CC2

1 Exercice : Modélisation

Dans cet exercice nous nous chargeons de lisser la surface en fonction des barycentres. Après avoir appliqué le filtre, on peut apercevoir que plus λ est proche de la valeur 1 et plus le lissage fonctionne bien.

Voici les résultats avec différents λ sur le chameau.

$\lambda = 0$

$\lambda = 0.5$

$\lambda = 0.7$

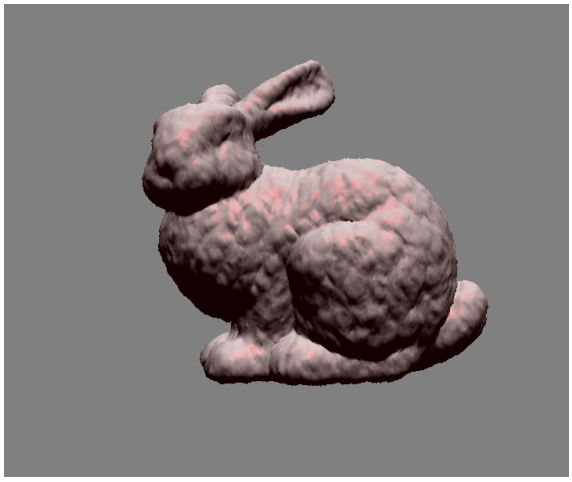
$\lambda = 1$

La même chose testé sur le lapin :

$\lambda = 0$

$\lambda = 0.5$

$\lambda = 0.7$

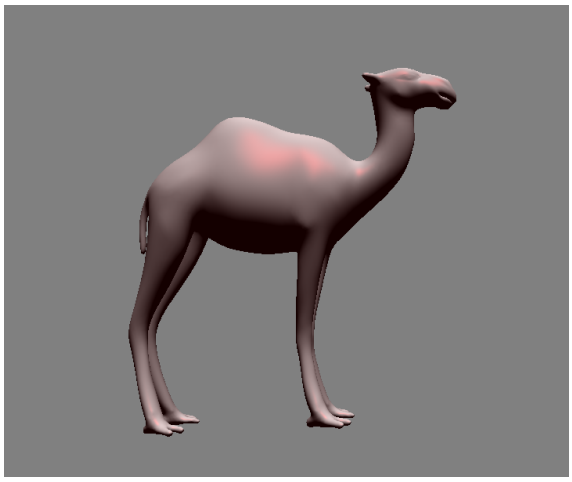


$\lambda = 1$

Bonus : on applique ensuite le second filtre : le filtre de Taubin. L'avantage de ce filtre, c'est qu'il permet de ne pas perdre de volume. Cela se voit facilement lorsque l'on applique plusieurs fois d'affilé les deux différents filtres :

Voici les résultats pour $\lambda = 0.5$ et $\mu = -0.51$ avec une itération puis 20 à la suite.

$\lambda = 0.5$



$\lambda = 0.5$, 20 applications

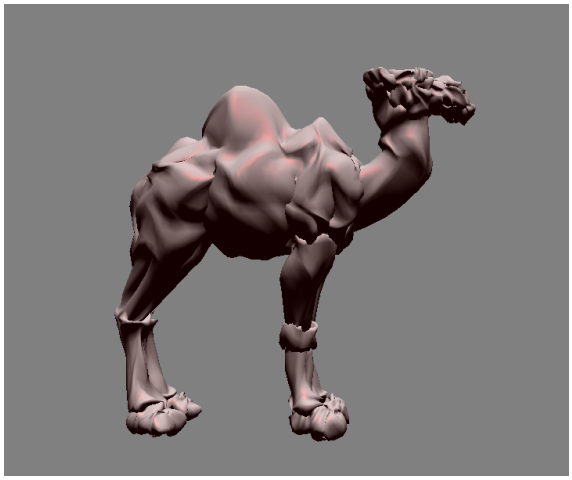
$\lambda = 0.5$, $\mu = -0.51$

$\lambda = 0.5$, $\mu = -0.51$, 20 applications

Note : j'ai fait en sorte que le filtre de Taubin s'applique en appuyant sur `t`. Il s'applique alors avec $\mu = -\lambda - 0.01$.

En revanche, appliquer le filtre de Taubin sur des valeurs extrêmes ne donne pas des résultats convainquants. C'est également le cas si μ est vraiment plus grand que λ (résultats après 20 applications du filtre pour bien voir le problème) :

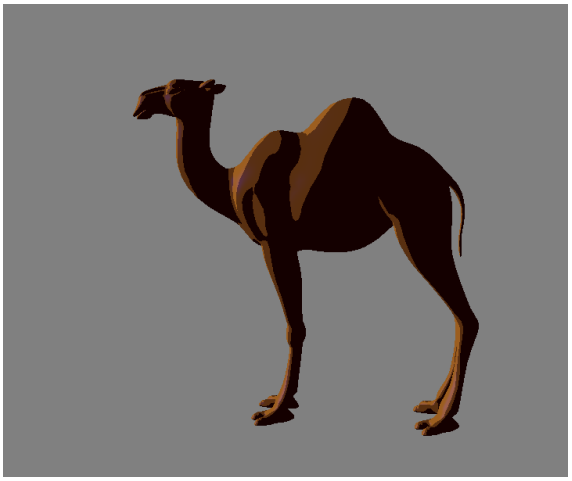
$\lambda = 0.99$, $\mu = -1$



$\lambda = 0.5$, $\mu = -0.8$

2 Exercice : Rendu

Voici les rendus après application du *Cel shading* sur le dromadaire :



$levels = 3$

$levels = 4$

$levels = 5$

On distingue nettement les *levels*, qui forment comme des zones entières en fonction de la lumière.

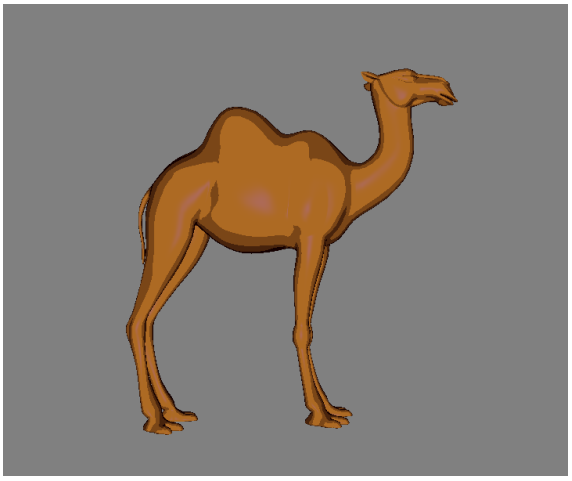
La lumière étant de base dirigée sur la droite, il est possible de la déplacer dans le code (dans la méthode `initLights`). Ici j'ai choisi d'utiliser la lumière numéro 3 (donc d'indice 2).

Nous devons placer la lumière sur la caméra et pour faire cela, nous avons besoin de récupérer la position de cette dernière avant de l'appliquer à la lumière. Cette information est récupérable facilement via la méthode `getPosition(float&, float&, float&)` de la classe `Camera`.

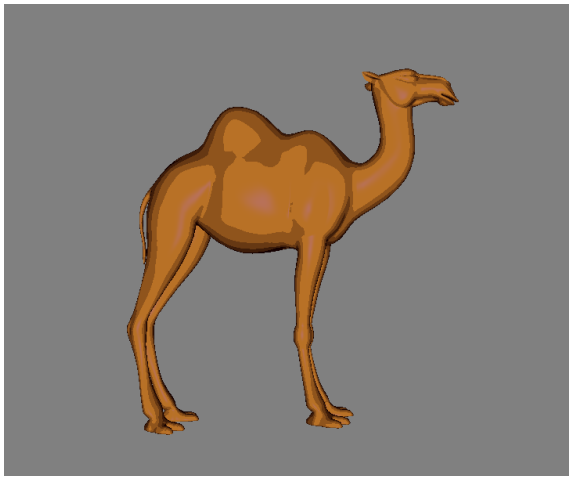
Cette modification est faite aisément grâce au code suivant :

```
void initLights () {
    // On récupère la position de la caméra
    float cam_pos_x, cam_pos_y, cam_pos_z;
    camera.getPosition(cam_pos_x, cam_pos_y, cam_pos_z);

    // ...

    // On l'applique notre nouvelle position et on commente l'ancienne
    GLfloat light_position_2[4] = {cam_pos_x, cam_pos_y, cam_pos_z, 0};
    //GLfloat light_position_2[4] = {-438, 167, -48, 0};

    // ...
}
```

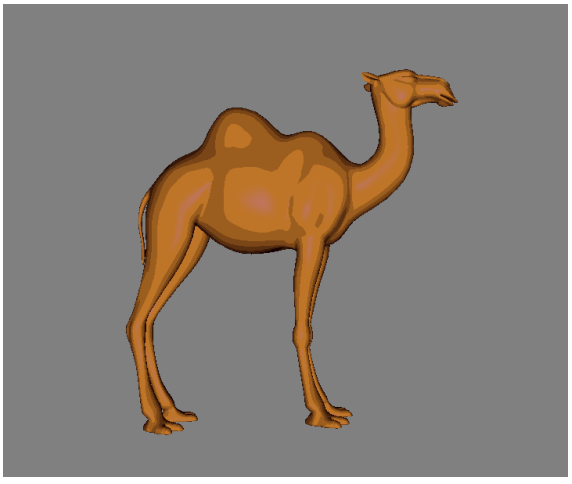
Maintenant que la lumière est déplacée, on voit nettement mieux les formes ! Voici le résultat obtenu :



$levels = 3$



$levels = 4$

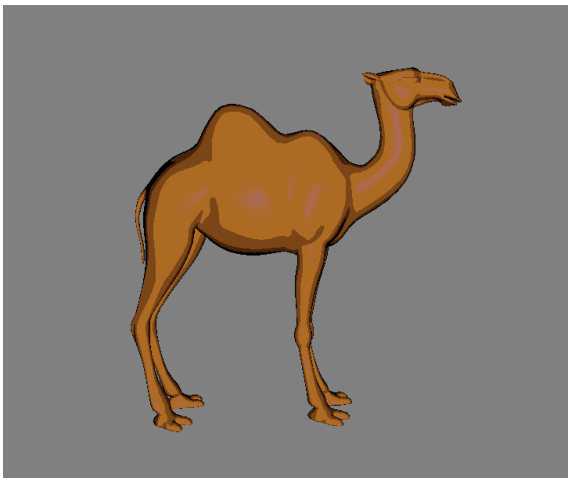


$levels = 5$

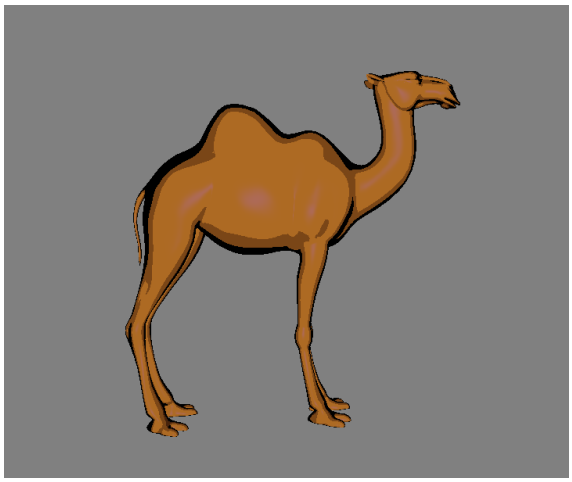
De base, lorsque le produit scalaire de la droite normale en un point (ou une face) et le vecteur de vue est négatif, c'est que le point n'est pas visible par la vue. Donc il n'a pas de couleur.

On a donc une couleur noire (mais qui n'est pas visible) lorsque $\vec{N}.\vec{V} < k$ avec $k = 0$. Pour ajouter un effet de bordure à notre modèle 3D, il suffit donc d'augmenter la valeur k.

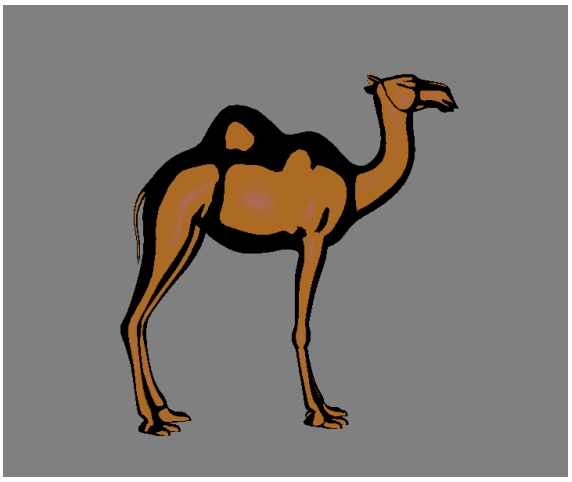
Voici les résultats pour différentes valeurs de k (testé avec $levels = 3$).



$k = 0.3$



$k = 0.5$



$k = 0.8$

3 Exercice : Animation

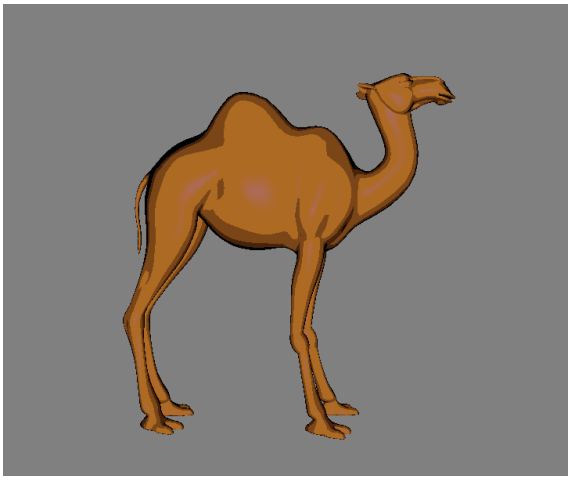
Pour faire une interpolation linéaire d'un point A un point B, il suffit d'appliquer la fonction suivante :

$$P = (1 - t)A + tB$$

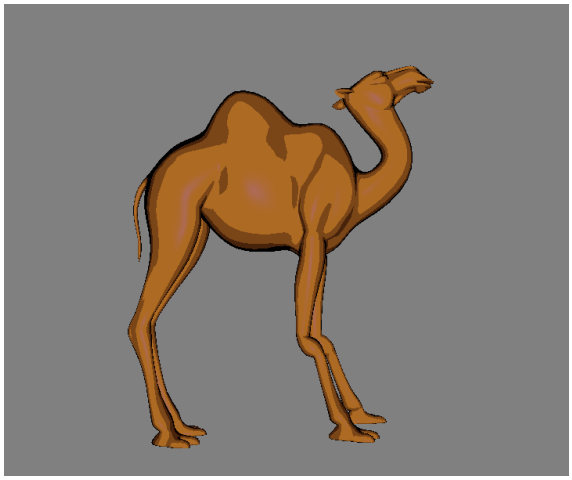
où $t \in [0, 1]$ représente le coefficient d'interpolation.

Dans notre cas nous avons les valeurs suivantes pour chaque point P d'indice i :
— $P =$ `current_mesh[i]`
— $A =$ `mesh_pose_0[i]`
— $B =$ `mesh_pose_1[i]`
— $t =$ `interpolant1`

Voici quelques résultats pour différentes valeurs de t.



$t = 0.5$



$t = 1$

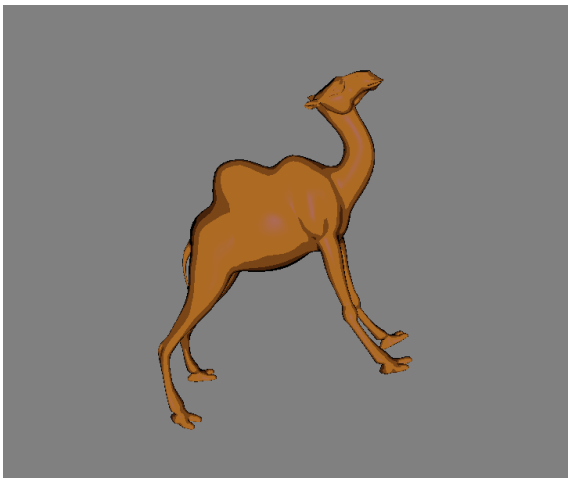
L'interpolation entre les trois formes est presque identique, la formule change juste un peu, mais la somme des poids pour chaque sommet est toujours égale à 1 donc il n'y a aucune normalisation à faire.

$$P = (1 - t_1 - t_2)A + t_1 B + t_2 C$$

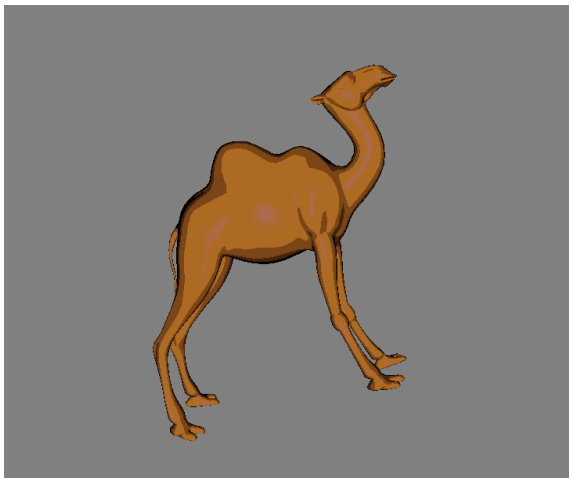
avec $t_1 + t_2 = 1$.

Dans notre cas nous avons les valeurs suivantes pour chaque point P d'indice i :
— $P =$ `current_mesh[i]`
— $A =$ `mesh_pose_0[i]`
— $B =$ `mesh_pose_1[i]`
— $C =$ `mesh_pose_2[i]`
— $t_1 =$ `interpolant1`
— $t_2 =$ `interpolant2`

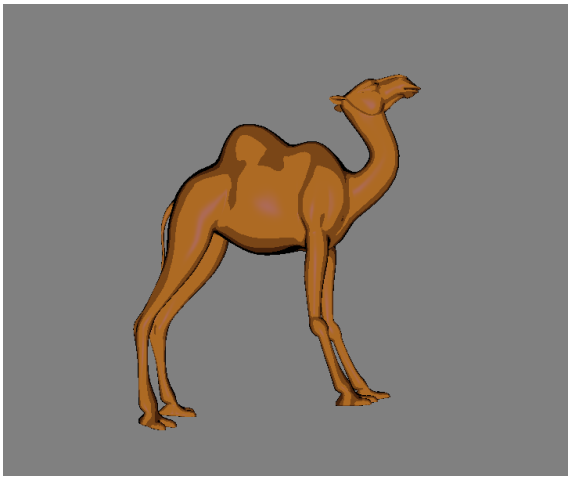
Voici quelques résultats pour différentes valeurs de t_1 et t_2.



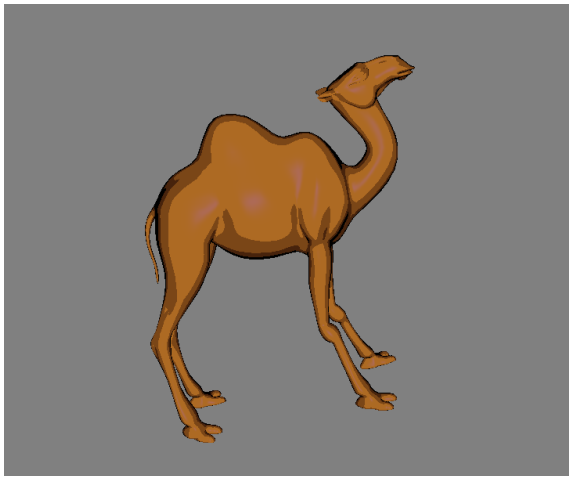
$t_1 = 0$, $t_2 = 1$



$t_1 = 0.3$, $t_2 = 0.7$



$t_1 = 0.5$, $t_2 = 0.5$



$t_1 = 0.7$, $t_2 = 0.3$

Bonus : Pour faire une animation continue, il faut simplement modifier une fonction qui est appelée par OpenGL à chaque frame. C'est le cas par exemple de la fonction `display()`.

Voici le petit bout de code ajouté qui a permis de faire cela :

```
void display () {
    if (animate) {
        static bool animation_revert = false;
        interpolant1 += 0.01f * (animation_revert ? -1 : 1);
        if (abs(1 - interpolant1) < 1e-6 || interpolant1 < 1e-6) {
            animation_revert = ! animation_revert;
        }
        updateInterpolation();
    }

    // ...
}
```

où `animate` est une nouvelle variable globale qui peut être activée en appuyant sur `g` (l'animation ne se lance pas de base).