

# TP - Préparation à la soutenance

[malek.bengougam@gmail.com](mailto:malek.bengougam@gmail.com)

## Partie 1 – Matrices monde

La première transformation que l'on rencontre usuellement consiste à placer un objet dans la scène.

La « World Matrix », appelée aussi « Local-To-World » ou « Object-To-World », contient la concaténation des différentes transformations linéaires (scale, rotation) et affine (translation).

Jusqu'à présent nous avons séparés ces différentes transformations en matrices distinctes, mais pour des raisons de performance et de simplification du code il est habituel de réduire ces transformations à une seule matrice envoyée au shader.

Cela permet ainsi de réduire le nombre de matrices transmises au GPU (d'autant que la mémoire pour les constantes uniform-s est relativement limitée) mais également d'augmenter les performances en réduisant le nombre de multiplications matricielles qui s'effectuent à chaque invocation du Vertex Shader.

**Exercice 1.1 :** implémenter en C ou C++ une fonction multipliant deux matrices et qui renvoie le résultat.

A titre de rappels, les matrices sont des matrices homogènes 4D stockées en mémoire en colonne d'abord (première colonne, puis seconde colonne, etc...).

L'ordre des transformations se lit de la droite vers la gauche. En infographie, pour la "World Matrix" on concatène les matrices suivantes (dans cet ordre) :

**Eq 1.1:**  $\text{World Matrix} = \text{Translation Matrix} * \text{Rotation Matrix} * \text{Scale Matrix}$

**Exercice 1.2 :** remplacer vos matrices de transformation locale par une unique transformation locale-vers-monde.

***Rappel important:*** TOUT point ou vecteur devant être transformé du repère local de l'objet vers le repère de la scène DOIT être transformé par la World Matrix

Ceci vaut bien évidemment pour les positions des Vertex, mais également pour les normales. Comme vu brièvement lors du TP précédent, les normales doivent être transformées par la transposée de l'inverse de la World Matrix c'est-à-dire

**Eq 1.2:**  $\text{Normal Matrix} = (\text{World Matrix}^{-1})^T$ , en GLSL:  $\text{NormalMatrix} = \text{transpose}(\text{inverse}(\text{WorldMatrix}))$ ;

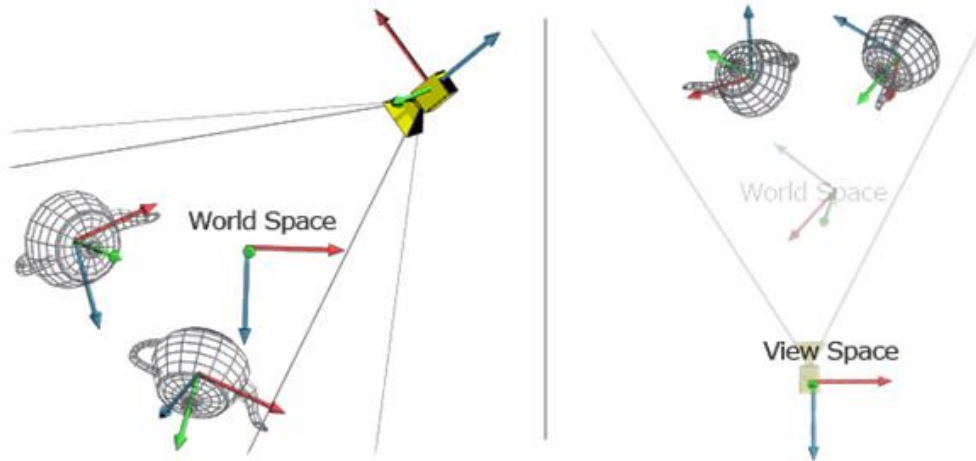
Il faut également noter que la translation n'a pas de sens pour une normale, c'est pour cela qu'une normale se représente en coordonnées homogènes (4D) avec une composante **w** à **zéro**. Mais le plus simple est ici de considérer NormalMatrix comme une **mat3**, donc de convertir WorldMatrix en **mat3**.

## Partie 2 – Matrice vue

La caméra est également un objet de la scène, et dispose donc d'une matrice World Matrix.

Pour simuler le fait que les objets sont vus depuis le point de vue de la caméra il faut également appliquer une transformation supplémentaire à l'ensemble des objets de la scène.

Cette transformation doit transformer du repère de la scène (world space) vers le repère de la caméra (aussi appelé "view space", "eye space" ou "camera space").



En fait, le repère de la caméra n'est rien d'autre que le repère local de la caméra. A savoir que dans ce repère, la caméra se trouve toujours située à l'origine.

( Pour une illustration voir cf. <https://webglfundamentals.org/webgl/lessons/webgl-3d-camera.html> )

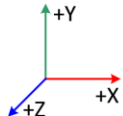
Pour simuler une caméra il suffit de calculer la transformation (inverse) Monde-Vers-Objet, de la World Matrix de la caméra.

Cela nécessite de faire appel (et programmer) une fonction inverse en C++ mais on peut faire mieux ici.

Une caméra ne subit pas de scale pas plus que de déformations de ses axes. C'est donc une transformation orthogonale. Ici,  $WorldMatrix_{cam}$  la transformation Locale-vers-Monde de la caméra :

$$WorldMatrix_{cam} = TranslationMatrix_{cam} * RotationMatrix_{cam}$$

$$\begin{bmatrix} right_x & up_x & forward_x & position_x \\ right_y & up_y & forward_y & position_y \\ right_z & up_z & forward_z & position_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Left (X) Axis

Up (Y) Axis

Forward (Z) Axis

Translation

$$\begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix}$$

Visuellement on a une matrice homogène qui mélange changement de base dans la sous-matrice 3x3 - chaque colonne représente l'orientation respectives des axes de la base- et une translation.

Comme on a seulement des rotations et des translations, on peut très bien se contenter d'inverser ces matrices pour calculer  $\text{WorldMatrix}_{\text{cam}}^{-1}$  l'inverse de  $\text{WorldMatrix}_{\text{cam}}$ :

$$\begin{aligned}\text{WorldMatrix}_{\text{cam}}^{-1} &= (\text{TranslationMatrix}_{\text{cam}} * \text{RotationMatrix}_{\text{cam}})^{-1} \\ &= \text{RotationMatrix}_{\text{cam}}^{-1} * \text{TranslationMatrix}_{\text{cam}}^{-1}\end{aligned}$$

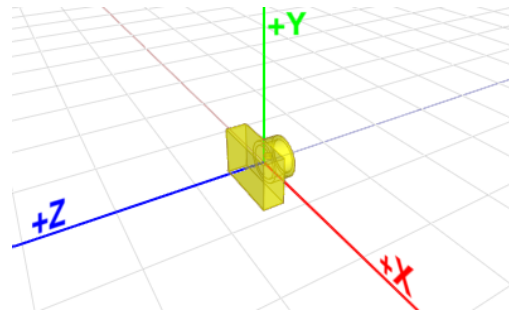
Une autre propriétés utiles, l'inverse d'une transformation linéaire orthogonale est une transposée. Et l'inverse d'une translation est simplement la translation opposée  $T^{-1} = -T$ . On optimise alors

$$\text{ViewMatrix} = \text{WorldMatrix}_{\text{cam}}^{-1} = \text{RotationMatrix}_{\text{cam}}^T * \text{TranslationMatrix}_{\text{cam}}^{-1}$$

$$\text{View Matrix: } \mathbf{M}_{\text{view}} = \mathbf{R} \mathbf{T} = \begin{bmatrix} x_x^c & x_y^c & x_z^c & 0 \\ y_x^c & y_y^c & y_z^c & 0 \\ z_x^c & z_y^c & z_z^c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -e_x \\ 0 & 1 & 0 & -e_y \\ 0 & 0 & 1 & -e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_x^c & x_y^c & x_z^c & -e_x \cdot x_c \\ y_x^c & y_y^c & y_z^c & -e_y \cdot y_c \\ z_x^c & z_y^c & z_z^c & -e_z \cdot z_c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Important: remarquez que les éléments de la dernière colonne sont issues du produit scalaire entre le vecteur  $-e$  (-translation) et chacune des colonnes de la matrice (3x3) de rotation de la caméra R qui forment le repère de la caméra.

$$\begin{pmatrix} R^T & -R^T t \\ 0^T & 1 \end{pmatrix}$$



### Fonction View Matrix "LookAt"

Plutôt que de raisonner en termes de compositions de matrices il est souvent plus facile d'utiliser des paramètres comme la position de la caméra et la cible (position) de la caméra.

On ajoute généralement un troisième paramètre référentiel "up" qui va nous permettre de former une base orthonormale.

Note: dans la plupart des cas la valeur du vecteur « up » correspond au « up » du monde soit [0, 1, 0]

**ATTENTION !** Ne confondez pas notre fonction `LookAt()` OpenGL et d'autres fonctions telles

`Transform.LookAt()` ou `Matrix4x4.LookAt()` de Unity.

Elles prennent les mêmes paramètres mais cette dernière fonction génère une matrice Locale-vers-Monde qui permet de positionner un objet (par exemple une caméra) par rapport à un autre (la cible de la caméra) et non une matrice Vue qui est en fait l'inverse de Locale-vers-Monde.

**Exercice 2.1:** codez la fonction `LookAt(vec3 position, vec3 target, vec3 up)` et passez la matrice au Vertex Shader comme View Matrix.

Pour générer la “View Matrix” il nous faut les éléments suivants:

1. Vecteur **“forward”** de la nouvelle orientation. C’est le vecteur  $-(\text{target} - \text{position})$  **normalisé**.

La base est exprimée en repère main-droite, le vecteur “forward” pointe ici HORS de l’écran. C’est pourquoi il y’a un signe moins (-).

2. Vecteur **“right”**. Il est formé par le **produit vectoriel** entre le vecteur “forward” et “up”.

Le produit vectoriel suit toujours la règle de la main-droite, à savoir, intuitivement, alignez le premier vecteur sur l’index de votre main droite, le second vecteur sur votre majeur, la direction du résultat est indiquée par votre pouce.

Ici, il faut donc que le premier vecteur soit “up” et le second “forward”.

3. Vecteur **“up”** corrigé. Il est formé par le **produit vectoriel** entre le vecteur “right” et “forward”.

Pareil ici, mais par commodité on va aligner le premier vecteur sur le pouce de votre main droite, le second vecteur sur votre index, la direction du résultat est indiquée par votre majeur.

Ici, faut donc que le premier vecteur soit “forward” et le second “right”.

4. Calculez **trois produits scalaires**: un entre la “position” et le vecteur “right”, un entre la “position” et le vecteur “up”, et un entre la “position” et le vecteur “forward”.

Cette opération correspond à une projection de la position (exprimée en repère monde) dans le repère local de la caméra afin de former une translation inverse. N’oubliez pas la négation des composants !

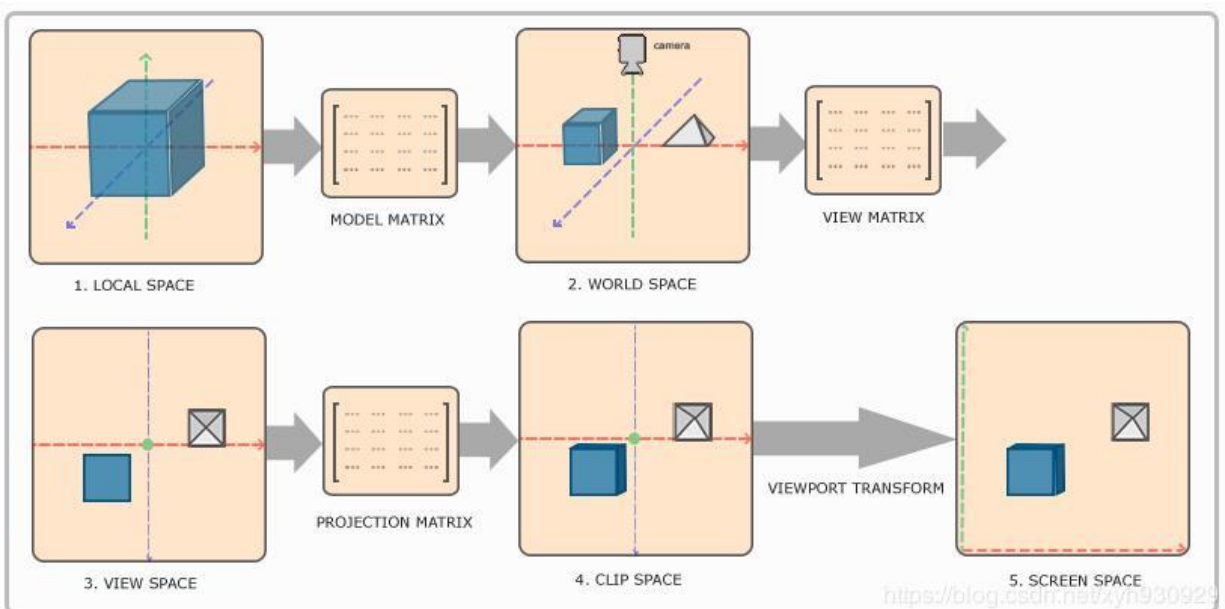
5. Assemblez le tout dans une matrice.

Les 3 premières colonnes contiennent la transposée des 3 vecteurs. On peut voir que la première colonne contient toutes les composantes ‘x’ des vecteurs, la seconde colonne toutes les composantes ‘y’ et la troisième colonne toutes les composantes ‘z’.

La quatrième colonne contient les résultats de la 4ième étape.

$$V = \begin{bmatrix} X_c \cdot x & X_c \cdot y & X_c \cdot z & -\text{dot}(X_c, P_c) \\ Y_c \cdot x & Y_c \cdot y & Y_c \cdot z & -\text{dot}(Y_c, P_c) \\ Z_c \cdot x & Z_c \cdot y & Z_c \cdot z & -\text{dot}(Z_c, P_c) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

On a maintenant réduit l'ensemble des transformations 3D -> NDC aux trois transformations "Locale-vers-Monde" représentée par la World Matrix, "Monde-vers-Camera" représentée par la View Matrix ainsi que "Vers-NDC" représentée par la Projection Matrix.



Références :

[http://www.codinglabs.net/article\\_world\\_view\\_projection\\_matrix.aspx](http://www.codinglabs.net/article_world_view_projection_matrix.aspx)

<http://www.opengl-tutorial.org/fr/beginners-tutorials/tutorial-3-matrices/>

### Partie 3 – Correction Gamma

Avant d'aborder la suite il nous faut discuter d'un point très important: la **colorimétrie**.

Les couleurs que vous visualisez sur un moniteur sont dites "perceptuelles" c'est-à-dire que le moniteur restitue une "réponse" en termes de signaux qui correspondent au triplet RGB.

Mais en réalité cet aspect perceptuel tient à la fois du fonctionnement interne des convertisseurs du moniteur (la façon dont le courant est transformé en intensité lumineuse) et à la perceptuelle visuelle humaine.

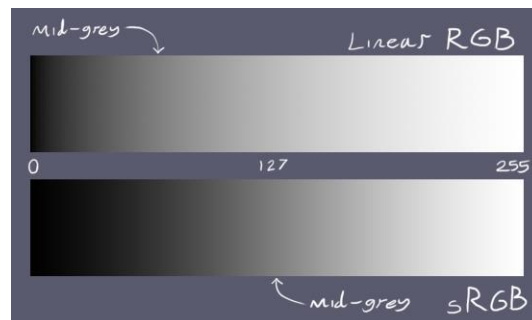
Premièrement, la perception visuelle humaine est plus à même de distinguer les variations d'intensité des couleurs très intenses (blancs) et très sombres (noirs) mais moins sensible dans les gris.

Secondement, le moniteur converti le courant en intensité à travers une loi de puissance (pow). Le facteur de puissance est appelé "gamma".

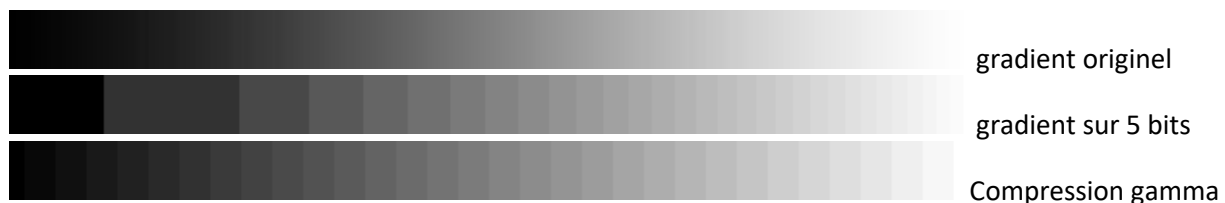
La plupart des moniteurs ont un gamma qui varie de l'ordre de 1.8 à 2.4. Il existe cependant une norme définie par Microsoft & HP nommée "sRGB" fixant le gamma approximativement à 2.2.

Tout ceci signifie deux choses: les moniteurs n'ont pas une réponse linéaire, c'est-à-dire que l'intensité croît en suivant une courbe de puissance (pow), et, cette courbe de puissance distribue plus de valeurs aux extrémités ce qui permet d'avoir des gradients plus lisses aux intensités minimales et maximales.

Pour prendre un exemple, lorsque vous souhaitez afficher un gris moyen à l'écran (127,127,127) - en OpenGL avec un appel à `glClearColor(0.5f, 0.5f, 0.5, 1.f)` - il s'agit d'un gris moyen dans le repère du moniteur, donc avec une correction (décompression) gamma. En linéaire ce gris moyen est plutôt un gris foncé (186,186,186)!



Prenons un autre exemple cette fois-ci afin de comprendre la finalité de la compression gamma. Supposons un gradient noir vers blanc que nous souhaitons afficher sur un écran ayant une très faible plage dynamique (32 couleurs, sur 5 bits donc), voici ce que cela donnerait :



C'est donc tout à la fois une bonne distribution des couleurs compte tenu de la faible dynamique (Low Dynamic Range – LDR) des moniteurs, et cela colle approximativement avec la vision humaine.

Cependant, cela pose un problème au programmeur : les opérations mathématiques -additions, soustractions, multiplications etc...- sont des opérations linéaires, mais les opérandes (les couleurs des pixels) ne sont pas linéaires (RGB) mais compressés gamma, on dira non-linéaire.

Ce que cela implique d'un point de vue colorimétrique :

- Le moniteur applique toujours une **décompression gamma** (mise à la puissance gamma)
- Une couleur visible sur un moniteur est donc toujours **non-linéaire**, car elle est exprimée dans le repère colorimétrique du moniteur.
- La plupart des formats d'image (PNG, JPEG etc...) stockent les images avec une **compression gamma** (donc non-linéaire également), majoritairement en sRGB 2.2.
- Une couleur compressée gamma (non-linéaire) traitée par un moniteur devient alors **linéaire** !

Il est alors nécessaire de se poser plusieurs questions lorsque l'on lit ou écrit des couleurs dans un shader.

(a) Si la couleur est linéaire on peut l'utiliser telle qu'elle dans nos opérations mathématiques. Autrement il faut la "linéariser" en appliquant une correction gamma.

Couleur\_SRC\_non-lineaire = couleur\_SRC<sup>1./2.2</sup> afin de corriger (linéariser) la couleur SRC il suffit d'appliquer la fonction inverse soit  $x^{2.2}$ , ce qui donne

$$\text{Couleur\_SRC\_non-lineaire}^{2.2} = (\text{Couleur\_SRC}^{1./2.2})^{2.2} = \text{Couleur\_SRC} !$$

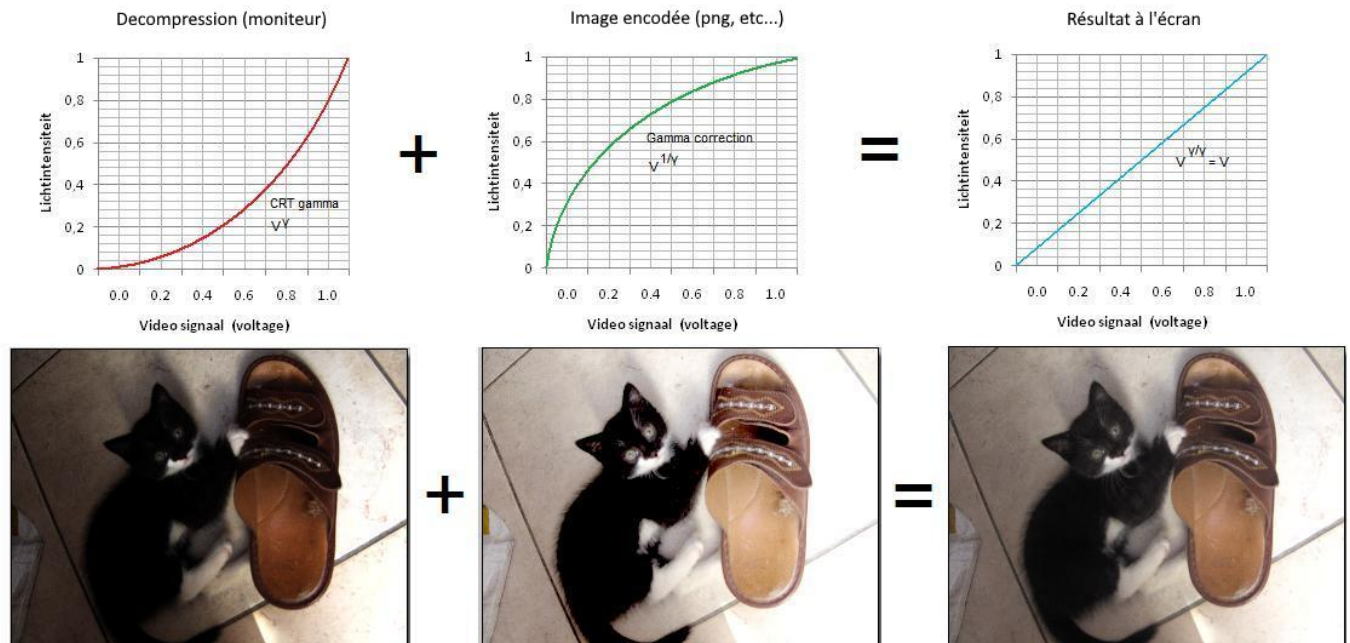
Comment savoir si une couleur en entrée est linéaire ? Il faut se poser la question de l'origine de la couleur. Est-ce que la couleur a été dessinée (image) ou capturée (color picker) ?

Si c'est le cas la couleur en entrée est non-linéaire.

Si au contraire la couleur a été générée par une équation (comme pour les normal maps, occlusion maps etc ...) alors les données en entrée sont déjà linéaires.

(b) Comme le moniteur applique automatiquement une correction gamma il faut donc s'assurer que les données que l'on va écrire dans le color buffer correspondent aux attentes du moniteur.

Ainsi, si la couleur en sortie du shader est non-linéaire (correction gamma 1./2.2, sRGB) tout est ok. Cependant si la couleur en sortie est linéaire il faut lui appliquer une compression gamma de 1./2.2.



**Exercice 3.1** appliquez les corrections gamma nécessaires en lecture et en écriture dans le Fragment Shader.

Heureusement pour nous, particulièrement pour les color buffers (y compris les textures) les GPUs sont capable d'effectuer pour nous ces conversions (à l'exception des couleurs passées en uniform ou attribut).

Ainsi on peut forcer la conversion automatique d'une texture lors de son échantillonnage en indiquant pour format interne un format sRGB comme **GL\_SRGB8** ou **GL\_SRGB8\_ALPHA8**.

Notez que ALPHA8 est séparé de SRGB8 ce qui indique que la composante Alpha est toujours linéaire.

En sortie, on peut forcer les écritures dans le color buffer à convertir automatiquement de linéaire vers gamma à l'aide de la fonction **glEnable(GL\_FRAMEBUFFER\_SRGB)**;

**Exercice 3.2** remplacez les changements précédents en appliquant les fonctions OpenGL aux textures ainsi qu'au Framebuffer.

## Références

<https://www.cambridgeincolour.com/tutorials/gamma-correction.htm>

[https://medium.com/@Jacob\\_Bell/programmers-guide-to-gamma-correction-4c1d3a1724fb](https://medium.com/@Jacob_Bell/programmers-guide-to-gamma-correction-4c1d3a1724fb)

<https://gamedevelopment.tutsplus.com/articles/gamma-correction-and-why-it-matters--gamedev-14466>

<https://learnopengl.com/Advanced-Lighting/Gamma-Correction>

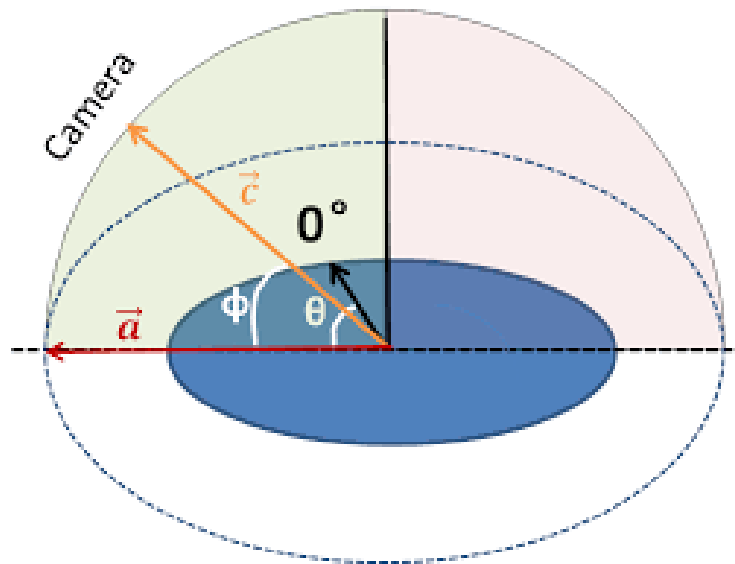


## TP à rendre – Caméra orbitale et Illumination ambiante

### 1. Caméra orbitale

Implémentez une caméra orbitale type “arcball” avec GLFW. Les “inputs” sont les suivants:

- Axe horizontal de la souris, rotation autour du “up” (azimut, aussi appelée assiette)
- Axe vertical de la souris, rotation autour de “right” (élévation)
- Molette de la souris, distance entre l’origine et la caméra



On considère que la position de la caméra est située sur une sphère de rayon R. Il est alors plus pratique de raisonner en coordonnées sphériques. Les mouvements des axes de la souris vont s’ajouter aux angle de rotation.

Utilisez la formule suivante pour déduire les coordonnées cartésiennes (X,Y,Z) à partir des coordonnées polaire (R, phi, theta), où “R” est le rayon, “phi” l’azimut et “theta” l’élévation.

$$Y = R * \sin(\text{Theta})$$

$$X = R * \cos(\text{Theta}) * \cos(\text{Phi})$$

$$Z = R * \cos(\text{Theta}) * \sin(\text{Phi})$$

Utilisez ensuite la fonction LookAt() avec comme “target” la position de votre objet.

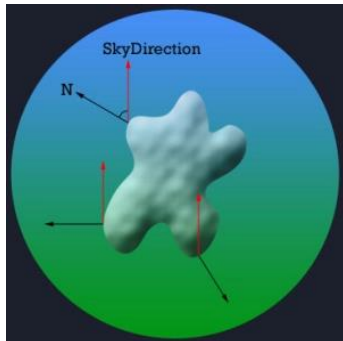
Attention ! Veillez à Limiter les angles horizontaux à  $(-\pi, +\pi)$ , et verticaux à  $(-\pi/2, +\pi/2)$ .

*Note: on peut également implémenter cette fonction sous la forme d’une rotation avec pivot. Les étapes sont : appliquez la matrice (–Translation), puis appliquez les rotations à l’origine, puis appliquez la matrice (Translation).*

## 2. Illumination hémisphérique ambiante (hemispherical ambient illumination)

Dans le TP précédent nous avons implémenté les bases du modèle d'illumination de Phong.

Nous allons ajouter un facteur d'ambiance qui, plutôt que d'être une couleur unie, va combiner deux couleurs, une pour l'hémisphère "up" (le ciel) et l'autre pour l'hémisphère "down" (le sol).



La technique repose sur l'utilisation d'un vecteur de référence ("*SkyDirection*" dans l'image, par exemple (0, 1, 0)) et l'utilisation du produit scalaire.

En fonction du signe du produit scalaire entre la normale du fragment et le vecteur de référence, on va choisir la couleur du ciel ou du sol comme valeur ambiante.

Mais on peut faire mieux: on va mélanger les deux couleurs en fonction de la valeur du produit scalaire en l'interprétant comme un pourcentage.

Cependant, le produit scalaire est défini entre [-1,+1] lorsque les vecteurs sont normalisés. Pour nous faciliter le mixage des couleurs on va re-parametrer le résultat produit scalaire de sorte à ce que le domaine soit [0,1].

Pour cela il suffit d'appliquer l'équation suivante :  $y = (x + 1) / 2 = x * \frac{1}{2} + \frac{1}{2}$ , ce qui nous donne :

```
HemisphereFactor = NdotSky * 0.5 + 0.5;
```

On utilise ensuite la fonction **mix()** du GLSL qui permet d'interpoler linéairement deux opérandes. On peut ainsi mixer deux couleurs SkyColor et GroundColor de façon paramétrique avec AmbientFactor.

```
vec3 AmbientColor = Ia * mix(SkyColor, GroundColor, HemisphereFactor);  
...  
vec3 FinalColor = AmbientColor + DiffuseColor + SpecularColor;
```

### Références

<https://thebookofshaders.com/glossary/?search=mix>

Unity implémente cette technique sous la forme du mode Trilight qui ajoute une troisième couleur pour l'équateur : <https://docs.unity3d.com/ScriptReference/Rendering.AmbientMode.Trilight.html>