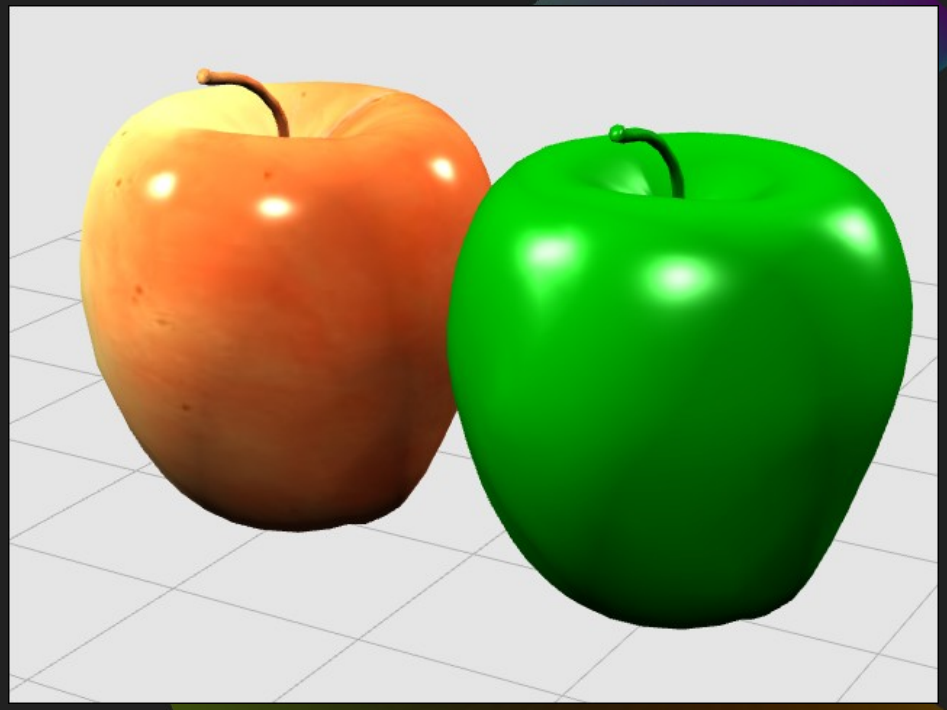
Partie 4 : Matériaux et lampes



- Matériaux
  - BRDF
  - Mise en œuvre
- Lumières
  - Types, calculs

Concepts généraux

- Un matériau définit l'apparence de la surface d'un maillage
  - Couleurs
  - Réactions à la lumière
- Pour effectuer les calculs, on a besoin de
  - Informations de surface : normales, tangentes...
  - Couleur et coordonnées des lampes
  - Matrices de transformation de la vue

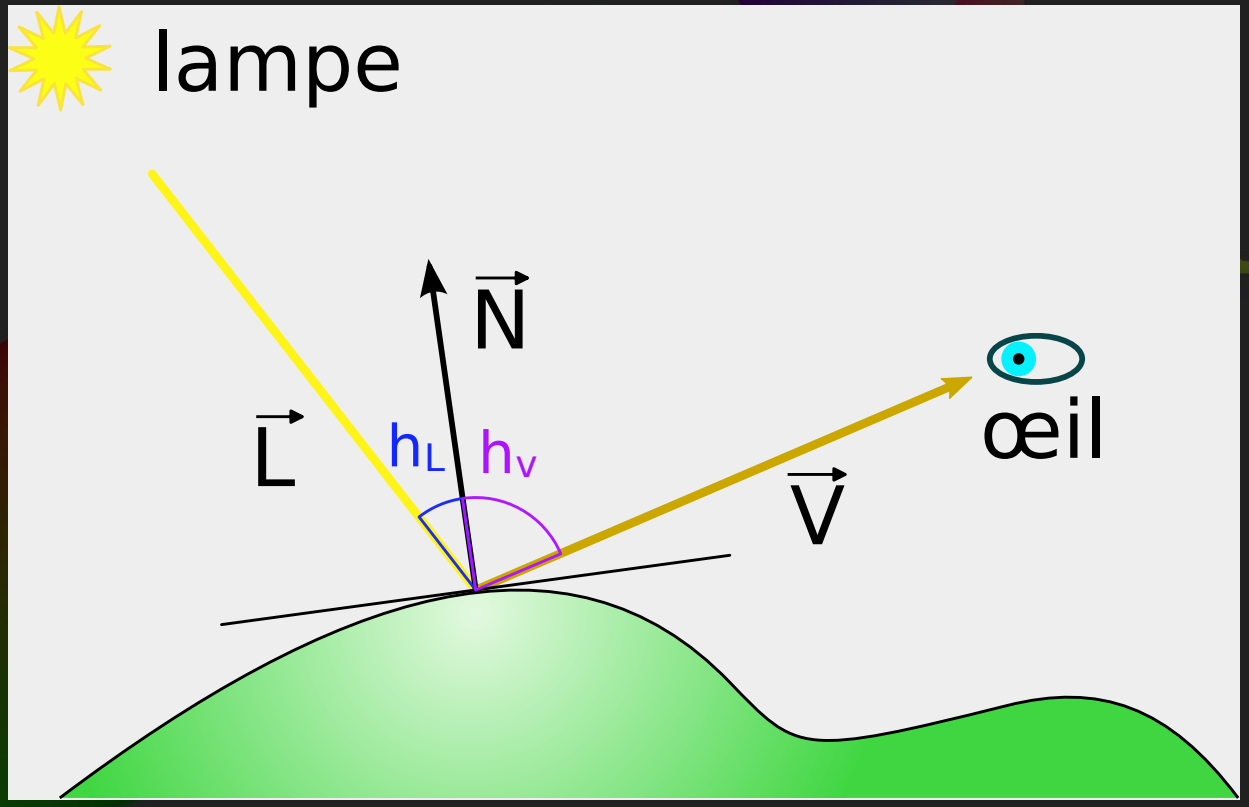
Aspects physiques optiques

- Une surface reçoit de la lumière et en renvoie une partie tout autour. L'œil voit une petite proportion de ce qui se réfléchit.
  - Composante diffuse :
    - La lumière incidente rebondit uniformément dans toutes les directions
    - Son intensité dépend de l'angle d'arrivée de la lumière initiale
  - Composante spéculaire :
    - La lumière incidente rebondit comme si la surface était un miroir
    - intensité proche de l'incidente dans un cône très fin

BRDF d'un matériau

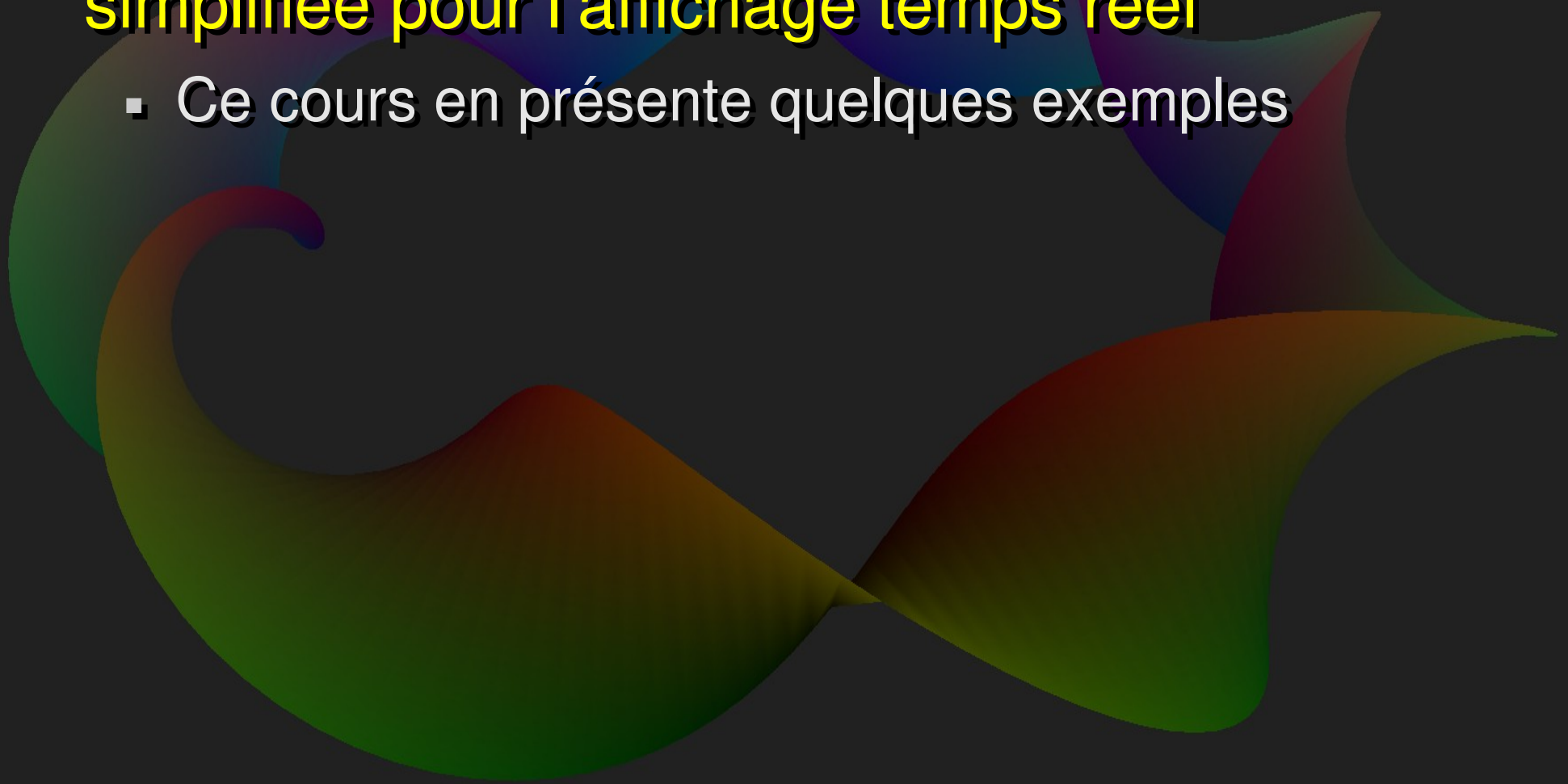
- Formellement, on définit une « Bidirectional Reflectance Distribution Function » qui modélise ce qui parvient à l'œil d'une surface éclairée

- 3 vecteurs essentiels :
  - N : normale
  - L : lampe
  - V : vue



Codage de la BRDF

- Cette fonction BRDF est programmée dans le Fragment shader, mais est généralement simplifiée pour l'affichage temps réel
  - Ce cours en présente quelques exemples



Plan de ce cours

- Présentation des sources de lumière
  - Types de lampes et calculs
- Présentation des matériaux
  - Composantes diffuse et spéculaire
- Mais avant, présentation de GLSL
  - Voir transparents de IMR2 - Synthèse d'images – GLSL.pdf

4.1 – Matériaux et maillages

- Comment intégrer la notion de matériau avec les maillages du chapitre précédent ?
  - Classe `Mesh` pour la forme des objets
    - Calcul des normales
    - Génération des VBO
    - Méthode de dessin
  - Classe `Material` pour l'apparence des objets
    - Couleur de chaque fragment
    - Gestion des lampes
    - Shaders

Classes

- Classe Mesh : elle crée les VBO nécessaires
  - Informations portées par les sommets
    - Couleurs, coordonnées locales, normales, tangentes...
- Classe Material : elle gère les shaders (VS, FS)
  - Variables attribute (`in`) : `glVertex`, `glColor`... à associer aux VBO du maillage
- Il faut connecter les deux
  - Ne pas créer de VBO(s) non utilisés par le shader

Classe Material

- Constructeur
  - Crée le shader (sources+compilation)
  - Emplacement des variables `attribute` (in vec3 ...) et `uniform`
- Méthode select(mesh, matP, matVM)
  - Active le shader, affecte les `uniform`
  - Demande les VBO nécessaires au maillage
  - Relie les VBO avec les `attribute`
- Méthode deselect()
  - Désactive VBO et shader

Classe Mesh revue

- Constructeur
  - Matériau passé en paramètre
- Méthode onDraw(matP, matVM)
  - Appelle la méthode `select` du matériau
    - Elle active le shader, affecte les uniform
    - Elle crée et active les VBO nécessaires
  - Active le VBO des indices
  - Dessine les triangles
  - Désactive le VBO des indices
  - Appelle la méthode `deselect` du matériau

Classe Mesh, suite

- Méthode `buildVBO` supprimée
  - Les VBO ne sont créés qu'à la demande de la méthode `select` du matériau
- Méthode `getVertexBufferId()`
  - Mécanisme de cache pour les VBO :
    - Si `this.m_VertexBufferId==null` alors créer le VBO des coordonnées et l'affecter à cette variable
  - Retourner `this.m_VertexBufferId`
- Méthodes `getColorBufferId()`, `getNormalBufferId()`, etc. similaires

Avantages, inconvénients

- Avantages
  - Les VBO ne sont créés que s'ils sont utiles, au dernier moment et une seule fois
  - C'est facile à comprendre
  - C'est extensible : on crée des sous-classes de `Material` et de `Mesh`
- Défauts
  - Les VBO doivent avoir la bonne largeur, celle qu'attend le shader, ex : vec3
  - Cette méthode ne permet pas d'afficher le même maillage avec des matériaux différents
    - Compliqué : il faut une autre classe entre les deux

4.2 – Sources de lumière

- On verra ce qui concerne les matériaux plus loin, mais c'est comme en lancer de rayons :
  - Calcul de la composante diffuse
  - Calcul de la composante spéculaire
- On va commencer par étudier les sources de lumière
  - PB à résoudre : calculer le vecteur L en tout point des objets

Types de lumières

- En synthèse temps réel, on peut définir :
  - Lampes ambiantes : une lumière uniforme baigne toute la scène, elle simule les éclairements indirects : rebonds multiples entre surfaces
  - Lampes ponctuelles
    - Directionnelles
    - Positionnelles, omnidirectionnelles
    - Spot
- Les sources étendues ne peuvent être qu'approximées

Lampe ambiante

- Le calcul est simple :
  - Éclairement = couleur fixe, généralement du gris sombre, ex : (0.2, 0.2, 0.2)
- Cet éclairement s'additionne aux éclairements des autres sources de lumière
  - Il simule les éclairements indirects et les éclairements globaux (ciel)
  - Il ne tient pas compte des occlusions (les objets qui sont devant, masquant partiellement la visibilité du ciel)

Lampes ponctuelles

- Ce sont des lampes qui émettent de la lumière à partir d'un point (aucune étendue)
  - Directionnelles : situées à l'infini, les rayons lumineux sont parallèles, ex : soleil
    - L'intensité lumineuse est constante quelque soit la distance du point considéré
  - Positionnelles : proches de la scène, les rayons lumineux changent de direction selon le point considéré
    - L'intensité lumineuse diminue selon le carré de l'éloignement

Lampes ponctuelles, calculs

- Le fragment shader calcule l'éclairement des lampes ponctuelles à l'aide de plusieurs vecteurs :
  - N : perpendiculaire à la surface en tout fragment
  - L : direction de la lumière
  - V : direction du regard
- Leurs calculs sont faits en plusieurs étapes :
  - Scene.onDrawFrame : préparation des matrices, positions et orientations des objets et des lampes
  - Vertex shader et Fragment shader

Rappel des transformations

- La méthode `Scene.onDrawFrame` définit les matrices :
  - `matM` : position/orientation d'un objet par rapport à la scène
  - `matV` : position/orientation de la scène par rapport à la caméra
  - `matP` : projection perspective sur écran
  - Les deux premières sont rassemblées dans `matVM` = matV * matM qui est fournie, avec matP, aux shaders via les méthodes `onDraw` des maillages et `select` des matériaux

Repère pour les calculs

- Pour calculer N, L et V, plusieurs repères pourraient être utilisés :
  - Repère local objet : `glVertex`
    - Non, car chaque objet a son propre repère, on ne peut pas y ramener les lampes
  - Repère scène : `matM * glVertex`
    - Non, car on n'a que `matVM`
  - **Repère caméra** : `matVM * glVertex` : oui !
  - Repère écran : `matP * matVM * glVertex`
    - Non, car il est distordu à cause de la perspective

=> on utilise donc le repère caméra, tous les calculs doivent y être ramenés

Calcul du vecteur N

- Chaque maillage définit le vecteur normal localement, au niveau de chaque sommet
  - Il faut le changer de repère pour l'amener dans le repère caméra

```
frgN = matN * glNormal;
```

  - La matrice normale `matN` est dérivée de `matVM` (inverse de la transposée de son coin 3x3 haut gauche)
- `frgN` est calculé par le vertex shader, il est interpolé vers le fragment shader où on écrit :

```
vec3 N = normalize(frgN);
```

Calcul du vecteur V

- Ce vecteur va du sommet ou fragment considéré vers la caméra
  - Dans son propre repère, la caméra est en (0,0,0)
  - Il faut calculer les coordonnées du point considéré dans le repère caméra :
    - Le vertex shader calcule cette variable « out » :

```
frgPosition = matVM * glVertex;
```

    - Le fragment shader n'a plus qu'à calculer :

```
vec3 V = -normalize(frgPosition.xyz);
```

    - Parfois, c'est -V qu'il faut calculer (cause fonction reflect)

Calcul du vecteur L

- Position ou direction de la lampe : vec4 $Lampe_{scène}$
  - Lampe positionnelle si w=1
  - Lampe directionnelle si w=0
- La matrice `matV` amène ces coordonnées dans le repère caméra :

  $Lampe_{caméra} = matV * Lampe_{scène}$
- Dans le fragment shader, le vecteur L vaut :
  - `L` = $Lampe_{caméra}$ − `frgPosition` si positionnelle
  - `L` = $Lampe_{caméra}$ si lampe directionnelle

Mise en œuvre

- Le constructeur de Scene définit la position ou direction de la lampe dans le repère scène

```
this.m_LightPositionScene =
    vec4.fromValues(-3.0, 3.0, 1.0, 1.0);
this.m_LightPositionCamera = vec4.create();
```

- Le changement vers le repère caméra est fait dans Scene.onDrawFrame

```
vec4.transformMat4(
    this.m_LightPositionCamera,
    this.m_LightPositionScene,
    this.m_MatV);
```

Mise en œuvre, suite

- Dans le fragment shader de chaque matériau, une variable `uniform vec4 LightPosition` reçoit les coordonnées dans le repère caméra
  - Lampe directionnelle (`LightPosition.w = 0`) :

    ```
    L = normalize(LightPosition.xyz);
    ```

  - Lampe positionnelle (`LightPosition.w = 1`) :

    ```
    L = normalize(LightPosition.xyz – frgPosition);
    ```

    avec `frgPosition = matV * glVertex` calculée par le vertex shader et aussi utilisée pour calculer V
- On peut unifier en utilisant `LightPosition.w`

  ```
  L = normalize(LP.xyz – LP.w*frgPosition);
  ```

Utilisation des vecteurs N, L et V

- Ces trois vecteurs sont cruciaux pour les calculs suivants, par exemple :
  - Éclairement diffus de Lambert :

```
gl_FragColor = Kd * Kl * dot(N, L);
```

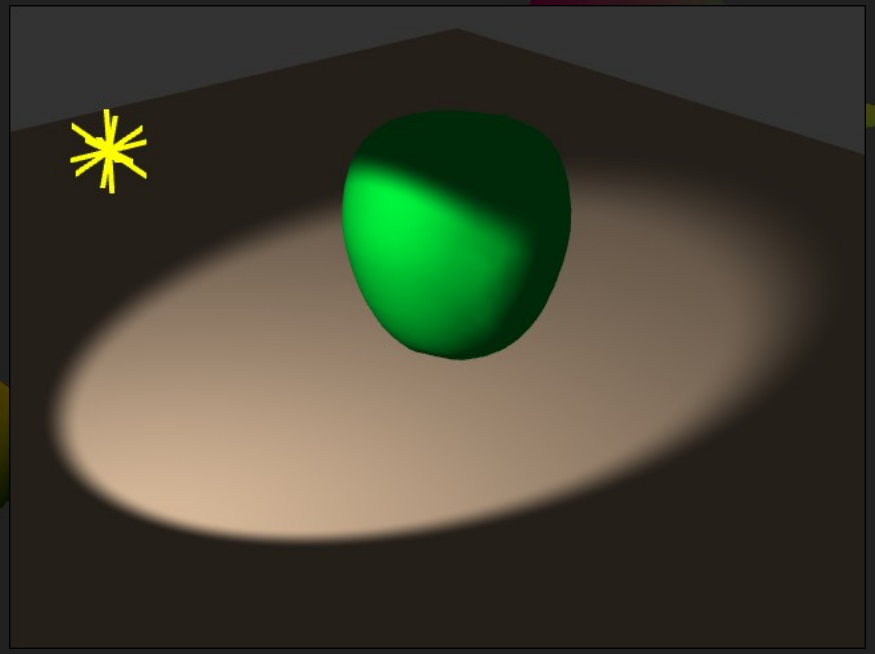
  - Éclairement spéculaire de Blinn :

```
vec3 H = normalize(V + L);
gl_FragColor += Ks * Kl * pow(dot(N, H), ns);
```

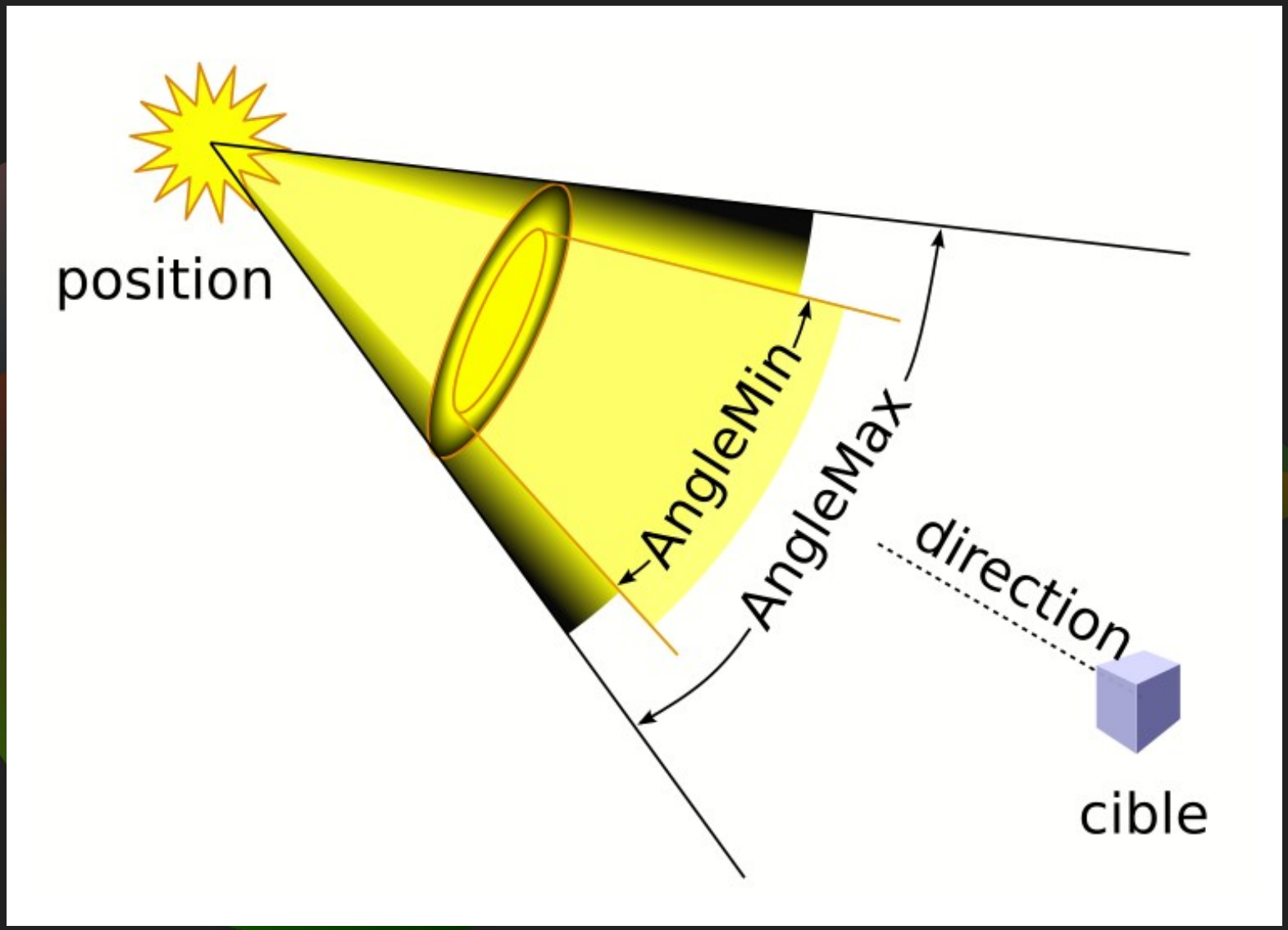
- Voir plus loin pour d'autres exemples

Lampe spot

- Une telle lampe émet de la lumière dans un cône. Elle possède :
  - Une position comme une lampe positionnelle
  - Une direction qui doit être transformée par `matV` également
  - Deux angles (option) :
    - plein éclairement :
      - anglemin
    - occultation totale :
      - anglemax



Lampe Spot, caractéristiques



Calculs de la lampe Spot

- Classe Scene
  - Constructeur
    - Définit couleur, position, direction et angles de la lampe
  - `OnDrawFrame`
    - Transforme position et direction, transmet le tout aux matériaux (via les objets)
- Calcul de visibilité dans les shaders
  - Le fragment shader détermine dans quelle mesure le fragment est dans le cône d'éclairage par :

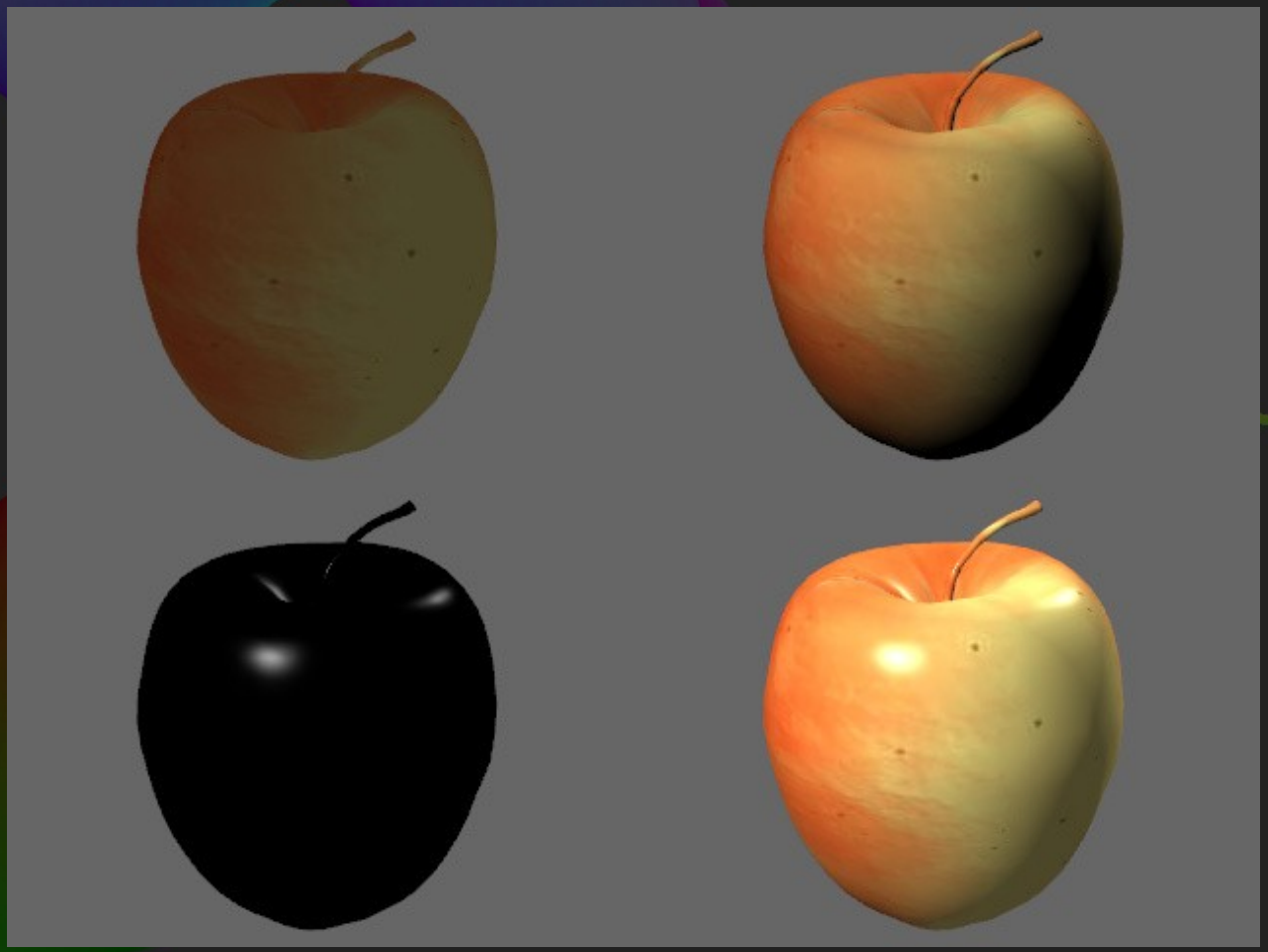
```
smoothstep( cos(anglemax), cos(anglemin),
            dot(-L, direction) )
```

À réfléchir

- Il serait pertinent de définir une hiérarchie de classes pour représenter les lampes
  - Difficulté : chaque lampe définit des instructions GLSL spécifiques ainsi que des variables uniform pour ses calculs, cela doit être placé dans chaque matériau (=> code source reconfigurable)
- Une autre méthode de dessin des éclairements a été inventée, beaucoup plus rapide, le dessin différé (*Deferred Shading*) qu'on ne pourra pas étudier faute de temps

4.3 – Composantes d'éclairement

- L'éclairement est la somme de plusieurs contributions
  - Ambiante
  - Diffuse
  - Spéculaire

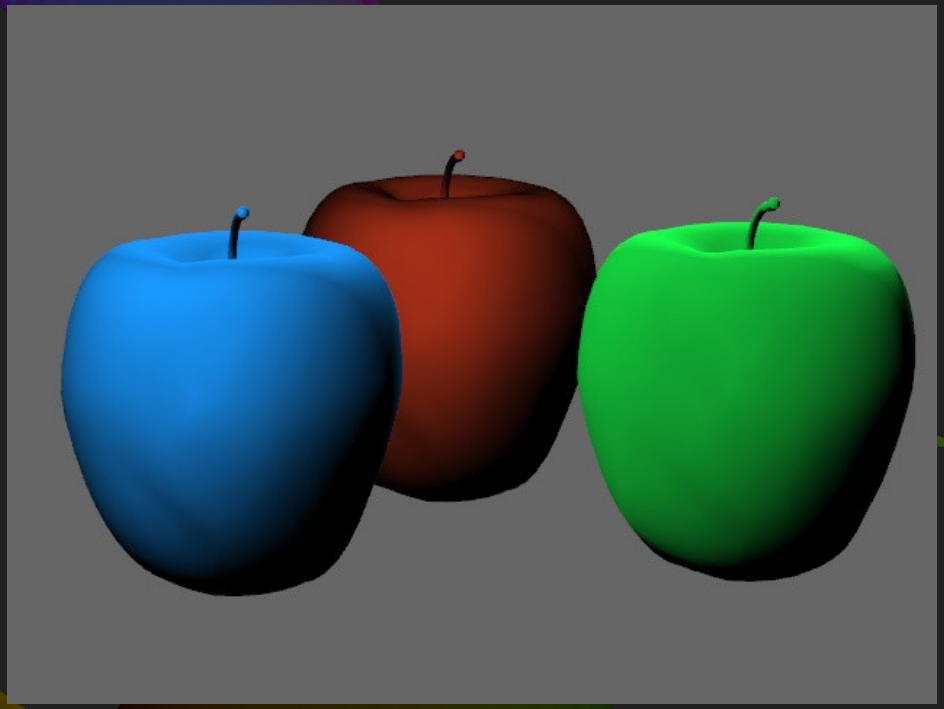


Composantes d'éclairement

- Chaque contribution A, D et S dépend des caractéristiques du matériau et de la lampe
- Elles sont chacune le produit de :
  - La couleur de la lampe (= intensité)
  - La couleur du matériau (constante ou texture)
  - Un coefficient 0..1 donnant l'effet de la lampe sur le matériau
    - Par exemple, Lambert : cos(angle(N, L)) = N.L
    - Tout est dans le calcul de ce coefficient

Éclairements diffus

- Il en existe plusieurs modélisations
  - Lambert (bleue)
  - Oren-Nayar (verte)
  - Minnaert (rouge)



Modèle de Lambert

- Pour des matériaux parfaitement poncés, ré-émettant la lumière dans toutes les directions
- Loi en « cosinus »

```
float dotNL = clamp(dot(N, L), 0.0, 1.0);
float D = dotNL;
```

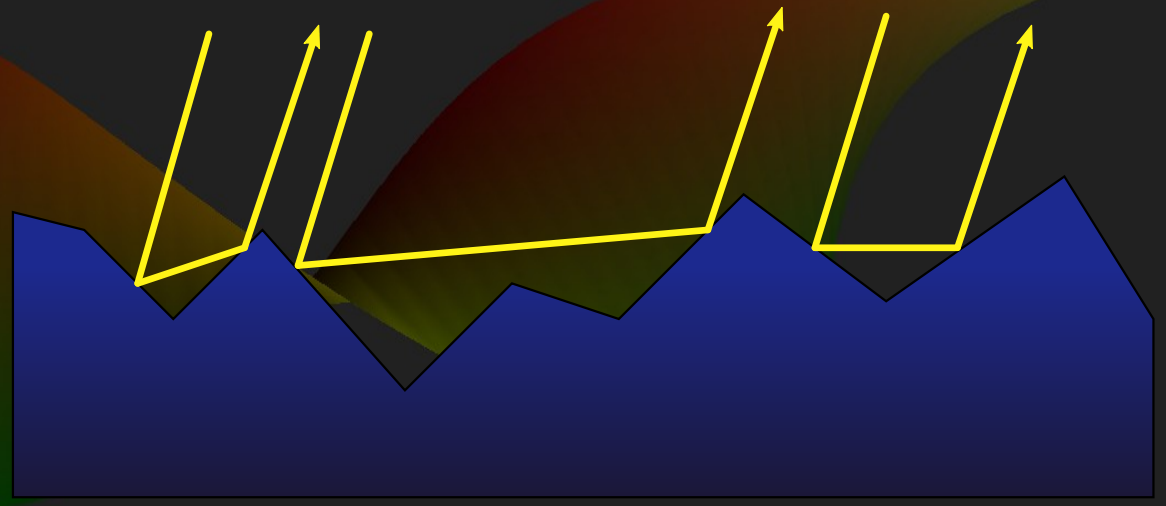
- Le calcul complet est :

```
gl_FragColor = LightColor * Kd * D;
```

- On peut aussi diviser par le carré de la distance de la lampe si elle est positionnelle

Modèle de Minnaert

- Pour des matériaux rugueux à petite échelle qui réfléchissent la lumière mais avec des auto-occultations et inter-réflexions comme les catadioptres (sols formés de cristaux cassés)



Modèle de Minnaert, calcul

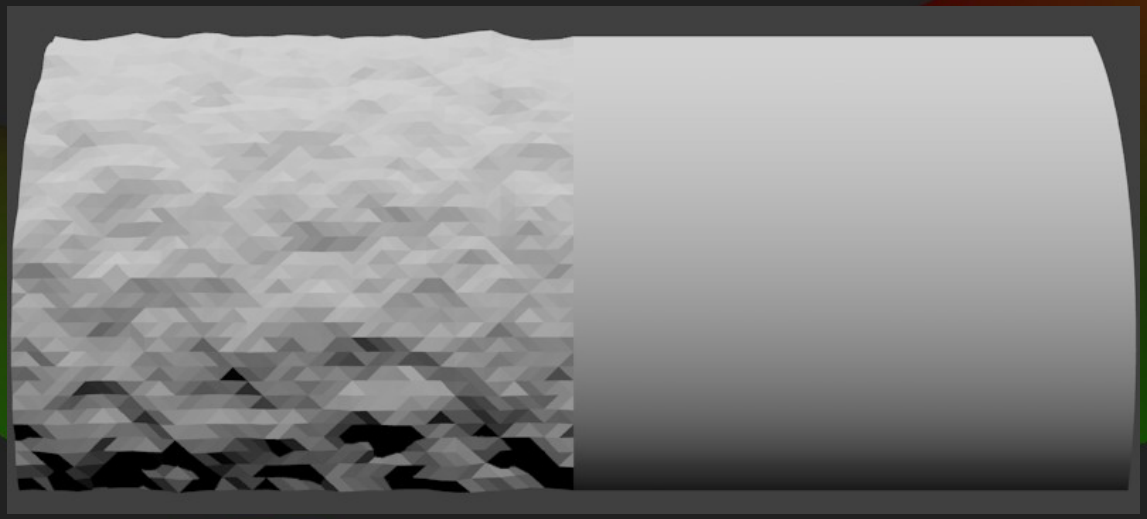
- Calcul du coefficient diffus D qui dépend d'un paramètre m caractérisant la surface :

```
float dotNV = clamp(dot(N,V), 0.0, 1.0);
float dotNL = clamp(dot(N,L), 0.0, 1.0);
float D = pow(dotNL*dotNV, m);
```

- Comme précédemment, ce coefficient D est multiplié par la couleur diffuse Kd et par la couleur de la lampe et éventuellement divisé par le carré de la distance à la lampe

Modèle de Oren-Nayar

- Il s'applique à des surfaces mates comme le plâtre, la terre cuite
  - L'état de surface microscopique (à gauche) n'est pas Lambertien (comme à droite)
  - La taille et l'inclinaison des facettes microscopiques interviennent dans le résultat



Modèle de Oren-Nayar, calcul

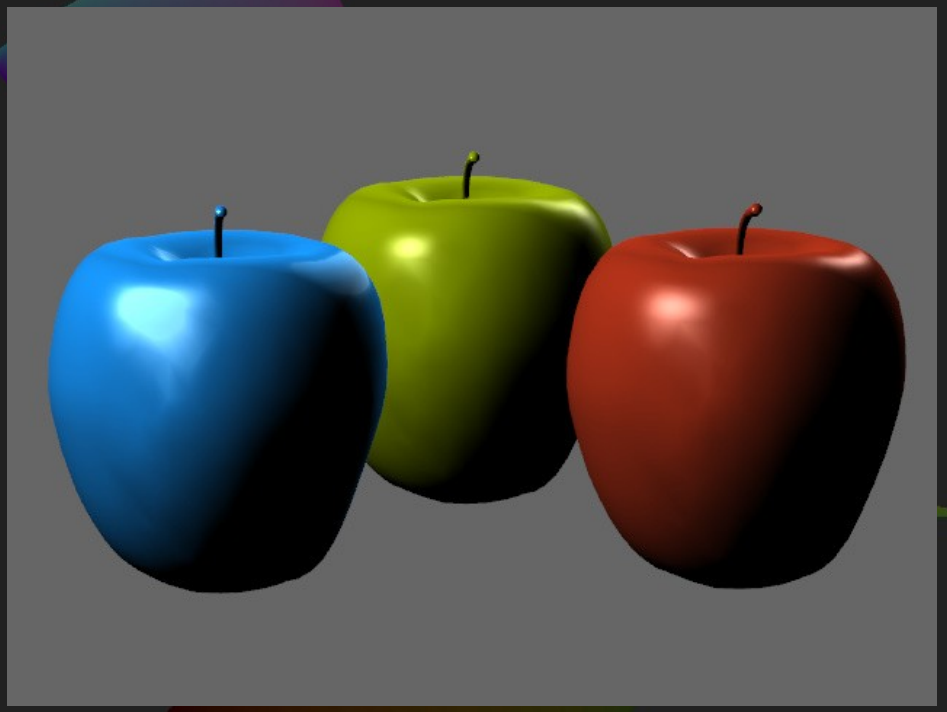
- Voici les étapes, d'abord quelques calculs trigonométriques :
  - ```
    float dotNL   = dot(N, L);
    float angleNL = acos(dotNL);
    ```
  - ```
    float dotNV   = dot(N,V);
    float angleNV = acos(dotNV);
    ```
- Puis vectoriels dans un repère TBN ad-hoc :
  - ```
    vec3 Ltb = normalize(L – N*dotNL);
    vec3 Vtb = normalize(V – N*dotNV);
    ```
    Voir cours 5 pour la définition de ce repère
    ```

Modèle de Oren-Nayar, calcul

- Suite et fin, sigma2 est un paramètre 0..1 caractérisant la rugosité de la surface
  - ```
    const float a =
        1.0 - 0.5 * sigma2 / (sigma2 + 0.57);
    const float b =
        0.45 * sigma2/(sigma2 + 0.09);
    ```
  - ```
    float alpha = max(angleNV, angleNL);
    float beta  = min(angleNV, angleNL);
    float c = sin(alpha) * tan(beta);
    float gamma = max(0.0, dot(Vtb, Ltb));
    ```
  - ```
    float D =
      clamp(dotNL,0.0,1.0) * (a + b*gamma*c);
    ```
    ```

Éclairements spéculaires

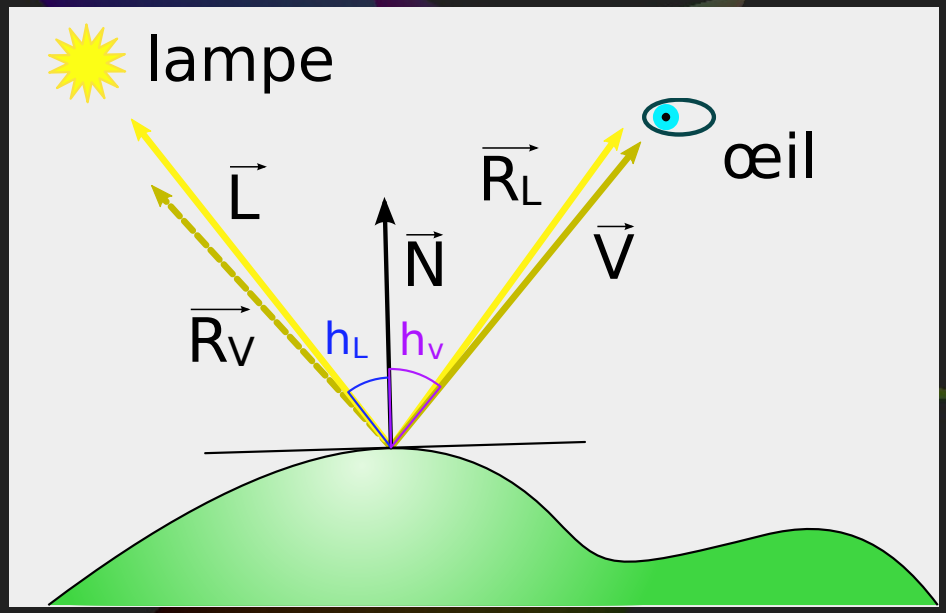
- Il existe plusieurs modélisations également :
  - Phong (verte)
  - Blinn-Phong (rouge)
  - Cook-Torrance (bleue)



- Le reflet apparaît quand V est à peu près aligné avec le miroir de L par rapport à N

Modèle de Phong

- Il faut calculer le vecteur R_V, miroir de V par rapport à N et le comparer avec L
  - R_V aligné avec L $\Leftrightarrow$ $R_V.L = 1$
  - $S = (R_V.L)^{ns} = (R_L.V)^{ns}$
  - *ns* caractérise le poli du matériau



- On calcule R_V plutôt que R_L car il ne change pas selon la lampe

Modèle de Phong, calcul

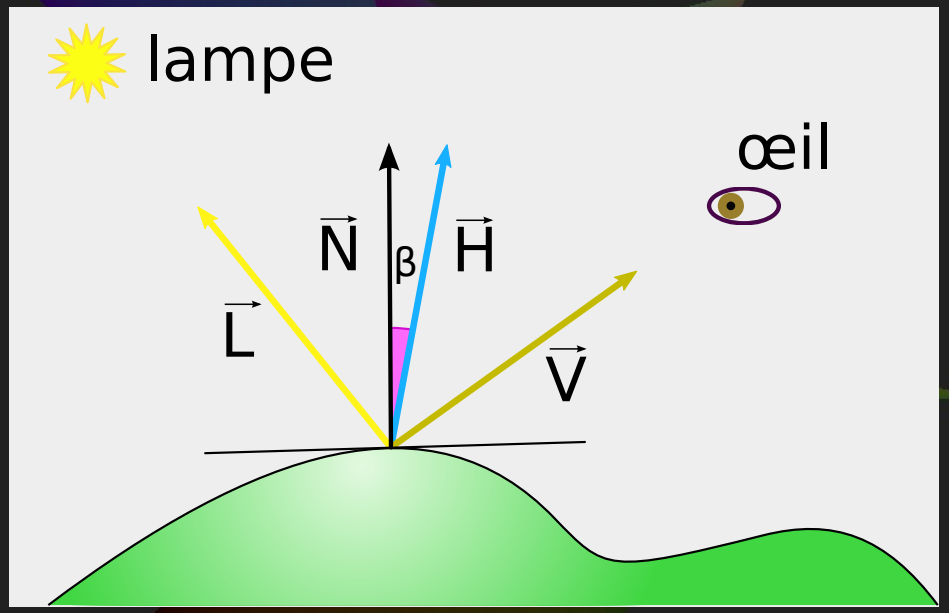
- Le fragment shader doit calculer :

```
vec3 mV = normalize(frgPosition.xyz);
vec3 Rv = reflect(mV, N);
float dotRvL = clamp(dot(Rv,L), 0.0,1.0);
float S = pow(dotRvL, ns);
```

- On travaille avec -V (noté mV) car la fonction `reflect` le demande ainsi
- Comme pour D, S est multiplié par la couleur Ks du matériau, celle de la lampe et divisé par le carré de la distance à la lampe

Modèle de Blinn-Phong

- C'est une simplification basée sur le vecteur H = milieu entre V et L
  - Quand H est aligné avec N (β nul) alors V est dans la direction du reflet de L
  - β quasi nul ⇔ N.H ≃ 1
  - On calcule S = $(N.H)^{ns}$



Modèle de Blinn-Phong, calcul

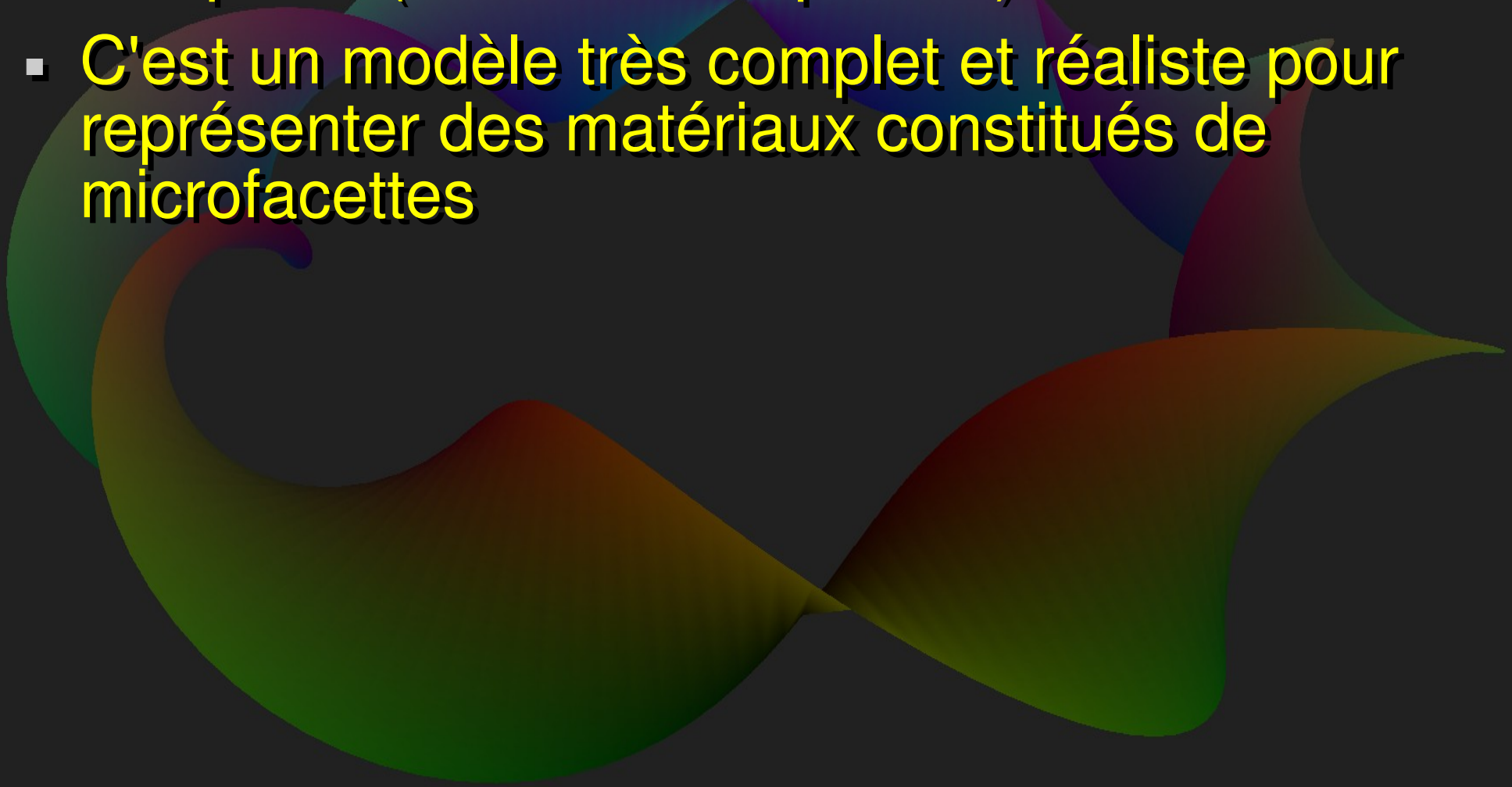
- Voici les calculs :

```
vec3 H = normalize(L + V);
float dotNH = clamp(dot(N,H), 0.0, 1.0);
float S = pow(dotNH, ns);
```

- … la simplicité et le réalisme :

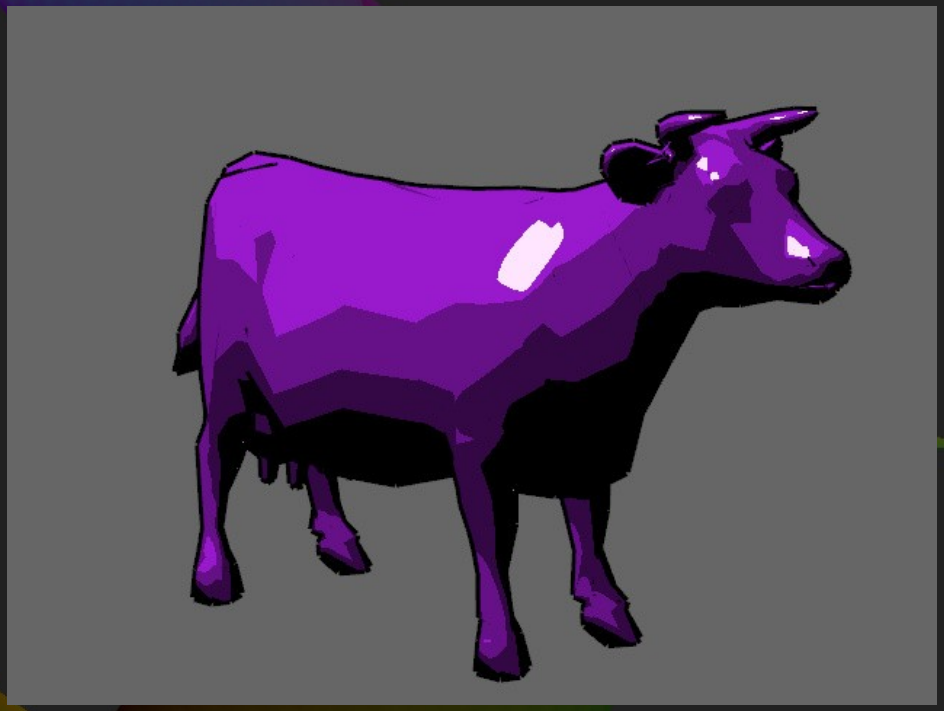
Modèle de Cook-Torrance

- Il ne sera pas abordé ici, il est un peu trop complexe (voir le livre OpenGL)
- C'est un modèle très complet et réaliste pour représenter des matériaux constitués de microfacettes



Éclairements non réalistes

- On peut aussi faire des fantaisies
  - Ici, les composantes D et S sont seuillées afin de limiter les valeurs différentes



- Par exemple :
  - `D = 0.4*step(0.5, D) + 0.2*step(0.7, D);`
  - `S = step(0.4, S);`