absence de source directe de lumière.

Ajouter un éclairage ambiant à la scène est assez facile. Nous multiplions la couleur de la lumière par un petit facteur constant, et multiplions ensuite le résultat par la couleur des objets pour obtenir la couleur du fragment :

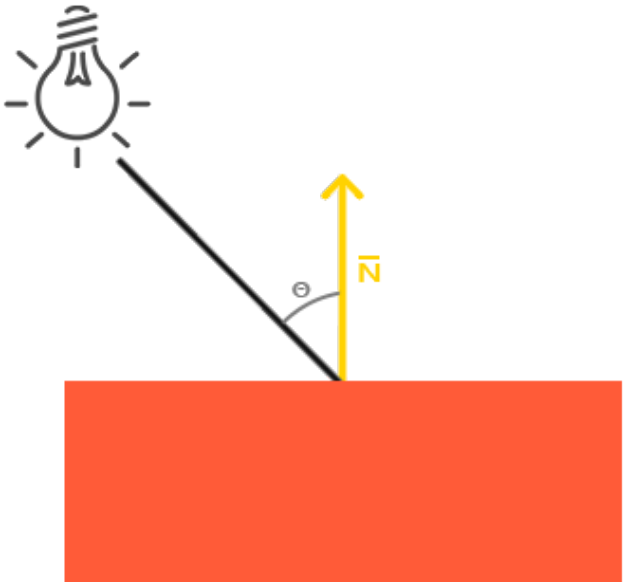
```
void main()
{
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;
    vec3 result = ambient * objectColor;
    FragColor = vec4(result, 1.0);
}
```

Si vous exécutez votre programme, vous noterez que la première étape de l'éclairage est appliquée à votre objet. L'objet est assez sombre, mais pas complètement grâce à cet éclairage ambiant (le cube lampe n'est pas affecté car il utilise un autre shader). Cela donne quelque chose comme ceci :



II-B - Éclairage diffus

L'éclairage ambiant ne produit pas des effets passionnants ; par contre, l'éclairage diffus va commencer à produire des effets visuels significatifs. Cet éclairage diffus rendra l'objet d'autant plus lumineux que ses fragments seront alignés avec les rayons lumineux de la source. Pour mieux comprendre cet effet, regardons l'image suivante :



On voit sur la gauche une source de lumière qui émet un rayon en direction d'un fragment de notre objet. Nous mesurons l'angle avec lequel ce rayon arrive sur l'objet. Si le rayon lumineux arrive perpendiculairement à la surface de l'objet, il aura l'effet maximum. Pour mesurer cet angle d'incidence, nous utilisons un vecteur appelé vecteur normal (ou normale), qui est un vecteur perpendiculaire à la surface (figuré ici par une flèche jaune) ; nous reverrons cela plus tard. L'angle d'incidence peut alors facilement être calculé au moyen du produit scalaire.

Vous vous rappelez du chapitre sur **les transformations** où nous avions vu que plus l'angle entre deux vecteurs unitaires est faible, plus le produit scalaire s'approche de la valeur 1. Si les deux vecteurs sont perpendiculaires, ce produit vaut 0. Ici, plus l'angle θ sera grand, moins la lumière aura d'effet sur la couleur de l'objet.

Notons que pour obtenir la valeur du cosinus de l'angle, nous devons travailler avec des vecteurs unitaires, il faudra donc s'assurer de normaliser les vecteurs sinon le produit scalaire donnerait une valeur différente.

Le produit scalaire retourne donc une valeur que nous utiliserons pour calculer l'effet de la lumière sur la couleur du fragment, résultant en des fragments éclairés différemment selon leur orientation par rapport aux rayons lumineux.

Pour calculer un éclairage diffus, il nous faudra :

- un vecteur normal : un vecteur perpendiculaire à la surface du fragment ;
- la direction de la lumière incidente, calculée comme la différence entre la position de la source et celle du fragment considéré.

II-B-1 - Vecteur normal (ou normale)

Une normale est un vecteur unitaire perpendiculaire à la surface d'un sommet. Mais comme un sommet ne possède pas de surface (c'est juste un point), on calcule le vecteur normal en utilisant les sommets voisins pour obtenir une surface. On peut utiliser le produit vectoriel pour calculer la normale des sommets de notre cube, mais comme ce cube n'est pas très compliqué, on peut simplement ajouter manuellement les normales aux données des sommets. Les nouvelles données de sommets se trouvent **ici**. Essayez de visualiser les normales comme des vecteurs perpendiculaires aux plans du cube (qui en compte 6).

Puisque nous avons ajouté des données au tableau des sommets, nous devons ajuster le vertex shader pour l'éclairage :

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
...
```

Maintenant que nous avons ajouté un vecteur normal à chaque sommet et mis à jour le vertex shader, il nous faut modifier les attributs de sommets. Notez que l'objet lampe utilise les mêmes sommets mais que pour cet objet on n'utilise pas les normales. Nous n'aurons pas à modifier les shaders de la lampe ni la configuration des attributs. Par contre, il faut mettre à jour la façon dont les données sont lues pour tenir compte de la nouvelle taille du tableau des sommets :

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

On n'utilisera que les trois premiers nombres pour chaque sommet en ignorant les trois derniers, mais il faut donner la taille de six entiers pour le stride.

Il peut paraître peu efficace d'utiliser le tableau complet pour la lampe, mais ces données sont déjà mémorisées dans le GPU pour le conteneur, nous n'utilisons pas de mémoire supplémentaire. Cela est plus efficace que de définir un nouveau VBO pour la lampe.

Tous les calculs d'éclairage sont effectués dans le fragment shader, il faut donc transmettre les normales du vertex shader au fragment shader.

```
out vec3 Normal;
void main()
{
 gl_Position = projection * view * model * vec4(aPos, 1.0);
 Normal = aNormal;
}
```

Il faut aussi déclarer cette nouvelle entrée dans le fragment shader :

```
in vec3 Normal;
```

II-B-2 - Calcul de la composante diffuse

Nous disposons d'une normale pour chaque sommet, mais nous devons encore calculer la position de la source et celle des fragments. Puisque la source ne change pas de position, on utilisera une variable uniforme dans le fragment shader :

```
uniform vec3 lightPos;
```

Ensuite, initialisons cette variable dans la boucle d'affichage (ou ailleurs). On utilise le vecteur lightPos déclaré dans le chapitre précédent comme position de la source :

```
lightingShader.setVec3("lightPos", lightPos);
```

Enfin, nous devons connaître la position du fragment considéré. Nous ferons tous les calculs d'éclairage dans l'espace monde, nous souhaitons donc connaître la position des sommets dans cet espace. En multipliant les attributs de position des sommets uniquement par la matrice de modèle (et pas les matrices de vue ni de projection), on obtiendra les coordonnées dans l'espace monde. Cela peut être réalisé dans le vertex shader, déclarons donc une variable de sortie et calculons ces coordonnées dans l'espace monde :

```
out vec3 FragPos;
out vec3 Normal;
void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0);
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = aNormal;
}
```

Et ajoutons une variable d'entrée dans le fragment shader :

```
in vec3 FragPos;
```

Nous avons maintenant toutes les variables nécessaires pour les calculs d'éclairage dans le fragment shader.

Calculons d'abord le vecteur représentant la direction de la lumière entre la source et la position du fragment. Nous avons montré que ce vecteur peut être calculé par la différence entre la position de la source et celle du fragment. Comme nous l'avons vu dans le chapitre sur les transformations, il suffit d'effectuer la différence entre ces deux vecteurs. Nous voulons également utiliser des vecteurs unitaires, par conséquent nous normalisons le vecteur de direction et la normale :

```
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
```

Pour calculer l'éclairage, on n'a pas besoin de la longueur ni de la position des vecteurs, mais seulement de leur direction. On peut donc utiliser des vecteurs unitaires pour simplifier

les calculs. Assurez-vous de normaliser les vecteurs pour effectuer ces calculs. Oublier cette opération est une erreur assez fréquente.

La composante diffuse de la lumière sur un fragment passe par le calcul du produit scalaire entre le vecteur norm et le vecteur lightDir. Le résultat est ensuite multiplié par la couleur de la lumière pour obtenir la composante diffuse, résultant en une lumière d'autant plus sombre que l'angle entre ces deux vecteurs est grand :

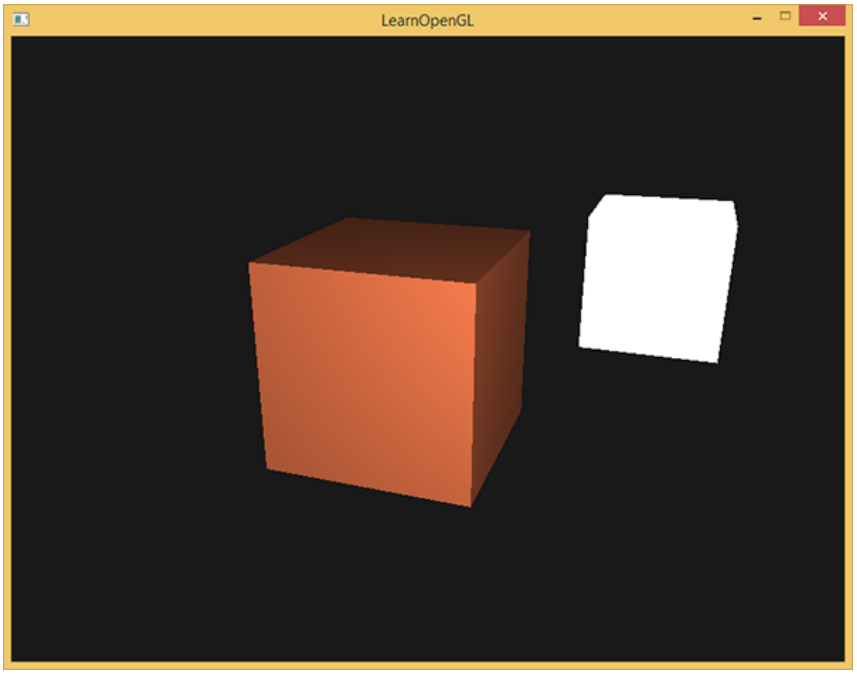
```
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuse = diff * lightColor;
```

Si l'angle entre les deux vecteurs est supérieur à 90°, le produit scalaire est négatif, ce qui donnerait une composante diffuse négative. Pour éviter cela, nous limitons le résultat aux valeurs positives en utilisant la fonction max. Les couleurs ne seront donc jamais négatives. Les couleurs négatives ne sont pas définies, mieux vaut s'en passer.

Nous disposons de la composante ambiante et de la composante diffuse de la lumière, nous ajoutons ces deux composantes et multiplions le résultat par la couleur de l'objet pour obtenir la couleur finale du fragment :

```
vec3 result = (ambient + diffuse) * objectColor;
FragColor = vec4(result, 1.0);
```

Si votre application et vos shaders sont corrects, vous devriez obtenir cette image :



On voit qu'avec la lumière diffuse, le cube commence à devenir plus réaliste. Essayez de visualiser les normales mentalement et tournez autour du cube pour voir que plus l'angle d'incidence est grand, plus la surface est sombre.

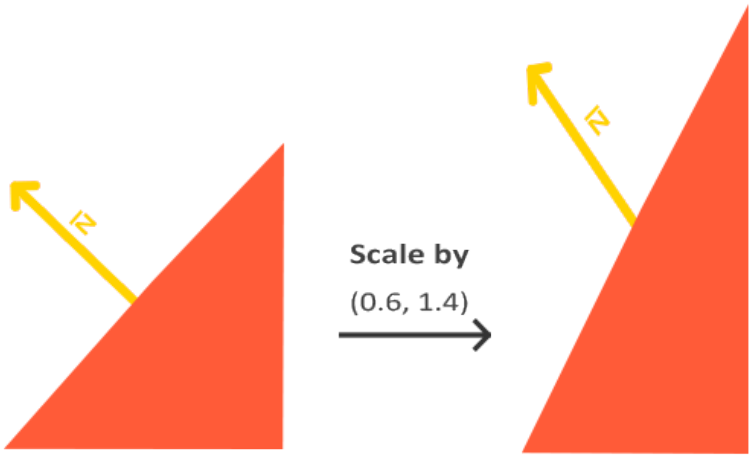
En cas de besoin, vous pourrez trouver le code complet de cette partie **ici**.

II-B-3 - Une dernière chose

Jusqu'ici, nous avons passé les normales directement du vertex shader au fragment shader. Cependant, les calculs que nous avons faits dans le fragment shader l'ont été avec les coordonnées de l'espace monde, ne devons-nous pas exprimer ces normales dans l'espace monde aussi ? Oui, mais ce n'est pas aussi simple qu'une multiplication par la matrice de modèle.

Tout d'abord, les normales ne représentent que des directions, sans position particulière dans l'espace. Ainsi, les normales n'ont pas de coordonnée homogène (la composante w de la position d'un sommet). Cela implique que les translations n'ont pas d'effet sur ces normales. Si nous souhaitons multiplier ces vecteurs par la matrice de modèle, il nous faut supprimer la partie translation de cette matrice en ne conservant que les trois premières lignes et les trois premières colonnes (on pourrait aussi donner la valeur 0 à la composante w de la normale et conserver une matrice 4x4). Les seules transformations que nous voulons effectuer sur les normales sont les homothéties et les rotations.

Ensuite, si la matrice de modèle effectuait une mise à l'échelle non uniforme, les normales ne seraient plus perpendiculaires aux surfaces, on ne pourrait donc pas utiliser une telle matrice de modèle pour transformer les normales. L'image suivante montre l'effet d'une telle matrice sur une normale :



En appliquant une mise à l'échelle non uniforme, les normales ne sont plus perpendiculaires aux surfaces, ce qui fausse le calcul de la lumière (une mise à l'échelle uniforme ne modifie pas la direction des normales — ni celle des surfaces — mais juste leur longueur, ce qui n'est pas gênant puisque l'on normalise ces vecteurs).

L'astuce consiste à utiliser une matrice de modèle différente, spécifiquement conçue pour les normales. Cette matrice est appelée matrice de normale et utilise quelques opérations de l'algèbre linéaire pour éviter cet effet indésirable. Si vous souhaitez savoir comment est calculée cette matrice, je vous suggère de consulter cet **article**.

La matrice de normale est définie comme « la transposée de l'inverse du coin en haut à gauche de la matrice de modèle ». Pas simple, et si vous ne comprenez pas ce que cela veut dire, pas de souci : nous n'avons pas parlé de l'inversion ni de la transposition des matrices. Notez que la plupart des ressources sur le sujet définissent la matrice de normale en appliquant ces opérations sur la matrice modèle-vue, mais puisque nous travaillons dans l'espace monde (et non pas dans l'espace de vue), nous n'utiliserons que la matrice de modèle.

Dans le vertex shader, on peut générer cette matrice de normale nous-mêmes en utilisant les fonctions *inverse* et *transpose*. Notons que l'on projette notre matrice sur une matrice 3x3 pour écarter la translation et pouvoir la multiplier par un vecteur de normale vec3 :

```
Normal = mat3(transpose(inverse(model))) * aNormal;
```

Dans le paragraphe consacré à la lumière diffuse, l'éclairage était correct, mais nous n'avions réalisé aucune opération de mise à l'échelle sur l'objet et nous n'avions donc pas besoin d'une matrice de normale, la matrice de modèle était suffisante. Si par contre vous utilisez une mise à l'échelle non uniforme, il est essentiel de multiplier les normales par la matrice de normale.

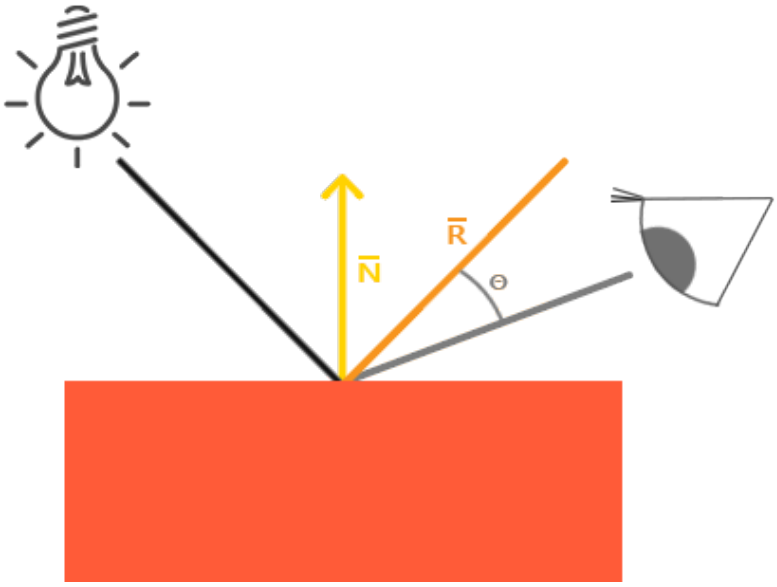
⚠ *L'inversion de matrices est une opération coûteuse, même pour les shaders, essayez d'éviter autant que possible ces opérations dans les shaders, car elles seront effectuées sur chaque sommet de la scène. Cela est intéressant pour apprendre, mais pour une application efficace,*

on effectuera ces calculs sur le CPU et on transmettra le résultat par une variable uniforme avant l'affichage (comme pour la matrice de modèle).

II-C - Éclairage spéculaire

Si vous n'êtes pas trop fatigué avec ces calculs d'éclairage, nous pouvons aborder l'éclairage spéculaire pour compléter le modèle de Phong.

De même que l'éclairage diffus, l'éclairage spéculaire est basé sur la direction des rayons lumineux et les normales, mais cette fois il faut aussi tenir compte de l'angle avec lequel la surface du fragment est perçue, c'est-à-dire du point de vue du spectateur. L'éclairage spéculaire utilise les propriétés de réflexion de la lumière. Si nous considérons la surface d'un objet comme un miroir, l'éclairage spéculaire sera le plus fort dans la direction de réflexion. On peut voir cela sur la figure suivante :



On calcule le vecteur de réflexion en symétrisant la direction de la lumière incidente par rapport à la normale. On calcule ensuite la distance angulaire entre cette direction de réflexion et la direction de vue. Plus ces deux directions seront proches, plus la composante spéculaire sera grande. L'effet résultant sera de voir une tache de lumière dans la direction de réflexion de la lumière incidente.

Le vecteur de vue est une variable de plus, nécessaire pour l'éclairage spéculaire, que nous pouvons calculer en utilisant l'espace monde du spectateur et la position des fragments. Ensuite, nous calculerons l'intensité spéculaire, nous multiplierons cela par la couleur de la lumière et on ajoutera le résultat à la lumière ambiante et à la lumière diffuse.

Nous choisissons de faire les calculs d'éclairage dans l'espace monde, mais la plupart des développeurs lui préfèrent l'espace de vue. L'avantage d'utiliser l'espace de vue est que la position du spectateur est toujours (0, 0, 0) et ainsi la position du spectateur est déjà connue. Cependant, je trouve que le calcul dans l'espace monde est plus intuitif pour apprendre. Si vous souhaitez faire les calculs dans l'espace de vue, il faut aussi transformer tous les vecteurs avec la matrice de vue (n'oubliez pas de modifier aussi la matrice de normales).

Pour obtenir les coordonnées du spectateur dans l'espace monde, il suffit d'utiliser le vecteur position de la caméra. Ajoutons donc une nouvelle variable uniforme au fragment shader et passons-lui la position de la caméra :

```
uniform vec3 viewPos;
```

```
lightingShader.setVec3("viewPos", camera.Position);
```

Nous disposons maintenant des variables nécessaires pour calculer l'intensité spéculaire. Premièrement, nous définissons une valeur d'intensité spéculaire pour donner à l'éclairage spéculaire une couleur moyennement brillante pour obtenir un effet modéré :

```
float specularStrength = 0.5;
```

Si nous donnions la valeur 1.0 à ce paramètre, nous aurions une composante très brillante, ce qui est un peu trop pour notre cube corail. Dans le prochain chapitre, nous verrons comment définir correctement ces propriétés et comment cela affecte les objets. Ensuite, nous calculons le vecteur de direction de la vue et le vecteur réflexion :

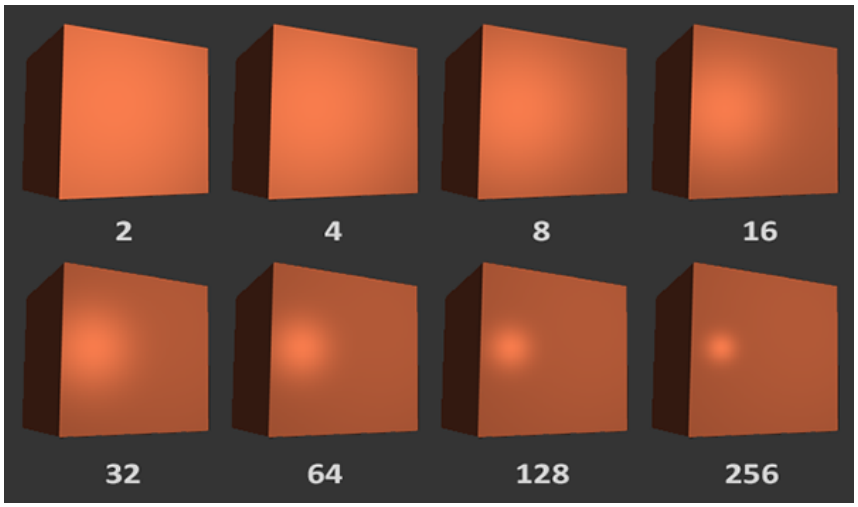
```
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
```

Notez que nous inversons le vecteur lightDir. La fonction reflect() demande à ce que le premier vecteur pointe de la source de lumière vers le fragment, alors que le vecteur lightDir pointe dans le sens inverse. Le second vecteur est le vecteur normal au fragment.

Il reste à calculer l'intensité de la composante spéculaire. On le réalise avec la formule suivante :

```
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
vec3 specular = specularStrength * spec * lightColor;
```

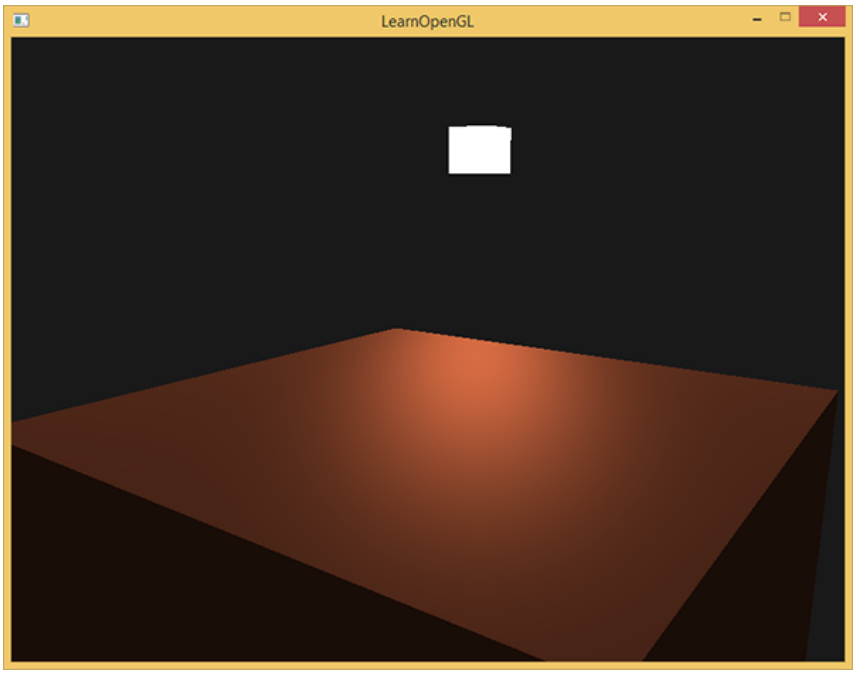
On calcule d'abord le produit scalaire entre la direction de vue et la direction de réflexion (en s'assurant que ce produit n'est pas négatif), et on élève le résultat à la puissance 32. Cette valeur 32 est la valeur de la brillance. Plus grande est cette valeur pour un objet, plus la lumière sera reflétée dans une direction précise et non diffusée dans toutes les directions. On peut voir dans l'image suivante l'influence de ce paramètre :



On ne souhaite pas que la composante spéculaire soit trop dominante, on choisit donc une valeur intermédiaire : 32. Il faut enfin ajouter cette composante spéculaire aux deux autres et multiplier le résultat par la couleur de l'objet :

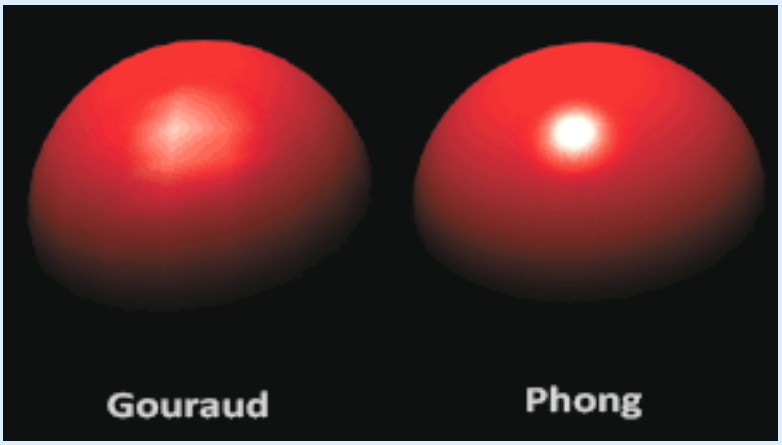
```
vec3 result = (ambient + diffuse + specular) * objectColor;
FragColor = vec4(result, 1.0);
```

Nous avons donc calculé toutes les composantes du modèle de Phong. Voilà le résultat de ces calculs avec un certain point de vue :



Le code complet de l'application se trouve **ici**.

Au commencement des shaders d'éclairage, les développeurs implémentaient le modèle de Phong dans le vertex shader. Cela est beaucoup plus efficace, car il y a généralement beaucoup moins de sommets que de fragments, les calculs d'éclairage sont donc moins nombreux. Cependant, la couleur résultante dans le vertex shader est celle d'un sommet seul et la couleur des fragments sera le résultat de l'interpolation des couleurs des sommets situés autour. Le résultat n'est pas très réaliste sauf si l'on dispose de nombreux sommets :



Si l'on implémente le modèle de Phong dans le vertex shader, on obtient le modèle de Gouraud. On voit qu'avec cette interpolation, la lumière semble moins précise. Le modèle de Phong donne un résultat plus net.

Vous commencez à voir la puissance des shaders. Avec peu d'information, les shaders sont en mesure de calculer comment la lumière affecte les couleurs des fragments de nos objets. Dans les chapitres suivants, nous étudierons plus profondément ce que l'on peut faire avec ces modèles d'éclairage.

II-D - Exercices

- Jusqu'à présent, la source de lumière était statique. Essayez de déplacer cette source autour de la scène, le résultat vous donnera un bon aperçu du modèle d'éclairage de Phong : **solution**.
- Modifiez les intensités des composantes ambiante, diffuse et spéculaire et observez le résultat. Modifiez aussi le facteur de brillance. Essayez de comprendre l'influence de chaque paramètre.
- Codez le modèle de Phong dans l'espace de vue plutôt que dans l'espace monde : **solution**.
- Implémentez le modèle de Gouraud à la place du modèle de Phong. Vous devriez voir une **lumière moins précise** (en particulier la composante spéculaire). Essayez de comprendre pourquoi vous obtenez un résultat étrange : **solution**.

II-E - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

III - Matériaux

Dans le monde réel, chaque objet réagit différemment à la lumière. Les objets en acier sont souvent plus brillants qu'un vase en argile, un conteneur en bois ne réagira pas à la lumière comme un conteneur métallique. Chaque objet donnera aussi des reflets différents. Certains refléteront la lumière sans trop la disperser, ce qui donnera de petits reflets intenses, d'autres donneront des reflets plus larges. Pour simuler ces différents comportements dans OpenGL, on définit des propriétés spécifiques pour chaque objet.

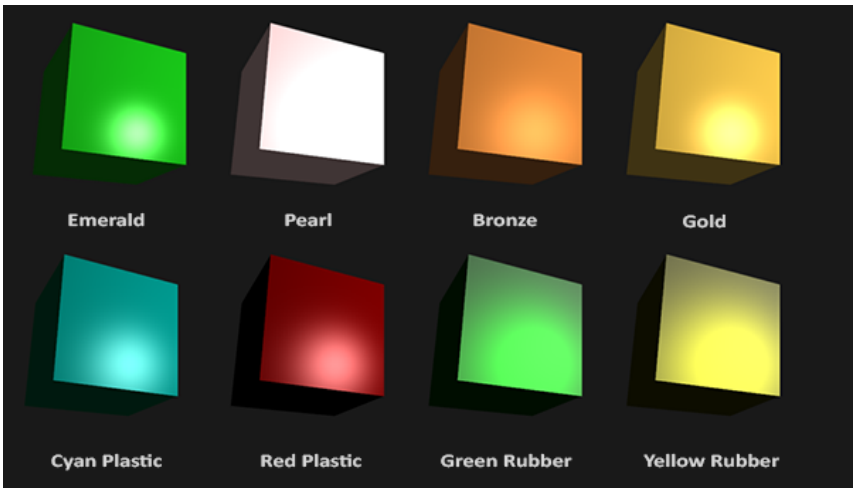
Dans le chapitre précédent, nous avons défini un objet et une couleur de lumière pour obtenir une image de l'objet, combinée avec une composante diffuse et une composante spéculaire. Pour décrire des objets, on peut définir une couleur de matériau pour chacune des trois composantes, ambiante, diffuse et spéculaire. On a ainsi un contrôle très fin sur la couleur produite par les objets. Ajoutons une composante de brillance à ces trois couleurs et nous aurons toutes les propriétés pour définir un matériau :

```
#version 330 core
struct Material {
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shininess;
};
uniform Material material;
```

Dans le fragment shader, nous créons une structure pour y placer les propriétés de matériau de l'objet. On peut aussi les représenter une à une avec des variables uniformes, mais l'utilisation d'une structure permet une meilleure organisation. Nous définissons le contenu de cette structure et déclarons ensuite une variable uniforme de ce nouveau type structure.

Comme on peut le voir, on utilise une couleur pour chacune des composantes du modèle de Phong. Le vecteur matériau ambient définit quelle couleur cet objet reflétera sous un éclairage ambiant, ce qui est en général la couleur de l'objet. Le vecteur diffuse définira la couleur de l'objet sous un éclairage diffus. Cette couleur est définie par la couleur de l'objet (comme pour l'éclairage ambiant). Le vecteur specular définit la couleur de la lumière reflétée (voire une couleur de reflet spécifique de l'objet). Enfin, la composante shininess (brillance) définira la netteté et la largeur de la tache lumineuse de la composante spéculaire.

Avec ces quatre composantes qui définissent les propriétés de matériau d'un objet, on peut simuler beaucoup de vrais matériaux. Le tableau disponible sur la page **devernay.free.fr** donne plusieurs propriétés de matériaux pour simuler la diversité du monde réel. L'image suivante montre l'effet de quelques matériaux sur notre cube (émeraude, perle, bronze, or, plastique cyan, plastique rouge, gomme verte, gomme jaune) :



Comme vous pouvez le voir, en spécifiant correctement les propriétés de matériau d'un objet, on change notre perception. Les effets sont nettement perceptibles, mais pour des effets plus réalistes, nous aurons besoin de formes plus complexes qu'un simple cube. Dans la section suivante, nous apprendrons à utiliser des formes plus complexes.

Choisir correctement les matériaux est un défi qui requiert de l'expérience et il n'est pas rare de dégrader la qualité visuelle d'un objet avec un matériau mal conçu.

Essayons maintenant de coder ces propriétés de matériau dans les shaders.

III-A - Mise en place des matériaux

Nous avons créé une structure Material dans le fragment shader, on doit donc maintenant modifier le calcul d'éclairage pour tenir compte du matériau. On peut accéder aux propriétés du matériau par la variable uniforme material :

```
void main()
{
    // ambient
    vec3 ambient = lightColor * material.ambient;
    // diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = lightColor * (diff * material.diffuse);
    // specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    vec3 specular = lightColor * (spec * material.specular);
    vec3 result = ambient + diffuse + specular;
    FragColor = vec4(result, 1.0);
}
```

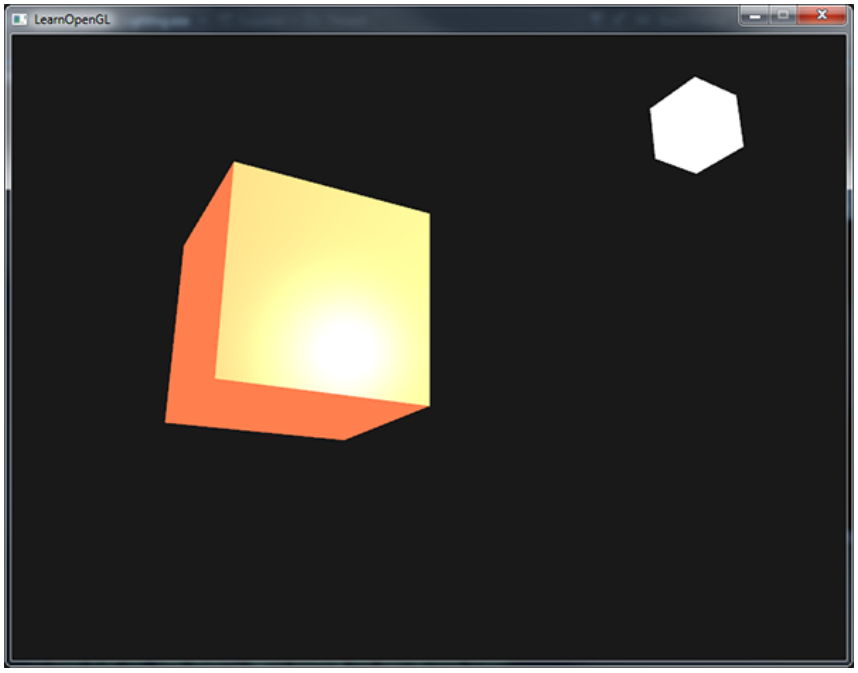
Toutes les propriétés du matériau sont accessibles par la variable material, et nous calculons la couleur finale en tenant compte des propriétés du matériau. Chacun des attributs de matériau est multiplié par la composante d'éclairage correspondante.

On peut définir le matériau d'un objet au moyen de la variable uniforme. Une structure en GLSL n'est pas un cas particulier. Une structure intervient pour encapsuler plusieurs variables uniformes, nous pouvons les utiliser comme des variables uniformes individuelles, préfixées par le nom de la variable structure :

```
lightingShader.setVec3("material.ambient",  1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.diffuse",  1.0f, 0.5f, 0.31f);
lightingShader.setVec3("material.specular", 0.5f, 0.5f, 0.5f);
lightingShader.setFloat("material.shininess", 32.0f);
```

Nous choisissons les composantes ambiante et diffuse de la couleur de l'objet et la composante spéculaire d'une couleur moyennement claire. Nous gardons aussi la brillance à la valeur 32. On pourra facilement modifier ces valeurs dans l'application.

Vous devriez obtenir quelque chose comme ça :



Ce n'est pas encore vraiment ça ?

III-B - Propriétés des sources lumineuses

L'objet est bien trop brillant, car les couleurs des composantes ambiante, diffuse et spéculaire sont reflétées avec trop d'intensité. Les sources de lumière ont aussi des intensités différentes pour chacune de leurs composantes (ambiante, diffuse, spéculaire). Dans le chapitre précédent, nous avions résolu cela par l'utilisation d'une variable d'intensité. De façon similaire, nous définirons un vecteur d'intensité pour chacune des trois composantes de la source. En visualisant `lightColor` comme un `vec3(1.0)` le code serait celui-ci :

```
vec3 ambient  = vec3(1.0) * material.ambient;
vec3 diffuse  = vec3(1.0) * (diff * material.diffuse);
vec3 specular = vec3(1.0) * (spec * material.specular);
```

Ainsi, chaque propriété du matériau rend pleinement l'intensité de chaque composante de la lumière. Ces vecteurs vec3(1.0) peuvent être modifiés séparément pour chacune des composantes de la source, c'est bien ce que nous voulons. Pour l'instant, la composante ambiante de l'objet est pleinement influencée par la couleur du cube, mais cette composante ambiante ne devrait pas avoir autant d'importance sur la couleur finale du cube, il faut donc la restreindre en fixant l'intensité de la lumière ambiante à une valeur plus faible :

```
vec3 ambient = vec3(0.1) * material.ambient;
```

On peut modifier les composantes diffuse et spéculaire de la source lumineuse de la même façon. C'est ce que nous avions fait dans le chapitre précédent ; nous avions déjà créé des propriétés différentes pour chaque composante de la source. Nous voulons créer quelque chose de similaire à la structure `material`, pour la source de lumière :

```
struct Light {
 vec3 position;
 vec3 ambient;
```

```
    vec3 diffuse;
    vec3 specular;
};
uniform Light light;
```

Une source de lumière aura différentes intensités pour chacune de ses composantes. La composante ambiante aura plutôt une intensité assez faible, car, en général, elle n'est pas dominante. Habituellement, la composante diffuse de la source prend la couleur exacte que nous voudrons donner à la source : souvent une couleur blanche brillante. La composante spéculaire est en général définie par un vec3(1.0) brillant à pleine intensité. Notez que nous avons ajouté la position de la source à la structure Light.

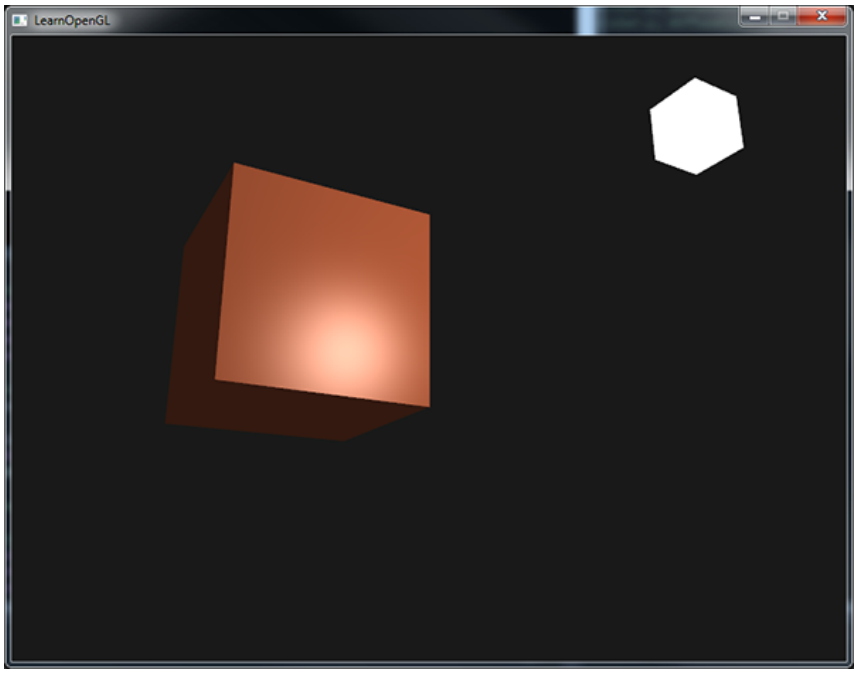
Comme pour la variable uniforme material, nous mettrons à jour le fragment shader :

```
vec3 ambient  = light.ambient * material.ambient;
vec3 diffuse  = light.diffuse * (diff * material.diffuse);
vec3 specular = light.specular * (spec * material.specular);
```

Nous définirons les intensités des composantes de la source dans l'application :

```
lightingShader.setVec3("light.ambient",  0.2f, 0.2f, 0.2f);
lightingShader.setVec3("light.diffuse",  0.5f, 0.5f, 0.5f); // assombri un peu la lumière pour
 correspondre à la scène
lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0f);
```

Dès lors, nous avons modulé l'influence de la lumière sur le matériau de l'objet et nous obtenons un effet plus proche de celui du chapitre précédent. Nous avons désormais le contrôle total sur la lumière et le matériau de l'objet :



Modifier l'aspect visuel de l'objet est très facile maintenant. Pimentons un peu les choses !

III-C - Différentes couleurs de sources lumineuses

Jusqu'ici nous n'avions modifié que l'intensité des composantes de la source, en choisissant des couleurs allant du blanc au noir et passant par le gris, n'affectant donc pas la couleur des objets, mais seulement leur intensité d'éclairage. Ayant accès aux propriétés de la source, on peut modifier les couleurs de ses composantes en fonction du temps pour obtenir des effets intéressants. Puisque tout est en place dans le fragment shader, changer les couleurs est facile et donne immédiatement des effets sympathiques :

Cliquer sur ce lien pour lancer l'animation

Comme on le voit, la couleur de la source influence énormément la couleur de l'objet. La couleur de la source influence directement les couleurs qui sont reflétées par l'objet (rappelez-vous le **chapitre 12** sur les couleurs), et la couleur perçue est donc influencée par la couleur de la source.

On peut facilement modifier les couleurs en fonction du temps au moyen des fonctions sin() et glfwGetTime() :

```
glm::vec3 lightColor;
lightColor.x = sin(glfwGetTime() * 2.0f);
lightColor.y = sin(glfwGetTime() * 0.7f);
lightColor.z = sin(glfwGetTime() * 1.3f);
glm::vec3 diffuseColor = lightColor   * glm::vec3(0.5f); // decrease the influence
glm::vec3 ambientColor = diffuseColor * glm::vec3(0.2f); // low influence
lightingShader.setVec3("light.ambient", ambientColor);
lightingShader.setVec3("light.diffuse", diffuseColor);
```

Testez vous-mêmes différents éclairages et différents matériaux et voyez comment cela change l'aspect des objets. Vous trouverez le code source de l'application **ici**.

III-D - Exercices

- Pourrez-vous simuler l'un des matériaux réels en définissant leurs propriétés comme nous l'avons vu au début du chapitre ? Noter que le **tableau** des valeurs ambiantes est différent de celui des valeurs diffuses, ils n'ont pas tenu compte des intensités de la source. Pour définir correctement ces valeurs, vous devrez définir les intensités de la source avec des vec3(1.0) pour obtenir la même sortie : **solution** pour un conteneur en plastique de couleur cyan.

III-E - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

IV - Textures de lumière (lighting maps)

Dans le **chapitre précédent**, nous avons montré l'intérêt de disposer pour un objet d'un matériau unique, caractérisant la façon dont l'objet réagit à la lumière. Cela permet de donner à chaque objet une apparence qui lui est propre, et le distingue des autres objets dans une scène éclairée, mais cela n'offre pas encore assez de souplesse quant à l'aspect visuel d'un objet.

Précédemment, nous avons défini un matériau pour un objet entier, mais en réalité, les objets ne sont pas constitués d'un seul matériau, mais de plusieurs. Prenez par exemple une voiture : la carrosserie est de nature brillante, les vitres reflètent l'extérieur de la voiture, les pneus sont ternes, les jantes sont très brillantes (si on les nettoie, bien sûr). Les couleurs ambiante et diffuse de la voiture ne sont pas non plus les mêmes pour toute la voiture. On voit bien qu'un objet doit avoir des propriétés de matériau différentes pour chacune de ses parties.

Les données de matériau du chapitre précédent ne sont donc suffisantes que pour les objets les plus simplistes et nous devons donc améliorer notre structure de données en introduisant des textures diffuse et spéculaire. Cela nous permettra de paramétrer avec plus de précision la composante spéculaire et la composante diffuse (et indirectement la composante ambiante puisqu'elle est presque toujours la même que la composante diffuse).

IV-A - Textures de lumière diffuse

Nous cherchons un moyen de définir la couleur diffuse d'un objet pour chaque fragment individuellement. Un moyen de retrouver la couleur en fonction de la position d'un fragment d'un objet.

Cela devrait vous paraître assez familier et pour être honnête, nous avons utilisé un tel système depuis quelque temps déjà. Cela ressemble à une texture, ce que nous avons présenté en détail dans l'un de nos **précédents tutoriels** et c'est bien cela : une texture. Nous utilisons un nom différent pour un même principe : utiliser une image plaquée sur l'objet, que l'on peut indexer pour définir la couleur de chaque fragment. Habituellement, on appelle cela une texture de lumière diffuse (diffuse map), car une telle texture représente toutes les couleurs diffuses de l'objet.

Pour montrer ces textures de lumière diffuse, nous allons utiliser l'**image suivante** d'un conteneur en bois muni d'un bord en acier :

L'utilisation d'une texture de lumière diffuse dans un shader utilise les mêmes principes que ceux expliqués dans le tutoriel consacré aux textures. Cependant, nous mémoriserons la texture comme un `sampler2D` à l'intérieur de la structure `Material`. Nous remplaçons le vecteur `vec3` défini pour la couleur diffuse par notre texture de lumière diffuse.

> ⚠ *Rappelez-vous que `sampler2D` est un type opaque, ce qui signifie qu'on ne peut pas instancier un tel type de données, mais seulement définir les variables de ces types comme uniformes. Si l'on instanciait cette structure autrement que par une variable uniforme (comme un paramètre de fonction), GLSL donnerait des erreurs étranges : la même chose s'applique à tous les types opaques.*

Nous allons aussi supprimer le vecteur de composante ambiante, car sa couleur est presque dans tous les cas la même que celle de la composante diffuse et il n'est pas utile de la mémoriser en plus.

```
struct Material {
```

```
    sampler2D diffuse;
    vec3      specular;
    float     shininess;
};
…
in vec2 TexCoords;
```

Si vous êtes très rigoureux et souhaitez conserver la composante ambiante du matériau avec une valeur différente, il suffit de conserver ce vecteur, mais la composante ambiante sera la même pour tout l'objet. Pour obtenir une composante ambiante différente pour chaque fragment, il vous faudra utiliser une texture de lumière ambiante.

Nous aurons besoin de coordonnées de textures dans le fragment shader, nous déclarons donc une nouvelle variable d'entrée. Il suffit ensuite d'échantillonner la texture pour retrouver la couleur diffuse pour chaque fragment :

```
vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuse, TexCoords));
```

Et n'oublions pas de définir la composante ambiante du matériau égale à sa composante diffuse :

```
vec3 ambient = light.ambient * vec3(texture(material.diffuse, TexCoords));
```

C'est suffisant pour utiliser une texture de lumière diffuse. Rien de bien nouveau, mais l'effet visuel en est très nettement amélioré. Pour que cela fonctionne, il nous faut mettre à jour les données de sommets avec les coordonnées de texture, les transférer comme attributs de sommets au fragment shader, charger la texture et lier l'unité de texture correspondante.

Les nouvelles données de sommets sont disponibles **ici**. Ces données incluent les positions, les normales et les coordonnées de texture pour chaque sommet du cube. Mettons à jour le vertex shader pour prendre les coordonnées de texture comme attributs de sommets et passons-les au fragment shader :

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;
…
out vec2 TexCoords;
void main()
{
…
    TexCoords = aTexCoords;
}
```

Assurez-vous de mettre à jour les pointeurs d'attributs de sommets des VAOs pour tenir compte de ces nouvelles données et chargez l'image du conteneur comme texture. Avant d'afficher le conteneur, assignons l'unité de texture à l'échantillonneur uniforme `material.diffuse` et lions la texture du conteneur à cette unité :

```
lightingShader.setInt("material.diffuse", 0);
…
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, diffuseMap);
```

L'utilisation de la texture de lumière diffuse améliore très nettement l'affichage et le conteneur devient réellement brillant. Le vôtre devrait ressembler à cela :

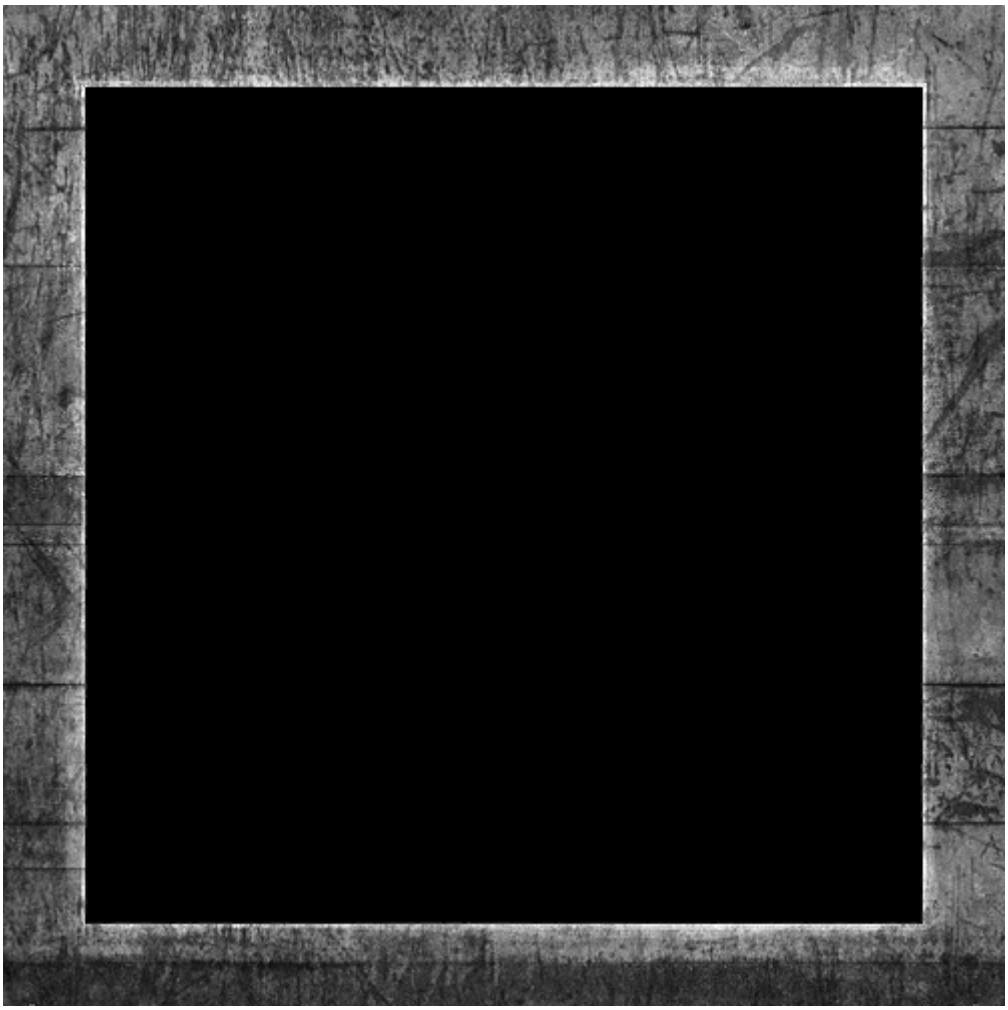
Vous trouverez le code complet de l'application **ici**.

IV-B - Textures de lumière spéculaire

Vous avez probablement remarqué que la composante spéculaire paraît un peu forte, car notre conteneur est composé de bois, et ne devrait renvoyer que peu de lumière. Nous pourrions rétablir cela en donnant à la composante spéculaire du matériau une valeur nulle, mais dans ce cas, les bords métalliques ne refléteraient plus rien, alors que l'acier est un matériau assez brillant. À nouveau, nous voulons contrôler quelles parties de l'objet doivent renvoyer de la lumière avec une intensité paramétrable.

Ce problème ressemble vraiment à celui traité précédemment pour la lumière diffuse.

On peut aussi définir une texture pour la composante spéculaire. Nous devons donc générer une texture noire et blanche (ou en couleur si vous le souhaitez) qui définit l'intensité de la propriété spéculaire du matériau pour chaque partie de l'objet. Un exemple de **texture spéculaire** est montré sur la figure suivante :



L'intensité de la composante spéculaire est déterminée par la brillance de chaque pixel de l'image. Chaque pixel de l'image de la texture spéculaire peut être vu comme un vecteur de couleur, où le noir est le vecteur `vec3(0.0)`, le gris un vecteur `vec3(0.5)` par exemple. Dans le fragment shader, on échantillonnera la valeur de la couleur et on la multipliera par l'intensité de la composante spéculaire de la source de lumière. Plus un pixel sera blanc, plus le résultat de la multiplication sera grand et plus la composante spéculaire sera intense.

Le conteneur est surtout composé de bois, ce matériau ne reflète que très peu la lumière et donc la partie correspondante de la texture est noire, la partie en bois ne donnera aucune composante spéculaire. Le bord du conteneur est en acier et aura des intensités de lumière spéculaire variables, l'acier étant très brillant, et les défauts, beaucoup moins.

En réalité, le bois donne aussi des reflets, mais peu intenses, et pour des raisons pédagogiques nous supposerons que le bois ne donne aucune composante de lumière spéculaire.

En utilisant des outils comme **Photoshop** ou **Gimp,** il est assez facile de transformer une texture diffuse en image spéculaire, en effaçant certaines parties, puis en convertissant l'image en nuances de gris et en jouant sur le contraste.

IV-C - Échantillonnage de texture spéculaire

Une texture de lumière spéculaire est juste comme toute autre texture et le code est similaire à celui utilisé pour les textures de lumière diffuse. Chargez l'image et générez un objet texture. Puisque nous utilisons une seconde texture

dans le même fragment shader, il faut une seconde unité de texture pour la texture de lumière spéculaire (voir le **chapitre sur les textures**), et ensuite il nous faut lier l'unité correcte avant d'effectuer le rendu :

```
lightingShader.setInt("material.specular", 1);
…
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, specularMap);
```

Mettons à jour les propriétés du matériau du fragment shader pour prendre un `sampler2D` comme composante spéculaire à la place d'un `vec3` :

```
struct Material {
    sampler2D diffuse;
    sampler2D specular;
    float     shininess;
};
```

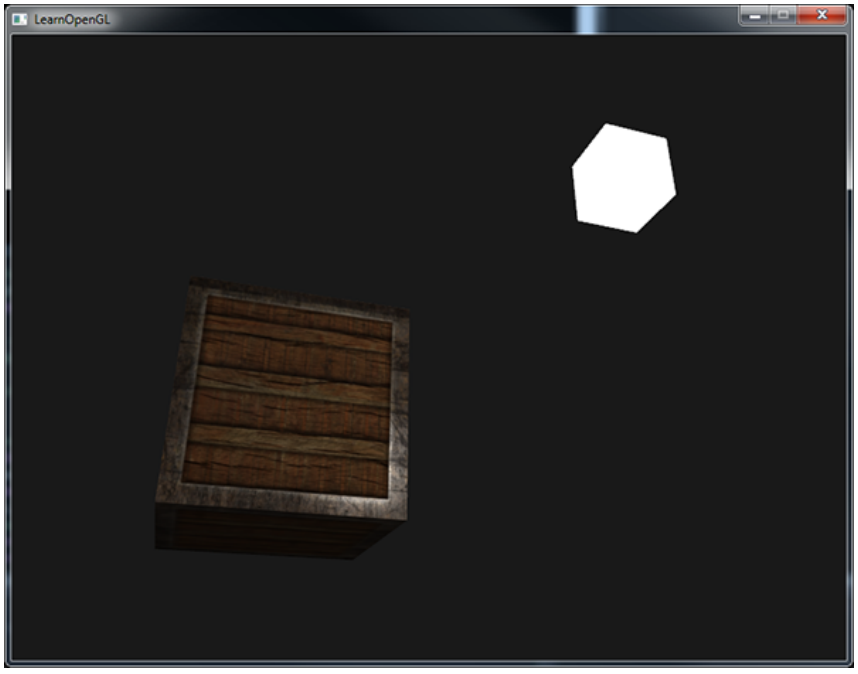
Et enfin, nous échantillonnons la texture spéculaire pour déterminer l'intensité spéculaire du fragment :

```
vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
FragColor = vec4(ambient + diffuse + specular, 1.0);
```

En utilisant une texture de lumière spéculaire, on peut définir très précisément quelles parties d'un objet sont brillantes et avec quelle intensité. Cela nous donne un niveau de contrôle supplémentaire.

Si vous le souhaitez, vous pouvez aussi utiliser les couleurs réelles de l'image pour la texture spéculaire, pour contrôler l'intensité, mais aussi la couleur de la lumière spéculaire. Mais, en réalité, la couleur des reflets est essentiellement déterminée par celle de la source de lumière et vous n'aurez donc pas des effets très réalistes (et c'est pourquoi on s'en tient à l'intensité et que l'on utilise des images en nuances de gris).

En lançant l'application, on voit bien que le matériau du conteneur ressemble beaucoup à celui d'un vrai conteneur en bois avec des bords métalliques :



Le code de l'application se trouve **ici**.

L'utilisation de textures pour la lumière diffuse et la lumière spéculaire permet d'ajouter de nombreux détails à des objets assez simples. On peut même augmenter cette précision en utilisant d'autres textures de matériau comme les textures de normales, ou encore des textures pour la réflexion, mais nous réservons cela pour des chapitres ultérieurs.

IV-D - Exercices

- Jouez avec les composantes de la source de lumière et voyez comment cela modifie l'aspect de la scène.
- Inversez les couleurs de la texture spéculaire dans le fragment shader de façon à ce que le bois devienne brillant et les bords ternes (les aspérités des bords deviennent alors brillantes) : **solution**.
- Créez une texture spéculaire à partir de la texture diffuse qui utilise les couleurs de l'image et voir que le résultat n'est pas très réaliste. Vous pouvez utiliser cette **texture spéculaire colorée** si vous ne voulez pas en générer une vous-même : **résultat**.
- Ajoutez ce qu'on appelle une texture d'émission qui contient des valeurs d'émission de lumière pour chaque fragment. Ces valeurs sont des couleurs qu'un objet peut émettre comme une source de lumière. De cette façon, un objet peut rayonner indépendamment des autres éclairages. Ces textures sont utilisées dans les jeux (comme les yeux d'un robot, une bande de lumière sur un conteneur). Ajouter la **texture suivante** (de creativesam) comme texture d'émission sur le conteneur et voyez si les lettres émettent de la lumière : **solution** ; **résultat**.

IV-E - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

V - Sources de lumières

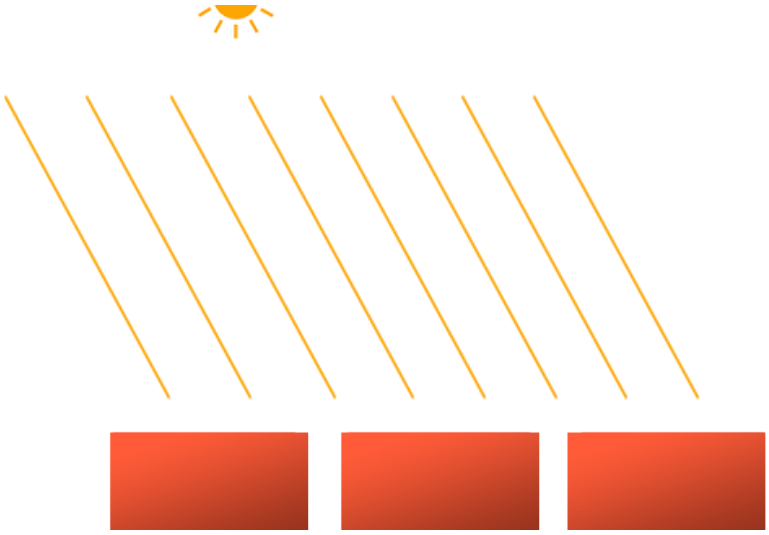
Les éclairages utilisés jusqu'ici provenaient d'une source unique concentrée en un point de l'espace. Cela donne de bons résultats, mais dans le monde réel, on trouve différents types de sources lumineuses qui ont des effets divers. Une source qui projette de la lumière sur les objets est appelée un projecteur (*light caster*). Dans ce chapitre, nous verrons différents types de projecteurs. Apprendre à simuler différentes sources de lumière est encore un outil supplémentaire pour enrichir vos scènes.

Nous verrons d'abord les sources directionnelles, puis les sources ponctuelles qui en sont une extension et finalement nous parlerons des spots lumineux. Dans le **chapitre suivant**, nous combinerons ces différents types d'éclairages dans une seule scène.

V-A - Sources directionnelles

Lorsqu'une source est très éloignée, les rayons lumineux qui proviennent de cette source sont pratiquement parallèles. Tout se passe comme si les rayons arrivaient dans la même direction, quelles que soient la position de l'objet et celle du spectateur. Quand une source est modélisée comme infiniment éloignée, on parle de source directionnelle, car les rayons lumineux arrivent tous dans la même direction, indépendamment de la position de la source.

Un bon exemple d'une telle source est le soleil, qui n'est pas infiniment loin, mais si loin que tout se passe comme si c'était le cas. Les rayons lumineux provenant du soleil sont modélisés comme étant parallèles comme on le voit sur la figure suivante :



Puisque les rayons sont parallèles, la position de l'objet par rapport à la source est sans importance, la direction de la lumière est la même pour tous les objets de la scène. Les calculs d'éclairage seront donc similaires pour chaque objet.

On modélise une source directionnelle en définissant un vecteur direction au lieu d'un vecteur de position. Les calculs dans les shaders sont les mêmes, mais on utilisera cette fois la direction de la lumière au lieu de calculer le vecteur lightDir avec la position de la source :

```
struct Light {
    // vec3 position; // inutile pour une source directionnelle
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
…
void main()
{
  vec3 lightDir = normalize(-light.direction);
  …
}
```

Noter que nous inversons le vecteur light.direction. Les calculs d'éclairage réalisés jusqu'à présent définissaient la direction de la lumière du fragment vers la source, mais les gens préfèrent généralement spécifier une lumière directionnelle avec une direction provenant de la source. Voilà pourquoi nous inversons le vecteur direction, afin qu'il pointe vers la source. N'oublions pas de normaliser ce vecteur, ce qui évite de supposer ce vecteur comme étant unitaire.

Le vecteur lightDir résultant est ensuite utilisé comme précédemment dans les calculs d'éclairage ambiant et diffus.

Pour montrer clairement qu'une source directionnelle produit le même effet sur tous les objets, nous reprenons la scène de la fin du **neuvième chapitre sur les systèmes de coordonnées**. Si vous n'avez pas étudié cette partie, nous avons défini dix positions différentes pour le même conteneur et généré une matrice de modèle par conteneur afin de définir la position de chacun d'entre eux au moyen d'une transformation de l'espace local vers l'espace monde

```
for(unsigned int i = 0; i < 10; i++)
{
    glm::mat4 model;
    model = glm::translate(model, cubePositions[i]);
    float angle = 20.0f * i;
    model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));
    lightingShader.setMat4("model", model);
    glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

N'oublions pas de définir la direction de la source (noter que nous la définissons comme provenant de la source ; on voit que la direction de la lumière pointe vers le bas) :

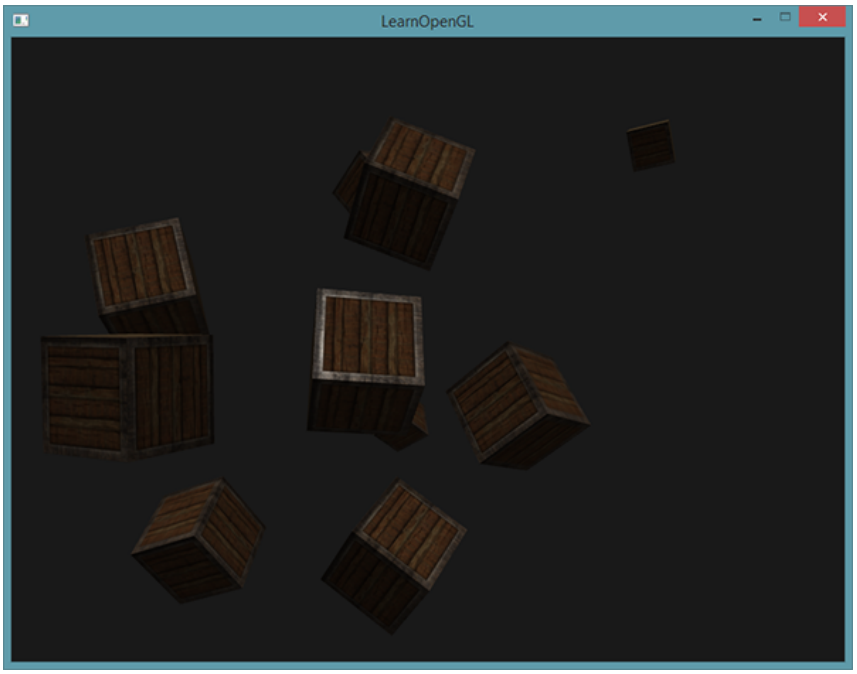
```
lightingShader.setVec3("light.direction", -0.2f, -1.0f, -0.3f);
```

Nous avons utilisé des vecteurs vec3 pour la position et la direction, mais certains préfèrent n'utiliser que des types vec4. Si l'on utilise des types vec4, il est important d'initialiser la composante w à la valeur 1.0, pour que les translations et les projections soient correctement effectuées. Cependant, pour un vecteur direction défini par un type vec4, nous ne voulons pas que les translations aient le moindre effet, et dans ce cas il faudra initialiser la composante w à 0.0.
Les vecteurs direction sont représentés comme suit : vec4(0.2f, 1.0f, 0.3f, 0.0f). Cela peut aussi servir de test pour le type de source lumineuse : si la composante w vaut 1.0, nous avons un vecteur de position pour la source, alors que si w vaut 0.0, nous avons un vecteur de direction ; on peut ajuster nos calculs en se basant sur cette valeur :

```
if(lightVector.w == 0.0) // note: attention aux erreurs de calculs pour les
 réels
    // Calculs pour une source directionnelle
else if(lightVector.w == 1.0)  // Calculs tenant compte de la position de
 la source
```

Anecdote : c'est vraiment comme cela que l'ancien OpenGL (pipeline fixe) déterminait si une source lumineuse était directionnelle ou non et ajustait les calculs d'éclairage en fonction.

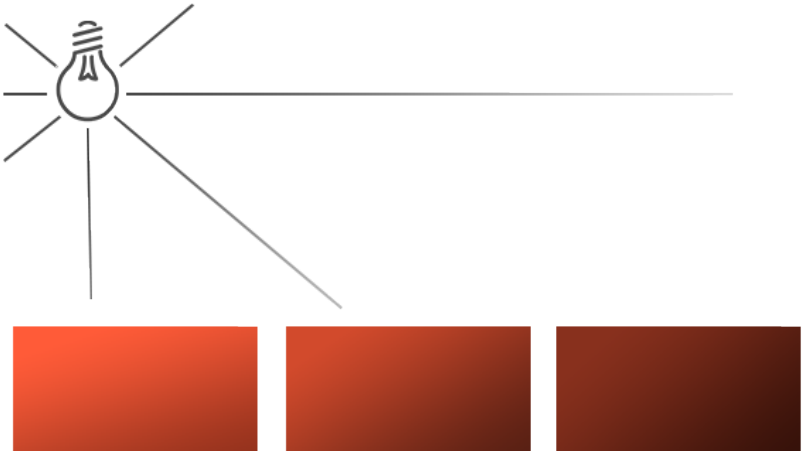
Si vous compilez l'application et observez la scène, on a l'impression que les objets sont éclairés par une source lumineuse comme un soleil. Notez que les composantes diffuses et spéculaires se comportent comme s'il y avait une source lumineuse dans le ciel. Voilà ce que ça donne :



Le code complet se trouve **ici**.

V-B - Sources ponctuelles

Les sources directionnelles sont parfaites pour la lumière qui éclaire la scène entière, mais à part cela on souhaitera aussi disposer de sources ponctuelles réparties dans la scène. Une source ponctuelle est une source de lumière placée à une position précise et qui éclaire dans toutes les directions, l'intensité des rayons lumineux s'atténuant avec la distance à la source. C'est le cas des ampoules et des bougies, qui sont des sources ponctuelles.



Dans les premiers chapitres, nous avons travaillé avec une source ponctuelle (très simple). Nous avions une source de lumière à une position donnée qui émettait de la lumière dans toutes les directions à partir de cette position. Cependant, cette source simulait des rayons sans atténuation, comme si elle était très puissante. Dans la plupart des simulations 3D, nous voudrons des sources qui n'éclairent qu'une partie de la scène, proche de la source, et non la scène entière.

Si vous ajoutez les dix conteneurs à la scène éclairée du chapitre précédent, vous verrez que le conteneur du fond est éclairé avec la même intensité que celui du premier plan ; on n'a pas défini de formule pour diminuer l'intensité de la lumière avec la distance. Nous voudrions que le conteneur du fond soit bien moins éclairé que ceux se trouvant près de la source lumineuse.

V-B-1 - Atténuation

La diminution de l'intensité de la lumière en fonction de la distance qu'un rayon lumineux parcourt se nomme atténuation. À cette fin, on pourrait simplement utiliser une fonction linéaire. Les objets éloignés seraient moins éclairés que les objets proches. Cependant, une fonction linéaire donnerait un résultat peu crédible. En réalité, les éclairages donnent une lumière assez forte quand on est près de la source, mais qui diminue assez rapidement lorsque l'on s'éloigne, puis s'atténue moins vite pour des distances plus grandes. Il nous faut donc une fonction différente pour rendre compte de l'atténuation.

Par chance, ce phénomène a déjà été modélisé. La formule suivante calcule l'atténuation en fonction de la distance entre le fragment et la source, valeur que l'on multipliera par le vecteur intensité de la lumière :

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

Dans cette formule, d est la distance entre le fragment et la source. Ensuite, on définit 3 paramètres (configurables) :

- Le paramètre constant

$$K_c$$

est conservé à la valeur 1.0, son rôle est d'assurer que le dénominateur est toujours supérieur à 1, sinon la lumière risquerait d'être plus forte que la source, ce qui n'est pas possible.

- Le terme linéaire

$$K_l * d$$

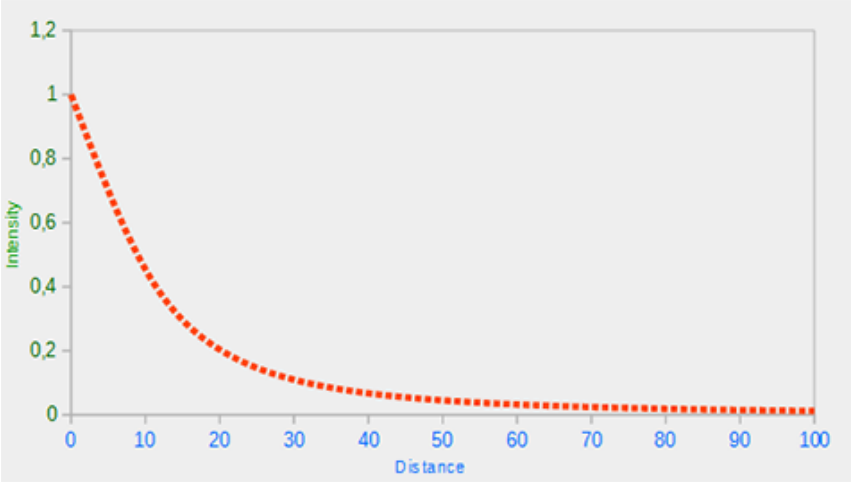
réduit la lumière de façon linéaire avec la distance.

- Le terme quadratique

$$K_q * d^2$$

réduit la lumière avec le carré de la distance. Ce terme est moins significatif que le terme linéaire quand la distance est faible, mais devient prépondérant si la distance augmente.

Du fait de ce terme quadratique, la lumière diminue linéairement pour des distances faibles, puis beaucoup plus vite pour des distances plus grandes. Le résultat est ainsi conforme à ce que l'on souhaite. La courbe suivante montre l'effet de l'atténuation sur une distance de 100 :



On voit que l'intensité est élevée pour de faibles distances, mais qu'elle diminue rapidement avec la distance pour tendre lentement vers 0. C'est bien ce que nous voulons.

V-B-2 - Choisir les bonnes valeurs

Comment choisir ces paramètres d'atténuation ? Cela dépend de nombreux facteurs : l'environnement, la distance que vous voulez couvrir avec votre lampe, le type de lumière, etc. Dans la plupart des cas, c'est une question d'expérience et d'ajustements. Le tableau suivant donne des valeurs pour simuler des éclairages réalistes qui couvrent une certaine distance. La première colonne indique la distance couverte par la lampe, les autres colonnes indiquent les paramètres. Ces valeurs constituent un bon point de départ pour la plupart des lampes, merci au wiki d'**Ogre3D** :

Distance	Constant	Linéaire	Quadratique
7	1.0	0.7	1.8
13	1.0	0.35	0.44
20	1.0	0.22	0.20
32	1.0	0.14	0.07
50	1.0	0.09	0.032
65	1.0	0.07	0.017
100	1.0	0.045	0.0075
160	1.0	0.027	0.0028
200	1.0	0.022	0.0019
325	1.0	0.014	0.0007
600	1.0	0.007	0.0002
3250	1.0	0.0014	0.000007

Le terme constant est fixé à 1.0 dans tous les cas. Le terme linéaire K_l est assez petit pour couvrir de grandes distances, et le terme quadratique est encore plus faible. Testez ces valeurs pour voir leur effet dans votre application. Dans notre cas, une distance de 32 à 100 est correcte pour la plupart des lampes.

V-B-3 - Implémenter l'atténuation

Pour implémenter l'atténuation, il nous faudra trois valeurs supplémentaires dans le fragment shader pour les trois paramètres de la formule précédente. Le mieux est de les placer dans la structure dont nous disposons. Nous calculons lightDir comme dans le chapitre précédent, et non comme au début de celui-ci pour les sources directionnelles.

```
struct Light {
    vec3 position;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float constant;
    float linear;
    float quadratic;
};
```

Nous initialisons ces valeurs dans l'application, en choisissant pour la source de couvrir une distance de 50, le tableau nous donne les valeurs des paramètres d'atténuation :

```
lightingShader.setFloat("light.constant",  1.0f);
lightingShader.setFloat("light.linear",    0.09f);
lightingShader.setFloat("light.quadratic", 0.032f);
```

Coder l'atténuation dans le fragment shader est assez direct : on calcule simplement la valeur de l'atténuation au moyen de la formule et on multiplie par les composantes ambiante, diffuse et spéculaire.

Dans cette formule, nous avons besoin de la distance à la source ; il faut se rappeler comment on calcule la longueur d'un vecteur. Il faut calculer la différence entre le vecteur position de la source et le vecteur position du fragment, et utiliser la norme de ce vecteur. On utilise la fonction GLSL length() à cet effet :

```
float distance    = length(light.position - FragPos);
float attenuation = 1.0 / (light.constant + light.linear * distance +
 light.quadratic * (distance * distance));
```

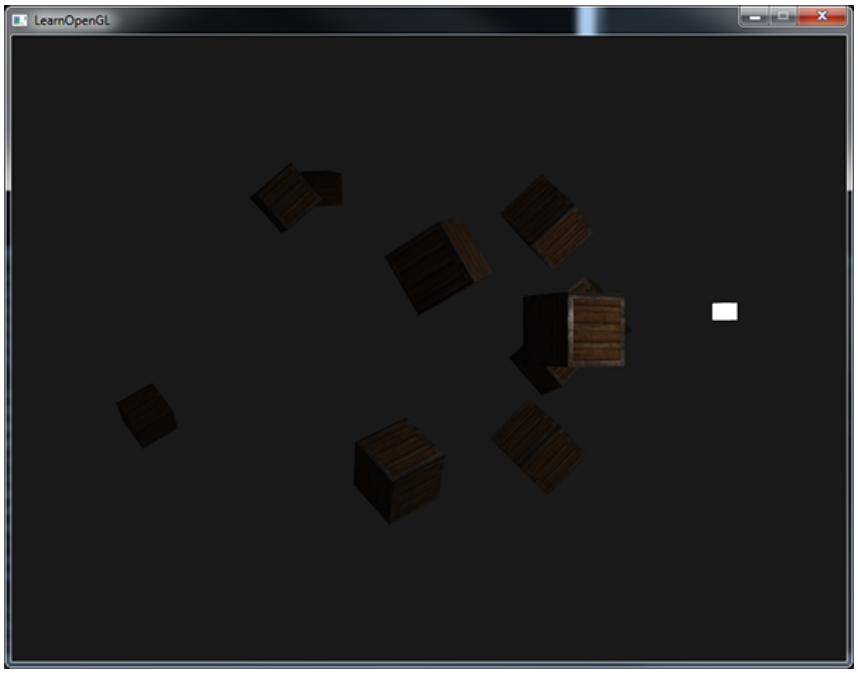
Ensuite, nous ajoutons cette valeur d'atténuation aux calculs d'éclairage.

On pourrait ne pas inclure la composante ambiante dans ce calcul, en supposant que la lumière ambiante n'est pas concernée par l'atténuation, mais si on utilise plusieurs sources,

les composantes ambiantes s'accumuleraient, ce qui n'est pas souhaitable. À vous de voir ce qui est le mieux dans votre environnement.

```
ambient  *= attenuation;
diffuse  *= attenuation;
specular *= attenuation;
```

En lançant l'application, on obtient ceci :



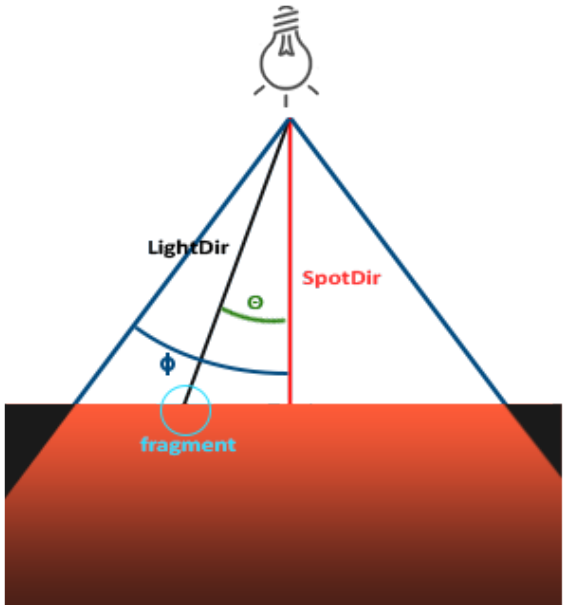
On voit que seuls les conteneurs de premier plan sont éclairés, le plus proche étant le plus lumineux. Les conteneurs du fond sont assez sombres, car ils sont trop loin de la source. Vous trouverez le code source **ici**.

Une source ponctuelle est donc une source de lumière située en un point précis et dont la lumière faiblit avec la distance. Voyons encore un autre type de source de lumière.

V-C - Spot lumineux

Un spot est une source placée à un endroit précis de la scène, mais qui n'éclaire que dans une seule direction. Seuls les objets situés dans un cône de lumière sont éclairés, le reste ne l'étant pas. Un bon exemple est un lampadaire de rue ou encore un flash d'appareil photo.

Un spot dans OpenGL est représenté par une position dans l'espace monde, une direction et un angle qui spécifie l'ouverture du cône de lumière. Pour chaque fragment, on calcule si le fragment se trouve dans le cône de lumière et dans ce cas, on tient compte de cette source pour l'éclairage du fragment. La figure suivante montre le principe de ce type de source :



- LightDir : le vecteur allant du fragment à la source.
- SpotDir : la direction dans laquelle pointe le spot.
- Phi # : l'angle d'ouverture du cône de lumière. En dehors de ce cône, point d'éclairage depuis le spot.
- Theta θ : l'angle entre le vecteur *LightDir* et le vecteur *SpotDir*. θ doit être inférieur à Φ pour que le fragment soit éclairé.

Nous calculerons le produit scalaire entre le vecteur *LightDir* et le vecteur *SpotDir* afin de trouver θ, puis de le comparer à #. Maintenant que vous avez compris ce qu'est un spot, nous allons en créer un sous forme de lampe frontale.

V-D - Lampe frontale

Une lampe frontale est un spot positionné comme le spectateur et destiné à avoir la même perspective que lui. Une lampe frontale est donc un spot classique, mais sa position et sa direction sont continuellement mises à jour en fonction de la position et de l'orientation de spectateur.

Les valeurs nécessaires pour les calculs dans le fragment shader sont le vecteur position du spot (pour calculer le vecteur direction de la lumière) le vecteur de direction du spot et l'angle d'ouverture. On peut placer ces valeurs dans la structure Light :

```
struct Light {
    vec3  position;
    vec3  direction;
    float cutOff;
    …
};
```

Nous passons les bonnes valeurs au shader :

```
lightingShader.setVec3("light.position",  camera.Position);
lightingShader.setVec3("light.direction", camera.Front);
lightingShader.setFloat("light.cutOff",   glm::cos(glm::radians(12.5f)));
```

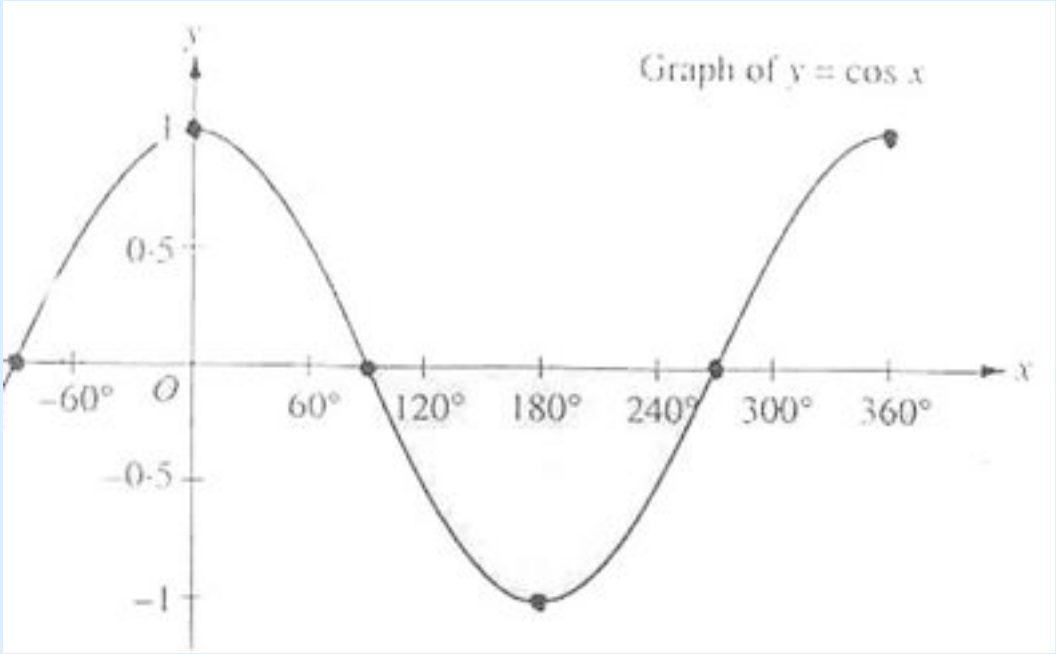
On peut remarquer que nous ne mémorisons pas l'angle du cône, mais passons directement le cosinus de cet angle au fragment shader. En effet, dans le fragment shader, nous utilisons le produit scalaire entre le vecteur LightDir et le vecteur SpotDir, nous trouvons ainsi le cosinus de l'angle formé par ces deux vecteurs, et non pas l'angle lui-même. Pour calculer l'angle, il faudrait utiliser la fonction inverse de cosinus, ce qui est une opération coûteuse. Il

est plus efficace de calculer le cosinus de l'angle Φ et de comparer les deux valeurs de cosinus, et de déterminer si le fragment est éclairé ou non :

```
float theta = dot(lightDir, normalize(-light.direction));
if(theta > light.cutOff)
{
 // calculs d'éclairement
}
else  // sinon, utiliser la lumière ambiante pour les parties de la scène non éclairées par le
 spot
  color = vec4(light.ambient * vec3(texture(material.diffuse, TexCoords)), 1.0);
```

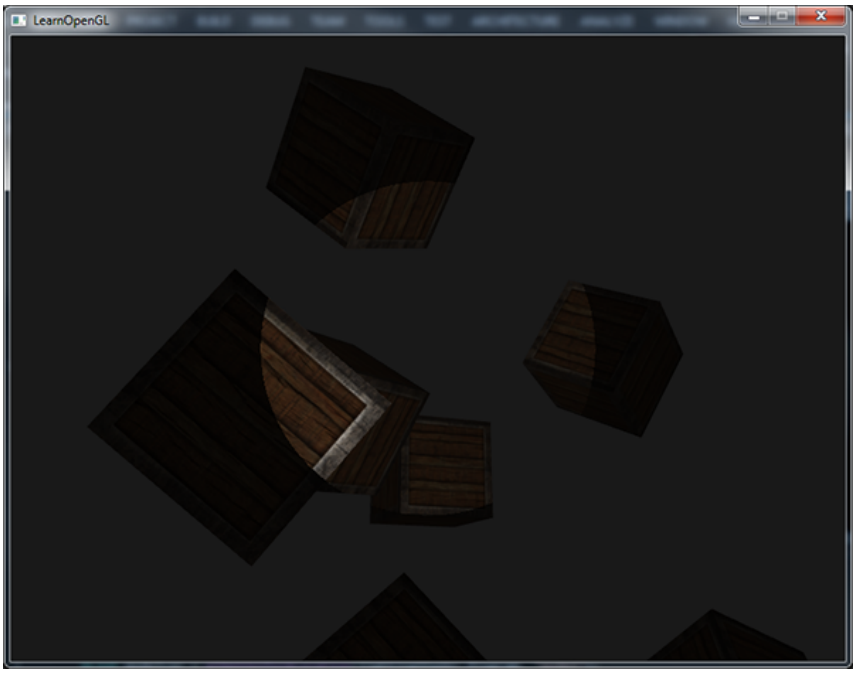
Nous calculons d'abord le produit scalaire entre le vecteur lightDir et l'opposé du vecteur direction (les deux vecteurs doivent pointer vers la source). Attention à normaliser tous ces vecteurs.

Pourquoi utiliser le signe > et non le signe < pour le test des angles ? N'oublions pas que nous utilisons le cosinus des angles, dont la courbe représentative est figurée ci-après :



Entre 0° et 90°, la fonction cosinus est décroissante, il faut inverser le signe de la comparaison.

L'application résulte en un spot lumineux qui n'éclaire que les fragments situés dans le cône de lumière :



Vous trouverez le code **ici**.

Mais le résultat semble encore un peu artificiel, car les bords du cône de lumière sont trop nets. Un fragment qui se trouve au bord du cône de lumière n'est pas du tout éclairé, au lieu de recevoir un peu de lumière. Un spot réaliste doit donner des bords où la lumière diminue en douceur.

V-E - Atténuation des bords

Pour créer un spot dont les bords donnent une lumière atténuée, nous simulerons un spot ayant un cône interne et un cône externe. Le cône interne est celui défini précédemment ; pour le cône externe, la lumière doit diminuer progressivement jusqu'à l'extérieur de ce cône.

Pour ce nouveau cône, nous définissons (par son cosinus) un angle γ donnant la mesure entre la direction du spot et le vecteur du cône extérieur. Si un fragment est dans le cône externe, mais pas dans le cône interne, nous calculerons une intensité entre `0.0` et `1.0`. Dans le cône interne, l'intensité sera égale à `1.0` et à `0.0` au-delà du cône extérieur.

On peut calculer cette intensité par la formule suivante :

$$I = \frac{\theta - \gamma}{\epsilon}$$

Où ϵ est le cosinus entre le cône interne (ϕ) et le cône externe (γ) ($\epsilon = \phi - \gamma$. Le résultat I est l'intensité du spot pour un fragment donné.

Il est un peu difficile de comprendre cette formule, voyons quels résultats on obtient pour quelques valeurs d'angles :

θ	θ en degrés	#	Φ	γ	γ en degrés	#	I	
	(cône inte				(cône exte			
0.87	30	0.91	25	0.82	35	0.91 - 0.82 = 0.09	0.87 - 0.82	- /

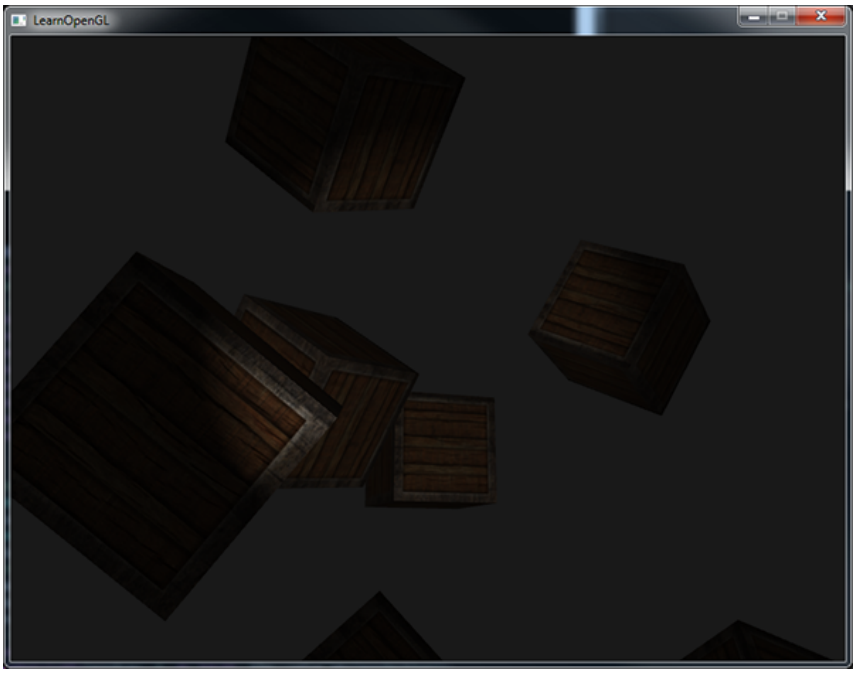
												0.09	=	
												0.56		
0.9	26	0.91	25	0.82	35	0.91	-	0.9	-					
							=	0.82	/					
								0.09	=					
												0.89		
0.97	14	0.91	25	0.82	35	0.91	-	0.97	-					
							=	0.82	/					
								0.09	=					
												1.67		
0.83	34	0.91	25	0.82	35	0.91	-	0.83	-					
							=	0.82	/					
								0.09	=					
												0.11		
0.64	50	0.91	25	0.82	35	0.91	-	0.64	-					
							=	0.82	/					
								0.09	=					
												-2.0		
0.966	15	0.9978	12.5	0.953	17.5	0.9978	-	0.966	-					
							=	0.953	/					
								0.0448	=					
												0.29		

C'est une interpolation entre les limites des deux cônes, basée sur l'angle θ. Si vous ne voyez pas bien comment cela fonctionne, pas d'inquiétude, vous pouvez utiliser la formule telle quelle, et y revenir plus tard.

Puisque nous avons une intensité qui devient négative en dehors du cône extérieur et plus grande que 1 dans le cône interne, nous allons limiter les valeurs au moyen de la fonction clamp(). Il suffit ensuite d'utiliser cette nouvelle valeur d'intensité :

```
float theta     = dot(lightDir, normalize(-light.direction));
float epsilon   = light.cutOff - light.outerCutOff;
float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
…
// nous conservons la lumière ambiante pour avoir un peu de lumière en dehors du cône du spot
diffuse  *= intensity;
specular *= intensity;
```

N'oublions pas d'ajouter le champ outerCutOff à la structure et d'affecter cette valeur uniforme dans l'application. Pour l'image suivante, nous avons choisi un angle interne de 12,5° et un angle externe de 17,5° :



Ah, c'est bien meilleur ! Jouez avec les angles internes et externes et créez un spot qui vous conviendra pour le mieux. Vous trouverez le code source **ici**.

Une lampe frontale de ce type est parfaite pour les jeux d'horreur et combinée avec des sources directionnelles et des sources ponctuelles, le résultat sera très réaliste. Dans le prochain chapitre, nous combinerons toutes ces lampes et ces astuces.

V-F - Exercices

- Essayez les différents types d'éclairages et leur fragment shader. Essayez d'inverser certains vecteurs et changez < en >. Interprétez les images obtenues.

V-G - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.

VI - Plus de lumières

Au cours des chapitres précédents, nous avons appris beaucoup sur l'éclairage dans OpenGL. Nous avons vu le modèle de Phong, les matériaux, les textures de lumière et différents types de lampes. Dans ce chapitre, nous allons combiner toutes ces connaissances en créant une scène entièrement éclairée au moyen de six sources lumineuses. On simulera la lumière du soleil avec une source directionnelle, quatre points lumineux répartis dans la scène, et nous ajouterons aussi une lampe frontale.

Pour utiliser plus d'une source de lumière dans la scène, nous allons effectuer les calculs d'éclairage dans des fonctions GLSL. En effet, le code devient vite très lourd si l'on effectue ces calculs, chaque lampe exigeant un calcul différent. Si l'on effectue ces calculs dans la fonction `main()`, le code devient inextricable.

Les fonctions en GLSL sont comme les fonctions en C. Nous avons un nom de fonction, un type pour la valeur de retour, et il faut déclarer un prototype au début du code si la fonction n'a pas été définie avant la fonction `main()`. Nous créerons une fonction par type de source de lumière : source directionnelle, point lumineux, spot.

Quand on utilise plusieurs sources dans une scène, l'approche est la suivante : on a un vecteur unique qui représente la couleur finale du fragment, et chaque source contribue à cette couleur finale. Ainsi, chaque source de la scène apportera sa contribution à la couleur finale. La structure générale du code ressemblera à cela :

```
out vec4 FragColor;
void main()
{
 // définition de la couleur finale
 vec3 output = vec3(0.0);
  // ajout de la contribution des sources directionnelles
  output += someFunctionToCalculateDirectionalLight();
  // idem pour les points lumineux
  for(int i = 0; i < nr_of_point_lights; i++)
        output += someFunctionToCalculatePointLight();
  // et enfin pour les spots ou autres sources
  output += someFunctionToCalculateSpotLight();
  FragColor = vec4(output, 1.0);
}
```

Le code réel sera légèrement différent selon les implémentations, mais la structure générale sera la même. On définit plusieurs fonctions qui calculent l'effet de chaque source et qui ajoutent la couleur résultante à la couleur finale. Si par exemple un fragment se trouve assez proche de deux sources, il sera plus clair que s'il n'est éclairé que par une seule source.

VI-A - Éclairage directionnel

Nous voulons définir une fonction dans le fragment shader qui calcule la contribution d'une source de lumière directionnelle sur le fragment : une fonction qui utilise des paramètres pour calculer et retourner la couleur de l'éclairage directionnel sur ce fragment.

Il nous faut déjà les variables que nous avions définies pour une source directionnelle. On peut placer ces variables dans une structure DirLight et la définir comme uniforme. Ces variables ont été vues au **chapitre précédent** :

c17_1_1
```
struct DirLight {
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
uniform DirLight dirLight;
```

On peut ensuite passer la variable uniforme dirLight en paramètre d'une fonction avec le prototype suivant :

c17_1_2
```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir);
```

Juste comme en C ou en C++, si nous voulons appeler une fonction (ici dans la fonction main()), la fonction doit être déclarée avant l'endroit où elle est appelée, ce qui est fait au moyen du prototype de la fonction. Nous placerons la définition de ces fonctions après la fonction main().

On voit que la fonction utilise une structure DirLight et deux vecteurs. Si vous avez suivi le tutoriel précédent, le code de cette fonction ne devrait pas vous surprendre :

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);
```

```
    // éclairage diffus
    float diff = max(dot(normal, lightDir), 0.0);
    // éclairage spéculaire
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // assemblage du résultat
    vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    return (ambient + diffuse + specular);
}
```

Nous avons simplement recopié le code du tutoriel précédent et utilisé les vecteurs passés en paramètres pour calculer la contribution de cette lumière directionnelle. Les composantes ambiante, diffuse et spéculaire sont ajoutées dans un unique vecteur couleur.

VI-B - Sources ponctuelles

Juste comme pour les sources directionnelles, nous définirons une fonction qui calcule la contribution d'une source ponctuelle sur un fragment, en tenant compte de l'atténuation. Nous définissons une structure qui comprend les données utiles pour ce type de source :

```
struct PointLight {
    vec3 position;
    float constant;
    float linear;
    float quadratic;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
#define NR_POINT_LIGHTS 4
uniform PointLight pointLights[NR_POINT_LIGHTS];
```

Comme on le voit, nous avons utilisé une directive du préprocesseur dans GLSL pour définir le nombre de sources ponctuelles de la scène. Nous utilisons ensuite la constante NR_POINT_LIGHTS pour créer un tableau de structures. Les tableaux en GLSL sont comme les tableaux en C et sont créés en utilisant deux crochets. Dès lors, nous avons quatre structures à initialiser.

On pourrait aussi définir une seule grande structure (au lieu d'une structure par type de source), contenant tous les champs nécessaires pour tous les types de sources et utiliser cette structure dans chaque fonction, en ignorant les champs non concernés. Cependant, je trouve l'approche proposée plus intuitive et plus efficace en termes de gestion mémoire, car une source ne requiert pas nécessairement tous les champs de tous les types de source.

Le prototype de la fonction pour une source ponctuelle est le suivant :

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir);
```

Cette fonction prend en paramètres les données nécessaires et retourne un vec3 qui représente la contribution de la source ponctuelle à la couleur finale du fragment. À nouveau, on utilise le code défini au chapitre précédent :

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // éclairage diffus
    float diff = max(dot(normal, lightDir), 0.0);
    // éclairage spéculaire
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
```

```
    // atténuation
    float distance    = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance +
    light.quadratic * (distance * distance));
    // assemblage du résultat
    vec3 ambient  = light.ambient  * vec3(texture(material.diffuse, TexCoords));
    vec3 diffuse  = light.diffuse  * diff * vec3(texture(material.diffuse, TexCoords));
    vec3 specular = light.specular * spec * vec3(texture(material.specular, TexCoords));
    ambient  *= attenuation;
    diffuse  *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

Encapsuler cette fonctionnalité dans une fonction comme celle-ci a l'avantage de pouvoir calculer l'éclairage pour plusieurs sources ponctuelles sans avoir besoin de dupliquer le code pour chaque source. Dans la fonction main(), on utilisera une boucle qui appellera la fonction CalcPointLight() pour chacune des sources ponctuelles.

VI-C - Tout rassembler

Maintenant que nous disposons d'une fonction pour les sources directionnelles et une autre pour les sources ponctuelles, nous allons les appeler dans la fonction main() :

```
void main()
{
    // Propriétés
    vec3 norm = normalize(Normal);
    vec3 viewDir = normalize(viewPos - FragPos);
    // phase 1 : éclairage directionnel
    vec3 result = CalcDirLight(dirLight, norm, viewDir);
    // phase 2 : lumières ponctuelles
    for(int i = 0; i < NR_POINT_LIGHTS; i++)
        result += CalcPointLight(pointLights[i], norm, FragPos, viewDir);
    // phase 3 : spot
    //result += CalcSpotLight(spotLight, norm, FragPos, viewDir);
    FragColor = vec4(result, 1.0);
}
```

Chaque type de source apporte sa contribution à la couleur finale. Si vous le souhaitez, vous pouvez aussi définir une fonction pour les spots, ce que nous laissons en exercice pour le lecteur.

Initialiser les variables uniformes ne devrait pas paraître trop nouveau, mais vous pouvez vous demander comment le faire dans ce cas, car les variables uniformes pour les sources lumineuses sont dans un tableau de structures. Nous n'avons pas encore vu ce cas de figure.

Par chance, ce n'est pas si compliqué. Pour affecter une variable uniforme qui est un tableau de structures, on la traite comme le champ d'une structure, mais, en plus, il faut utiliser l'indice du tableau pour accéder à la structure :

```
lightingShader.setFloat("pointLights[0].constant", 1.0f);
```

Ici nous pointons la première structure du tableau pointLights et le champ constant de cette structure. Cela implique qu'il faudra affecter manuellement tous les champs de chacune des quatre sources ponctuelles, soit 28 appels, ce qui est fastidieux. On pourrait essayer d'améliorer cela en utilisant une classe pour les sources ponctuelles, mais il faudrait toujours initialiser ces variables uniformes.

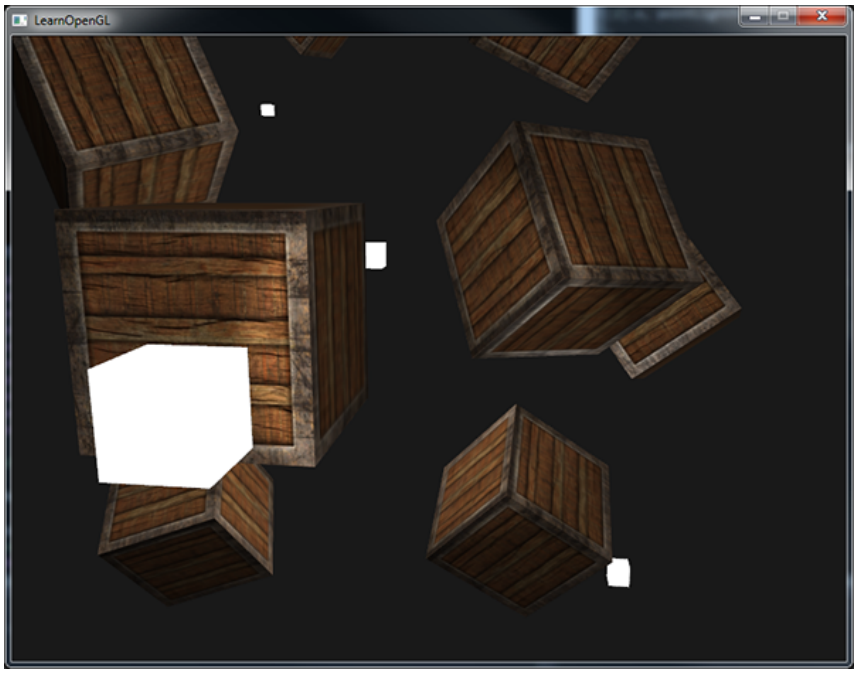
N'oublions pas que nous devons aussi définir un vecteur position pour chaque source ponctuelle, que nous répartissons dans la scène. Ces positions seront définies par un tableau de vecteurs vec3 :

```
glm::vec3 pointLightPositions[] = {
        glm::vec3( 0.7f,  0.2f,  2.0f),
        glm::vec3( 2.3f, -3.3f, -4.0f),
```

```
        glm::vec3(-4.0f,  2.0f, -12.0f),
        glm::vec3( 0.0f,  0.0f, -3.0f)
};
```

Pointez ensuite la structure du tableau pointLights correspondant à la source et affectez sa position avec les valeurs du tableau de positions. Pensez à afficher quatre cubes pour les lampes, en créant une matrice de modèle pour chacun d'entre eux, comme nous l'avons fait pour les conteneurs.

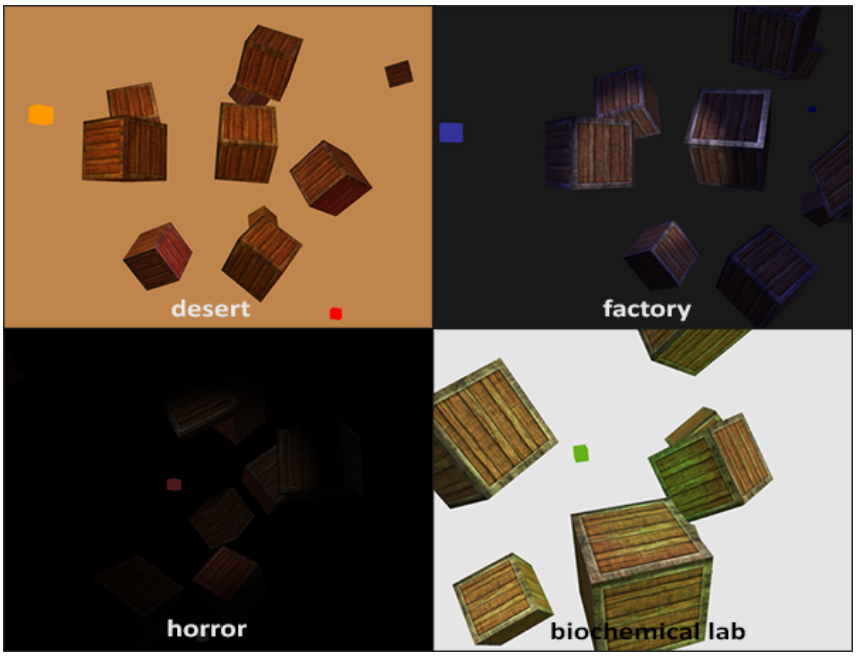
Si vous utilisez aussi une lampe frontale, vous devriez obtenir une image comme celle-ci :



On distingue une sorte de lumière globale (comme le soleil) quelque part dans le ciel, nous avons quatre lampes réparties dans la scène et une lampe frontale est visible pour le spectateur.

Vous trouverez le code de l'application **ici**.

L'image est obtenue avec toutes les sources de lumière paramétrées avec les propriétés définies dans les tutoriels précédents, mais si vous modifiez ces valeurs vous pourrez obtenir des résultats intéressants. Les artistes et les éditeurs de niveaux règlent ces valeurs dans un éditeur pour que les éclairages soient conformes à leurs souhaits. En utilisant le simple environnement éclairé que nous venons de créer, vous pouvez obtenir des résultats visuels intéressants en jouant sur les paramètres des sources lumineuses :



Nous avons aussi modifié la couleur du fond pour mieux correspondre à l'éclairage. On voit qu'en jouant sur la lumière, on obtient des atmosphères très variées.

Maintenant, vous devriez avoir une assez bonne compréhension de l'éclairage dans OpenGL. Avec ces connaissances, vous pouvez déjà créer des environnements et des atmosphères visuellement avancées.

VI-D - Exercices

- Pouvez-vous recréer de nouvelles atmosphères de la dernière scène en modifiant les attributs des sources lumineuses ? **Solution**.

VI-E - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL.**

VII - Résumé

Félicitations pour être arrivé jusqu'ici ! Je ne sais pas si vous l'avez remarqué, mais en fait, les chapitres sur l'éclairage n'ont rien apporté de nouveau sur OpenGL lui-même, excepté quelques détails comme l'accès aux tableaux de variables uniformes. Tous les tutoriels jusque-là concernaient la manipulation des shaders en utilisant des techniques et des formules pour obtenir des effets visuels réalistes. Cela vous montre à nouveau la puissance des shaders. Les shaders sont très souples et vous êtes témoin qu'avec juste quelques vecteurs 3D et quelques variables de configurations, on pouvait créer des scènes étonnantes.

Les derniers chapitres concernaient les couleurs, le modèle d'éclairage de Phong (incluant des lumières ambiantes, diffuses et spéculaires), les matériaux, les propriétés de la lumière, les textures diffuses et spéculaires, différents types d'éclairages, et comment combiner tout cela dans une seule application. Expérimentez les différentes couleurs, les matériaux, les propriétés de la lumière, et créez vos propres ambiances, soyez créatif.

Dans les chapitres suivants, nous ajouterons des objets plus complexes à la scène, qui donneront un bon rendu grâce aux éclairages que nous avons vus.

VII-A - Glossaire

- **Vecteur couleur** : un vecteur représentant la plupart des couleurs réelles au moyen d'une combinaison de rouge, vert et bleu (RVB ou RGB en anglais). La couleur d'un objet est en fait la composition des couleurs reflétées, celles qui ne sont pas absorbées.
- **Modèle de Phong** : un modèle pour approcher le comportement réel des éclairages et utilisant les composantes ambiante, diffuse et spéculaire.
- **Lumière ambiante** : approximation d'un éclairage global peu intense qui concerne chaque objet, évitant ainsi qu'un objet soit complètement noir.
- **Lumière diffuse** : éclairage d'autant plus intense qu'un fragment est face à la source de lumière. Les calculs de cette lumière nécessitent les normales.
- **Normale** : un vecteur unitaire perpendiculaire à la surface du fragment.
- **Matrice normale** : une matrice 3x3 qui est une matrice de modèle (ou de modèle-vue) sans translation. Cette matrice est modifiée (avec inversion et transposition) pour conserver les normales dans la bonne direction en cas de mise à l'échelle non uniforme de l'objet.
- **Lumière spéculaire** : donne un éclairage spéculaire, d'autant plus brillant que l'on se trouve dans la direction de réflexion d'une source. Les calculs reposent sur la direction du spectateur, celle de la source ainsi qu'un paramètre de brillance pour déterminer la netteté des reflets.
- **Éclairage de Phong** : application du modèle de Phong dans le fragment shader.
- **Éclairage de Gouraud** : application du modèle de Phong dans le vertex shader. Cela produit des effets artificiels si l'on utilise peu de sommets. Gain en temps de calcul mais dégrade la qualité visuelle.
- **Matériau** : les couleurs ambiante, diffuse et spéculaire qu'un objet reflète. Cela définit les couleurs de l'objet.
- **Propriétés de la lumière** : intensités des composantes ambiante, diffuse et spéculaire d'une lumière. Elles peuvent être de couleurs quelconques et définissent la couleur et l'intensité de chaque composante du modèle de Phong.
- **Texture diffuse** : une texture qui définit la couleur diffuse pour chaque fragment.
- **Texture spéculaire** : une texture qui définit la couleur spéculaire de chaque fragment. Permet de limiter la composante spéculaire à certaines parties de l'objet.
- **Source directionnelle** : une source de lumière qui n'émet que dans une seule direction. Modélisée comme se trouvant à une distance infinie, les rayons étant tous parallèles pour la scène entière.
- **Source ponctuelle** : une source située à un endroit précis de la scène, dont l'intensité décroît avec la distance.
- **Atténuation** : le phénomène de réduction d'intensité avec la distance, utilisé avec les sources ponctuelles et les spots.
- **Spot lumineux** : une source ponctuelle dont les rayons forment un cône de lumière dans une direction donnée.
- **Lampe frontale** : un spot positionné selon la vue du spectateur.
- **Tableau de variables uniformes** : un tableau de variables uniformes. Fonctionne comme un tableau en C, excepté qu'il ne peut pas être alloué dynamiquement.

VII-B - Remerciements

Ce tutoriel est une traduction réalisée par Jean-Michel Fray dont l'original a été écrit par Joey de Vries et qui est disponible sur le site **Learn OpenGL**.