Surfaces2

November 26, 2020

1 Tracé de surfaces (II)

Marc Lorenzi

26 novembre 2020

[1]:
```
import matplotlib.pyplot as plt
from vect_utils import *
import surf01
```

[2]:
```
plt.rcParams['figure.figsize'] = (8, 6)
```

Je pars du principe que vous avez lu le premier notebook sur le tracé de surfaces. Un certain nombre de fonctions que nous y avons écrites sont stockées dans le fichier `surf01.py` et ne seront pas ré-expliquées ici. De même, le module `vect_utils` contient des fonctions permettant d'effectuer quelques opérations sur les vecteurs de $\mathbb{R}^3$.

Ce premier notebook nous a permis de dessiner une surface en éliminant ses parties cachées par **l'algorithme du peintre**. L'un des défauts de nos tracés était que l'on distinguait parfaitement les faces de la surface qui étaient des triangles adjacents de couleurs différentes. Ce que nous nous proposons de faire ici, c'est

- de « lisser » la surface, c'est à dire de faire disparaître ces discontinuités dans les couleurs en utilisant des algorithmes d'interpolation. Nous allons étudier deux de ces algorithmes : celui de **Gouraud** et celui de **Phong**.
- d'améliorer le modèle d'**illumination** en y incluant ce que l'on appelle la **réflexion spéculaire**. Le modèle que nous allons mettre en place est le modèle de Phong (le même Phong que pour le lissage). Ce modèle est utilisé par exemple dans les technologies OpenGL et DirectX.

1.1 1. Interpolation

Commençons par examiner comment l'interpolation peut résoudre nos problèmes de coloriage.

1.1.1 1.1 Barycentres

Soit $T = [A, B, C]$ un triangle du plan. Tout point $M \in \mathbb{R}^2$ est un **barycentre** de A, B, C. Précisément, il existe trois réels x, y, z vérifiant $x + y + z = 1$ tels que, pour tout point Ω,

$$\overrightarrow{\Omega M} = x\overrightarrow{\Omega A} + y\overrightarrow{\Omega B} + z\overrightarrow{\Omega C}$$

Que valent x, y, z ? Prenons $\Omega = A$. Il vient

$$\overrightarrow{AM} = y\overrightarrow{AB} + z\overrightarrow{AC}$$

Posons $\overrightarrow{AB} = (a, c)$, $\overrightarrow{AC} = (b, d)$ et $\overrightarrow{AM} = (p, q)$. On obtient le système

$$(S) \quad \begin{cases} ay + bz &= p \\ cy + dz &= q \end{cases}$$

Ce système se résout sans difficulté. En posant $D = ad - bc$, on obtient

$$\begin{cases} Dy &= pd - qb \\ Dz &= qa - pc \end{cases}$$

d'où y, z et $x = 1 - y - z$.

La fonction `coef_barycentre` prend en paramètres un point M du plan et un triangle $T = [A, B, C]$. Elle renvoie le triplet (x, y, z) que nous venons de calculer. Elle utilise la petite fonction `vecteur2` qui renvoie le vecteur d'origine A et d'extrémité B dans le plan.

[3]:
```
def vecteur2(A, B):
    a, b = A
    c, d = B
    return (c - a, d - b)
```

[4]:
```
def coef_barycentre(M, T):
    A, B, C  = T
    a, c = vecteur2(A, B)
    b, d = vecteur2(A, C)
    p, q = vecteur2(A, M)
    D = a * d - b * c
    if D == 0: return (None, None, None)
    else:
        y = (p * d - q * b) / D
        z = (q * a - p * c) / D
        return (1 - y - z, y, z)
```

[5]:
```
T = [(0,0), (2,0), (1,1)]
coef_barycentre((0.5, 0.5), T)
```

[5]:
```
(0.5, 0.0, 0.5)
```

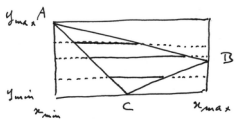
1.1.2 1.2 Lissage

Supposons donné un triangle $T = [A, B, C]$ dont nous possédons les couleurs des sommets, c_A, c_B et c_C. Comment colorier de façon judicieuse le reste du triangle ? Pour tout point M intérieur au triangle, soient α, β, γ tels que M soit le barycentre de A, B, C affectés des coefficients α, β, γ. L'idée est de colorier le point M avec la couleur

$$c = \alpha c_A + \beta c_B + \gamma c_C$$

Mais comment trouver les points qui sont à l'intérieur du triangle ? L'idée est de tout d'abord inclure le triangle dans un rectangle. La fonction `bornes` trouve les coordonnées minimales et maximales de ce rectangle.

[6]:
```
def bornes(T):
    A, B, C = T
    xmin = min(A[0], B[0], C[0])
    xmax = max(A[0], B[0], C[0])
    ymin = min(A[1], B[1], C[1])
    ymax = max(A[1], B[1], C[1])
    return (xmin, xmax, ymin, ymax)
```



On « scanne » ensuite le rectangle en parcourant toutes les lignes horizontales (par exemple) de celui-ci. Pour chaque point scanné, on calcule les coefficients α, β, γ. Le point est dans le triangle si et seulement si ces trois coefficients sont positifs ou nuls.

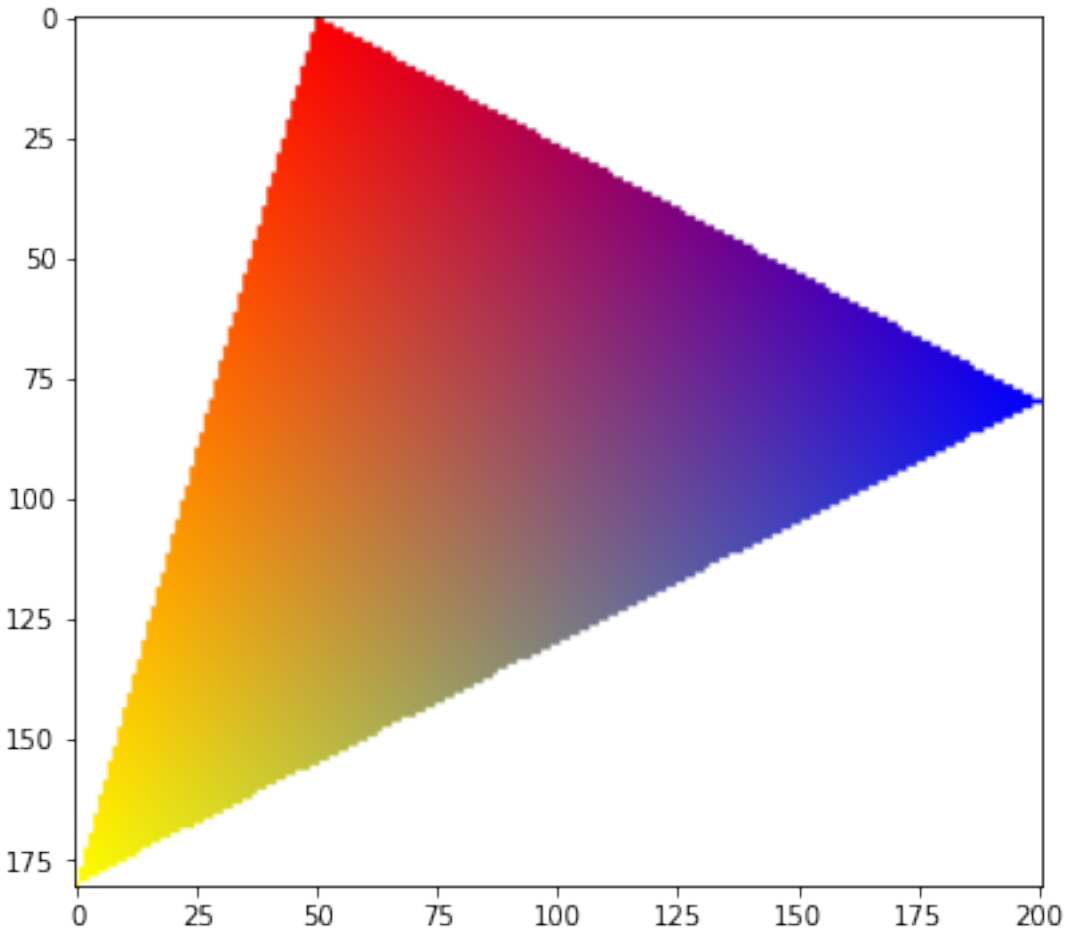
La fonction `test_lissage` va nous permettre de constater que notre idée est prometteuse … À partir d'un triangle et de trois couleurs affectées à ses sommets, elle interpole ces couleurs à tous les points du triangle, puis elle affiche le résultat.

[7]:
```
def test_lissage(T, couls):
    xmin, xmax, ymin, ymax = bornes(T)
    m = (xmax - xmin + 1) * [None]
    for i in range(xmax - xmin + 1): m[i] = (ymax - ymin + 1) * [[1, 1, 1]]
    for x in range(xmin, xmax + 1):
        for y in range(ymin, ymax + 1):
            a, b, c = coef_barycentre((x, y), T)
            if a >= 0 and b >= 0 and c >= 0:
                coul = add(mul(a, couls[0]), add(mul(b, couls[1]), mul(c,␣
↪couls[2])))
                m[x - xmin][y - ymin] = coul
    plt.imshow(m)
```

Testons tout cela sur un triangle avec trois sommets de couleurs rouge, jaune et bleue. Profitez-en pour remarquer que `matplotlib` interprète un triplet de réels appartenant à $[0, 1]$ comme une

couleur de type (R, G, B) (c'est à dire Red, Green, Blue).

[8]:
```
test_lissage([(20, 50), (200, 0), (100, 200)], [(1, 0, 0), (1, 1, 0), (0, 0, 1)])
```



1.2 2. Le lissage de Gouraud (Gouraud Shading)

Inutile de faire durer le suspense. L'algorithme de Gouraud trace chacune des faces d'une surface en calculant la couleur à affecter aux sommets de la face, puis en interpolant ces couleurs aux autres points de la face.

1.2.1 2.1 Scènes

Comme nous allons devoir manipuler beaucoup de paramètres en même temps, il n'est pas inutile de définir une classe `Scene` qui permettra de tout stocker. Un objet de cette classe possède un certain nombre de champs que nous décrirons au fur et à mesure que nous les utiliserons. Les paramètres du constructeur de la classe sont

- Une fonction F qui définit la surface paramétrée que nous voulons dessiner.
- Un paramètre `bornes` qui contient les valeurs minimales et maximales des paramètres de la surface.

Nous avons déjà utilisé ces deux paramètres dans le premier notebook.

- Un paramètre `taille_discr` qui est un couple (n_u, n_v) qui nous dit en combien de morceaux il faut discrétiser la surface (c'est à dire en gros le nombre de faces).
- Un paramètre `Obs` qui est un triplet de flottants donnant la position de l'observateur.
- Un paramètre `light` qui est un vecteur de $\mathbb{R}^3$ donnant la direction de la source de lumière.
- Un paramètre `taille_canv` qui est un couple d'entiers dont nous parlerons un peu plus loin.

[9]:
```
def matrice(p, q, a):
    m = p * [None]
    for i in range(p):
        m[i] = q * [a]
    return m
```

[10]:
```
class Scene:

    def __init__(self, F, bornes, taille_discr, Obs, light, taille_canv):
        self.F = F
        self.umin, self.umax, self.vmin, self.vmax = bornes
        self.nu, self.nv = taille_discr
        self.Obs = Obs
        self.base = surf01.base_adaptee(Obs)
        self.light = normaliser(light)
        self.mat_3d = matrice(self.nu + 1, self.nv + 1, (0, 0, 0))
        self.mat_proj = matrice(self.nu + 1, self.nv + 1, (0, 0))
        self.mat_norm = matrice(self.nu + 1, self.nv + 1, (0, 0, 0))
        self.ni, self.nj = taille_canv
        self.canvas = matrice(self.nj + 1, self.ni + 1, (0.2, 0.1, 0))
```

1.2.2 2.2 Discrétiser les points de la surface

Comme dans le premier notebook, la première étape est la discrétisation. On commence par discrétiser les **sommets** des faces. On stocke tous les résultats dans le champ `S.mat_3d` de la scène S. C'est tout l'avantage d'avoir défini une classe, l'objet S peut stocker tout ce dont nous aurons besoin plus tard.

[11]:
```
def discretiser_points(S):
    us = surf01.subdi(S.umin, S.umax, S.nu)
    vs = surf01.subdi(S.vmin, S.vmax, S.nv)
    S.mat_3d = [[S.F(u,v) for v in vs] for u in us]
```

1.2.3 2.3 Discrétiser les normales

Tant que nous y sommes, discrétisons aussi les vecteurs normaux aux sommets des faces. Stockons les dans le champ `mat_norm` de l'objet S qui représente la scène.

Rappelons que si la surface paramétrée est donnée par $(u, v) \mapsto F(u, v)$, un vecteur normal au point de paramètre (u, v) est

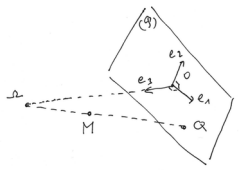
$$\overrightarrow{n} = \frac{\partial F}{\partial u} \wedge \frac{\partial F}{\partial v}$$

On normalise ensuite ce vecteur en le divisant par sa norme. Si jamais cette norme est nulle, votre ordinateur explose . Bon, en fait c'est pas vrai : la fonction `normaliser` renvoie le vecteur nul lorsque le vecteur est nul.

[12]:
```
def vecteur_normal(F, u, v, bornes):
    w1 = surf01.diffu(F, u, v, bornes)
    w2 = surf01.diffv(F, u, v, bornes)
    n = prod_vect(w1, w2)
    return normaliser(n)
```

[13]:
```
def discretiser_normales(S):
    us = surf01.subdi(S.umin, S.umax, S.nu)
    vs = surf01.subdi(S.vmin, S.vmax, S.nv)
    S.mat_norm = [[vecteur_normal(S.F, u, v, (S.umin, S.umax, S.vmin, S.vmax))␣
 ↪for v in vs] for u in us]
```

1.2.4 2.4 Projeter



Je ne ferai aucun commentaire sur la phase de projection, si ce n'est que les projections des sommets des faces sont stockées dans le champ `mat_proj` de l'objet S. Voir le premier notebook pour des détails !

[14]:
```
def projeter(S):
    us = surf01.subdi(S.umin, S.umax, S.nu)
    vs = surf01.subdi(S.vmin, S.vmax, S.nv)
    S.mat_proj = [[surf01.projeter_point(S.F(u,v), S.Obs, S.base) for v in vs]␣
 ↪for u in us]
```

1.2.5 2.5 Mettre à l'échelle

C'est ici qu'il convient de réaliser que certaines des opérations faites dans le premier notebook relevaient de la magie pure, ou plutôt de la technique de la boîte noire : on créait des `Polygon`, on faisait quelques `plot`, et hop ! la surface s'affichait magiquement, pafraitement centrée dans le graphique, les polygones bien remplis avec la bonne couleur … Mais nous ne somme plus à l'école des sorciers. Nous devons maintenant remplir nos polygones pixel par pixel, et afficher une image en deux dimensions de notre surface en 3 dimensions.

Étape 1. Quelles sont les coordonnées minimales et maximales des points de notre surface projetée ? La fonction `bornes_proj` trouve ces coordonnées.

[15]:
```
def bornes_proj(S):
    ws = [S.mat_proj[u][v] for u in range(S.nu + 1) for v in range(S.nv + 1)]
    wsx = [w[0] for w in ws]
    wsy = [w[1] for w in ws]
    return (min(wsx), max(wsx), min(wsy), max(wsy))
```

Étape 2. Nous allons stocker une **image** de la surface dans une matrice (appelée par les initiés un **canvas**) qui est stockée dans le champ `canvas` de la scène. Chaque élément de la matrice est un futur pixel de l'image. Quelle est la taille de ce canvas ? Le paramètre `taille_canv` du constructeur de la classe `Scene` contient justement un couple (n_i, n_j) où n_i et n_j sont le nombre de lignes et le nombre de colonnes de la matrice en question.

Un point de la surface projetée est un point du plan $M = (x, y)$ où $x_{min} \leq x \leq x_{max}$ et $y_{min} \leq y \leq y_{max}$. Il s'agit de transformer ce point en un pixel $P = (i, j)$ où $0 \leq i \leq n_i$ et $0 \leq j \leq n_j$.

Appelons C le centre de la surface projetée et C' le centre du canvas.

$$C = (x_{med}, y_{med}) = \frac{1}{2}(x_{min} + x_{max}, y_{min} + y_{max})$$

et

$$C' = \frac{1}{2}(n_i, n_j)$$

Posons ensuite

$$Z = \frac{9}{10} \min\left(\frac{n_i}{x_{max} - x_{min}}, \frac{n_j}{y_{max} - y_{min}}\right)$$

Z est le **facteur de zoom**. On prend alors

$$P = C' + Z(M - C)$$

Tout cela paraît un peu compliqué mais l'idée est que

- Notre image de la surface soit centrée dans le canvas.

- Qu'il reste un peu de marge sur les bords, d'où le facteur $\frac{9}{10}$ (un facteur 1 aurait collé la surface au bord).
- Que l'image ne soit pas déformée, d'où la multiplication de x et y par un même facteur.

La fonction `mettre_echelle` modifie les valeurs de la matrice `mat_proj`, qui sont des couples de flottants, pour les transformer en couples d'entiers compatibles avec la taille du canvas.

[16]:
```
def mettre_echelle(S):
    xmin, xmax, ymin, ymax = bornes_proj(S)
    xmed = (xmin + xmax) / 2
    ymed = (ymin + ymax) / 2
    Z = 0.9 * min(S.ni / (xmax - xmin), S.nj / (ymax - ymin))
    for u in range(S.nu + 1):
        for v in range(S.nv + 1):
            x, y = S.mat_proj[u][v]
            x1 = math.floor(S.ni / 2 + Z * (x - xmed) + 0.5)
            y1 = math.floor(S.nj / 2 + Z * (y - ymed) + 0.5)
            S.mat_proj[u][v] = (x1, y1)
```

Une fois la fonction `mettre_echelle` exécutée, le champ `mat_proj` de l'objet S contient des coordonnées de pixels.

1.2.6 2.7 Les faces d'une scène, tracé

Ici, rien de changé par rapport à ce qui a été dit dans le premier notebook.

On commence par calculer la liste des faces de la surface :

[17]:
```
def faces(S):
    s1 = [[(u, v), (u+1, v), (u+1, v+1)] for u in range(S.nu) for v in range(S.
↪nv)]
    s2 = [[(u, v), (u, v+1), (u+1, v+1)] for u in range(S.nu) for v in range(S.
↪nv)]
    return s1 + s2
```

La fonction de tracé est maintenant facile à écrire :

1. On discrétise les points de la surface (fonction `discretiser_points`).
2. On discrétise les normales (fonction `discretiser_normales`).
3. On projette (fonction `projeter`).
4. On met à l'échelle du canvas (fonction `mettre_echelle`).
5. On calcule les faces (fonction `faces`).
6. On trie ces faces par distances décroissantes à l'observateur.
7. On trace les faces (fonction `tracer_triangle`).

[18]:
```
def centre_face(S, T):
    x1, y1, z1 = S.mat_3d[T[0][0]][T[0][1]]
    x2, y2, z2 = S.mat_3d[T[1][0]][T[1][1]]
    x3, y3, z3 = S.mat_3d[T[2][0]][T[2][1]]
```

```
    return ((x1 + x2 + x3) / 3, (y1 + y2 + y3) / 3, (z1 + z2 + z3) / 3)
```

[19]:
```
def distance2(A, B):
    return norme2(vecteur(A, B))
```

[20]:
```
def tracer_gouraud(S):
    discretiser_points(S)
    discretiser_normales(S)
    projeter(S)
    mettre_echelle(S)
    fs = faces(S)
    fs.sort(key=lambda T: -distance2(centre_face(S, T), S.Obs))
    for T in fs:
        tracer_triangle(S, T)
    plt.axis('off')
    plt.imshow(S.canvas)
```

1.2.7 2.8 Colorier, enfin !

Oui, effectivement, le lecteur sagace aura remarqué que la fonction `tracer_triangle` n'est pas encore écrite. Il nous suffit de reprendre les idées vues dans la section 1. sur l'interpolation. Pour dessiner le triangle T, on calcule un rectangle qui englobe ce triangle.

[21]:
```
def bornes_triangle(S, T):
    x0, y0 = S.mat_proj[T[0][0]][T[0][1]]
    x1, y1 = S.mat_proj[T[1][0]][T[1][1]]
    x2, y2 = S.mat_proj[T[2][0]][T[2][1]]
    return (min(x0, x1, x2), max(x0, x1, x2), min(y0, y1, y2), max(y0, y1, y2))
```

On calcule les couleurs des sommets de ce triangle en utilisant la même technique de coloration que celle du premier notebook.

[22]:
```
def couleurs_sommets_triangle(S, T):
    c1 = couleur_sommet(S, T[0][0], T[0][1])
    c2 = couleur_sommet(S, T[1][0], T[1][1])
    c3 = couleur_sommet(S, T[2][0], T[2][1])
    return (c1, c2, c3)
```

[23]:
```
def couleur_sommet(S, u, v):
    n = S.mat_norm[u][v]
    I = 0.3 + 0.7 * abs(prod_scal(n, S.light))
    if I >= 1: I = 1
    return (I, 0.7 * I, 0)
```

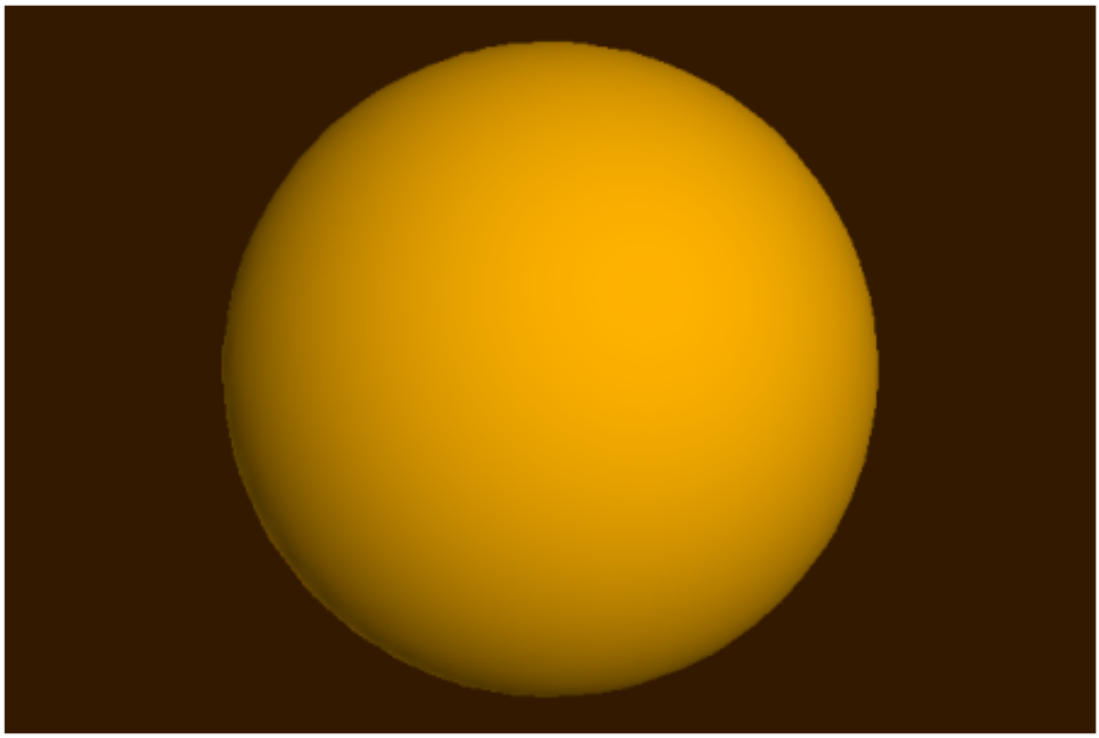
Le tracé du triangle T par l'algorithme de Gouraud consiste à colorier les points de T en interpolant les couleurs des sommets de T.

[24]:
```
def tracer_triangle(S, T):
    A = S.mat_proj[T[0][0]][T[0][1]]
    B = S.mat_proj[T[1][0]][T[1][1]]
    C = S.mat_proj[T[2][0]][T[2][1]]
    T1 = [A, B, C]
    imin, imax, jmin, jmax = bornes(T1)
    c1, c2, c3 = couleurs_sommets_triangle(S, T)
    for i in range(imin, imax + 1):
        for j in range(jmin, jmax + 1):
            a, b, c = coef_barycentre((i, j), T1)
            if a != None and a >= 0 and b >= 0 and c >= 0:
                S.canvas[S.nj - j][i] = add(mul(a, c1), add(mul(b, c2), mul(c,␣
 ↪c3)))
```

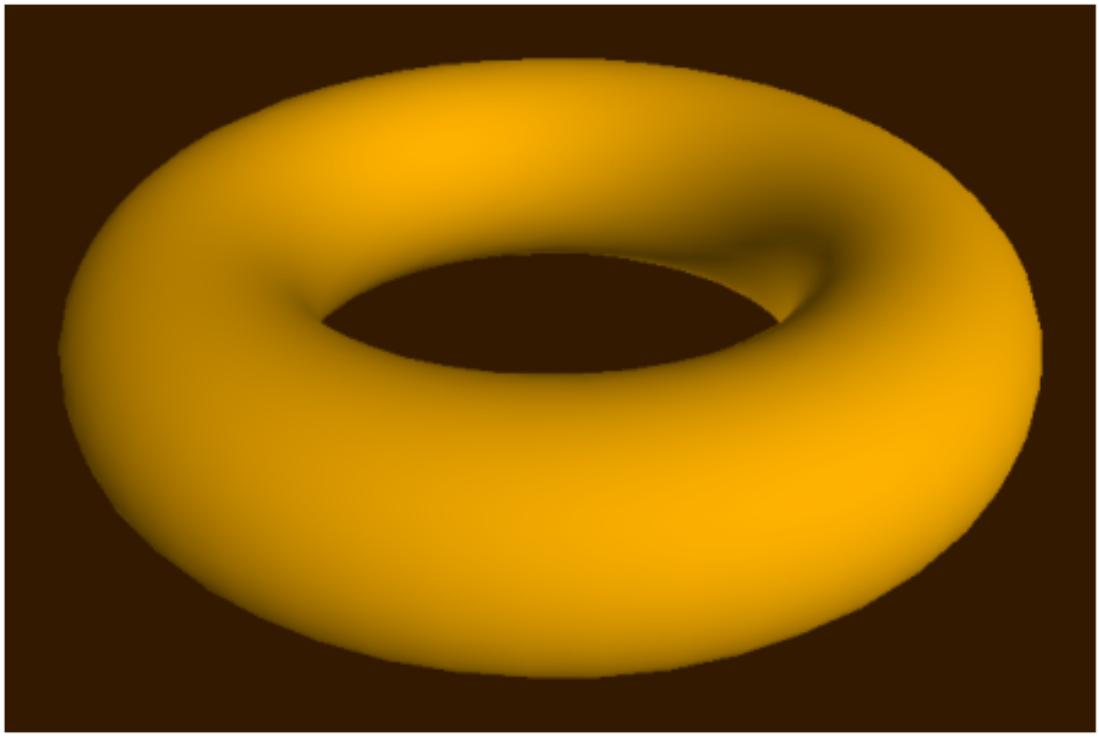
Enfin nous y sommes. Nous allons pouvoir dessiner nos surfaces préférées.

1.2.8 2.9 Exemples

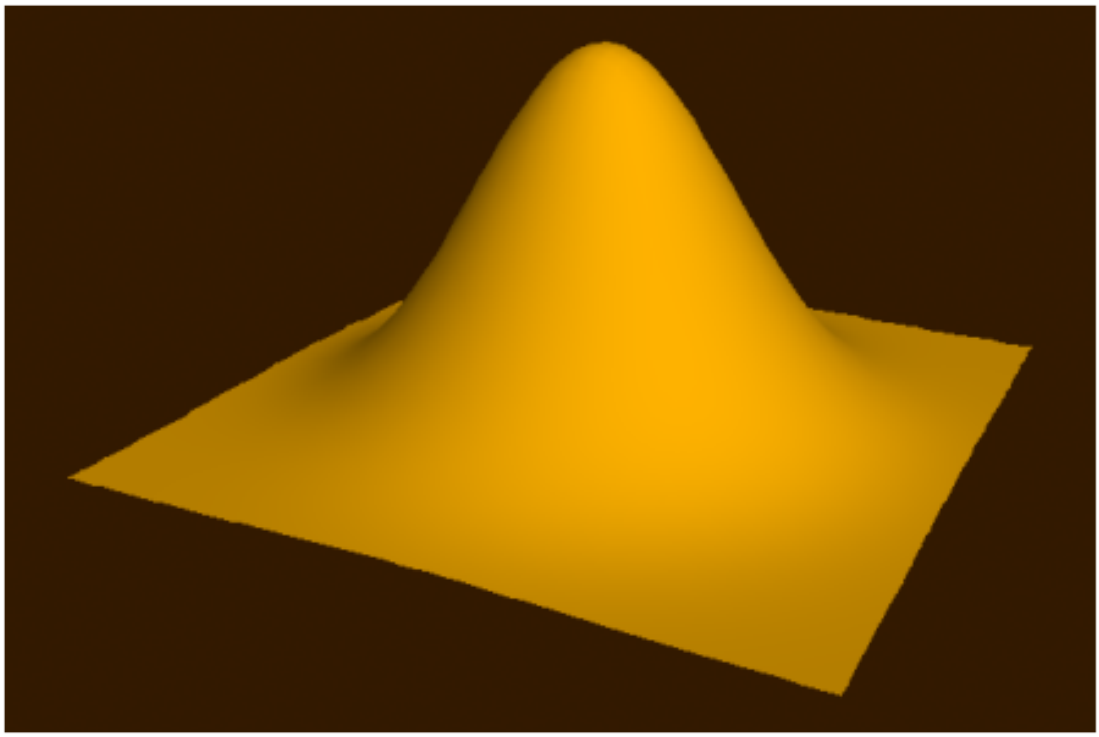
[25]:
```
S = Scene(surf01.F_sphere, surf01.bornes_sphere, (50, 50), (6, 3, 3), (1, 1, 1),␣
 ↪(600, 400))
tracer_gouraud(S)
```



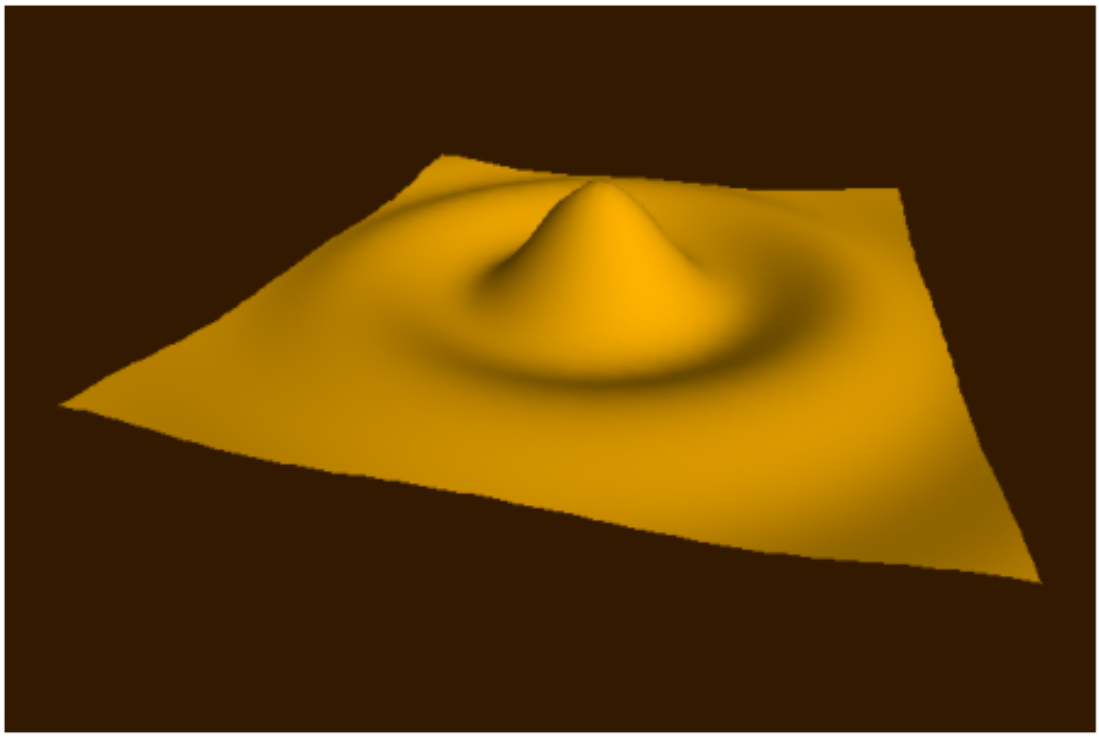
[26]:
```
S = Scene(surf01.F_tore, surf01.bornes_tore, (50, 50), (10, 3, 6), (1, 1, 1),␣
↪(600, 400))
tracer_gouraud(S)
```



[27]:
```
S = Scene(surf01.F_exp, surf01.bornes_exp, (50, 50), (6, 3, 3), (1, 1, 1), (600,␣
↪400))
tracer_gouraud(S)
```



[28]:
```
S = Scene(surf01.F_sin, surf01.bornes_sin, (50, 50), (20, 5, 10), (1, 1, 1),␣
↪(600, 400))
tracer_gouraud(S)
```



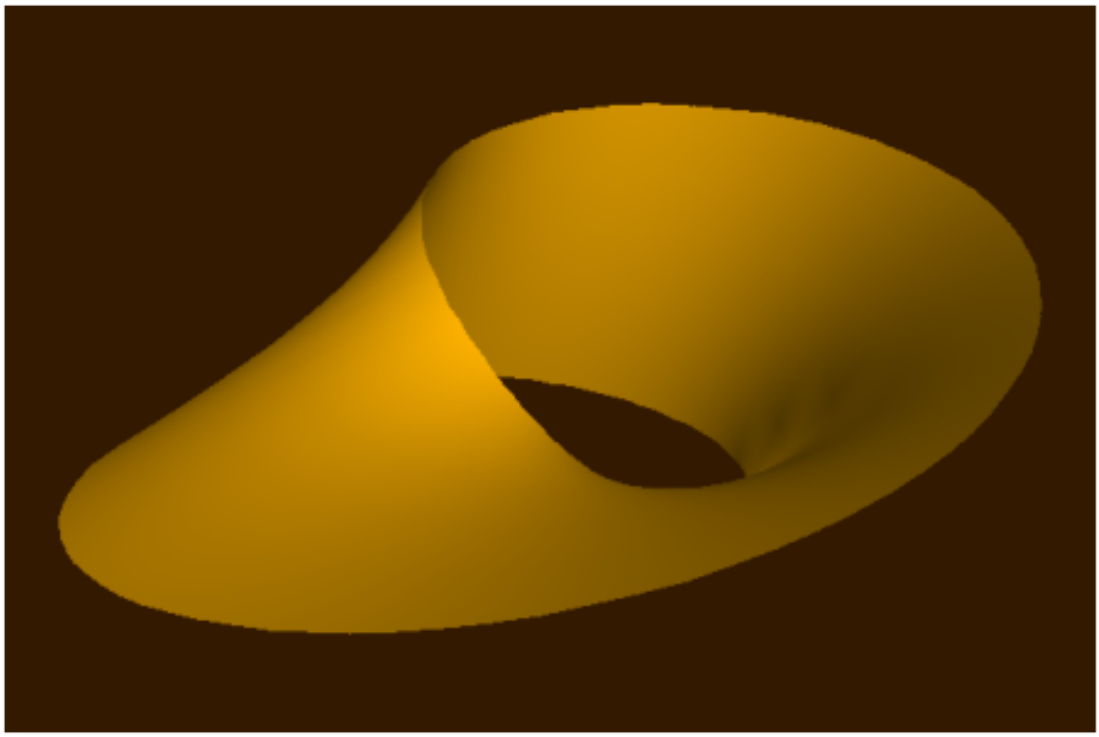
1.2.9 2.10 Encore deux exemples

Nous avions également dans le premier notebook tracé un **ruban de Möbius** et une **surface de Boy**. Les voici.

[29]:
```
def F_mobius(s, t, R=3):
    x = (R + s * math.cos(t / 2)) * math.cos(t)
    y = (R + s * math.cos(t / 2)) * math.sin(t)
    z = s * math.sin(t / 2)
    return (x, y, z)

bornes_mobius = [-2, 2, 0, 2 * math.pi]
```

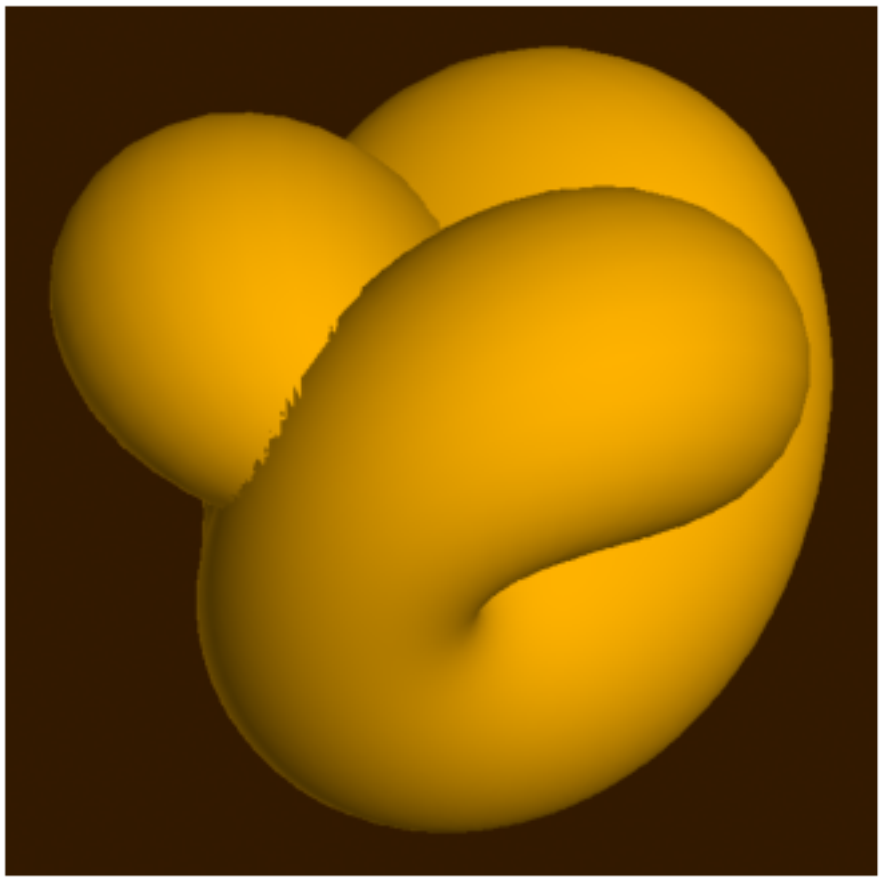
[30]:
```
S = Scene(F_mobius, bornes_mobius, (5, 100), (10, 5, 5), (1, 1, 1), (600, 400))
tracer_gouraud(S)
```



[31]:
```
def F_boy(u, v):
    r5 = math.sqrt(5)
    w = u * (math.cos(v) + 1j * math.sin(v))
    w1 = w ** 6 + r5 * w ** 3 - 1
    g1 = -3/2 * (w * (1 - w ** 4) / w1).imag
    g2 = -3/2 * (w * (1 + w ** 4) / w1).real
    g3 = ((1 + w ** 6) / w1).imag - 1/2
    r = g1 ** 2 + g2 ** 2 + g3 ** 2
    return (g1 / r, g2 / r, g3 / r)

bornes_boy = [0, 1, 0, 2 * math.pi]
```

[32]:
```
S = Scene(F_boy, bornes_boy, (120, 200), (10, 5, 5), (1, 1, 1), (600, 600))
tracer_gouraud(S)
```



1.3 3. Illumination de Phong

Le modèle d'éclairement que nous allons maintenant étudier est utilisé entre autres par les technologies OpenGL et DirectX. La lumière émise par un objet d'une scène peut être décomposée en trois composantes :

- La réflexion ambiante.
- La réflexion diffuse.
- La réflexion spéculaire.

Nous avons déjà utilisé les deux premières composantes dans l'algorithme de Gouraud, mais nous allons y revenir brièvement.

1.3.1 3.1 Réflexion ambiante

Cette composante permet de modéliser ce que l'on appelle la *réflexion indirecte*. Un objet plongé, par exemple, dans l'air, est légèrement éclairé même dans ses parties qui sont à l'ombre. Ce ne serait pas le cas dans le vide. Aucune difficulté ici, il suffit d'attribuer une intensité lumineuse

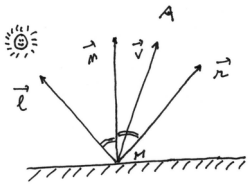
constante à tous les points de la surface que nous désirons tracer. Le composant d'illumination ambiante est précisément

$$I_a = k_a L_a$$

où $k_a \geq 0$ est une constante que nous choisirons plus tard et L_a est un triplet (R, G, B). où $0 \leq R, G, B \leq 1$ sont les quantités de rouge, de vert et de bleu de la couleur.

[33]:
```
def ambiante(M, S, ka, La):
    return mul(ka, La)
```

1.3.2 3.2 Réflexion diffuse



Soit M un point d'une surface. Soit $\overrightarrow{n}$ le vecteur normal à cette surface qui pointe vers le demi-plan où se situe l'observateur Ω.

Notons $\overrightarrow{\ell}$ le vecteur unitaire indiquant la direction de la source lumineuse. La composante d'illumination diffuse est alors

$$I_d = k_d \max(0, < \overrightarrow{n}, \overrightarrow{\ell} >) L_d$$

où $k_d > 0$ est une constante et L_d est un triplet (R, G, B).

[34]:
```
def diffuse(M, S, norm, kd, Ld):
    v = vecteur(M, S.Obs)
    if prod_scal(v, norm) < 0:
        norm = mul(-1, norm)
    k = kd * max(0, prod_scal(norm, S.light))
    return mul(k, Ld)
```

1.3.3 3.3 Réflexion spéculaire

Il existe plusieurs façons de calculer cette composante.

La première méthode est celle retenue par OpenGL. Soit $\overrightarrow{v}$ le vecteur unitaire colinéaire à $\overrightarrow{M\Omega}$. Soit $\overrightarrow{h}$ le vecteur « à mi-chemin » entre $\overrightarrow{\ell}$ et $\overrightarrow{v}$:

$$\overrightarrow{h} = \frac{\overrightarrow{\ell} + \overrightarrow{v}}{||\overrightarrow{\ell} + \overrightarrow{v}||}$$

La composante d'illumination spéculaire est alors

$$I_s = k_s \max(0, < \overrightarrow{n}, \overrightarrow{h} >)^s L_s$$

où $k_s > 0$ est le **coefficient de réflexion spéculaire**, $s > 0$ est la **brillance**, et L_s est un triplet (R, G, B).

[35]:
```
def speculaire(M, S, norm, ks, s, Ls):
    v = normaliser(vecteur(M, S.Obs))
    h = normaliser(add(v, S.light))
    k = ks * abs(prod_scal(norm, h)) ** s
    return mul(k, Ls)
```

La seconde méthode consiste à utiliser l'angle entre le vecteur $\overrightarrow{v}$, défini dans la cellule précédente, et le vecteur $\overrightarrow{r}$ qui est le vecteur symétrique de $\overrightarrow{\ell}$ par rapport à $\overrightarrow{n}$. Ce vecteur est la direction dans laquelle un rayon lumineux issu de la source de lumière se réfléchit, dans le cas où notre surface est parfaitement réfléchissante. Il s'agit de la première loi de Descartes. On a

$$\overrightarrow{r} = 2 < \overrightarrow{n}, \overrightarrow{\ell} > \overrightarrow{n} - \overrightarrow{\ell}$$

La composante d'illumination spéculaire est alors

$$I_s = k_s \max(0, < \overrightarrow{r}, \overrightarrow{v} >)^s L_s$$

Voici la fonction correspondante. De celle ci et de la précédente, choisissez celle qui vous fait plaisir. Mieux encore, essayez les deux et comparez les résultats.

[36]:
```
def speculaire(M, S, norm, ks, s, Ls):
    v = normaliser(vecteur(M, S.Obs))
    r = sub(mul(2 * prod_scal(norm, S.light), norm), S.light)
    k = ks * max(0, prod_scal(r, v)) ** s
    return mul(k, Ls)
```

1.3.4 3.4 Combinons le tout

L'illumination d'un point est finalement gouvernée par

- 4 paramètres réels k_a, k_d, k_s, s.
- 3 paramètres vectoriels (des couleurs) L_a, L_d et L_s.

La fonction `illumination` regroupe en un calcul les illuminations ambiante, diffuse et spéculaire. Elle prend en paramètre une liste `consts` qui contient tous les paramètres d'illumination.

[37]:
```
def illumination(M, S, norm, consts):
    ka, kd, ks, s, La, Ld, Ls = consts
    Ia = ambiante(M, S, ka, La)
    Id = diffuse(M, S, norm, kd, Ld)
```

```
    Is = speculaire(M, S, norm, ks, s, Ls)
    IR, IG, IB = add(add(Ia, Id), Is)
    if IR >= 1: IR = 1
    if IG >= 1: IG = 1
    if IB >= 1: IB = 1
    return (IR, IG, IB)
```

1.4 4. Lissage de Phong

1.4.1 4.1 Interpolation des normales

Au contraire de l'algorithme de Gouraud qui colorie les points d'une face en interpolant les **couleurs** des sommets de la face, l'algorithme de Phong effectue une interpolation des **normales** aux sommets de la face. Une fois cette interpolation effectuée, la couleur est calculée par le modèle d'éclairement avec cette valeur de la normale.

La fonction `tracer_triangle_phong` reprend cette idée. Pour chaque point du triangle à tracer :

- On calcule une normale en ce point par interpolation.
- On calcule également par interpolation un point M qui approche le point de la surface en lequel la surface aurait cette normale.
- On calcule la couleur à affecter à ce point.

Évidemment, l'interpolation de Phong est plus coûteuse en calculs que l'interpolation de Gouraud.

[38]:
```
def tracer_triangle_phong(S, T, consts):
    A = S.mat_proj[T[0][0]][T[0][1]]
    B = S.mat_proj[T[1][0]][T[1][1]]
    C = S.mat_proj[T[2][0]][T[2][1]]
    T1 = [A, B, C]
    MA = S.mat_3d[T[0][0]][T[0][1]]
    MB = S.mat_3d[T[1][0]][T[1][1]]
    MC = S.mat_3d[T[2][0]][T[2][1]]
    imin, imax, jmin, jmax = bornes(T1)
    norm1 = S.mat_norm[T[0][0]][T[0][1]]
    norm2 = S.mat_norm[T[1][0]][T[1][1]]
    norm3 = S.mat_norm[T[2][0]][T[2][1]]
    for i in range(imin, imax + 1):
        for j in range(jmin, jmax + 1):
            a, b, c = coef_barycentre((i, j), T1)
            if a != None and a >= 0 and b >= 0 and c >= 0:
                norm = normaliser(add(mul(a, norm1), add(mul(b, norm2), mul(c,␣
↪norm3))))
                M = add(mul(a, MA), add(mul(b, MB), mul(c, MC)))
                S.canvas[S.nj - j][i] = illumination(M, S, norm, consts)
```

1.4.2 4.2 Tracé

La fonction `tracer_phong` est un quasi copier-coller de la fonction `tracer_gouraud`. La seule modification est l'appel à `tracer_triangle_phong`.

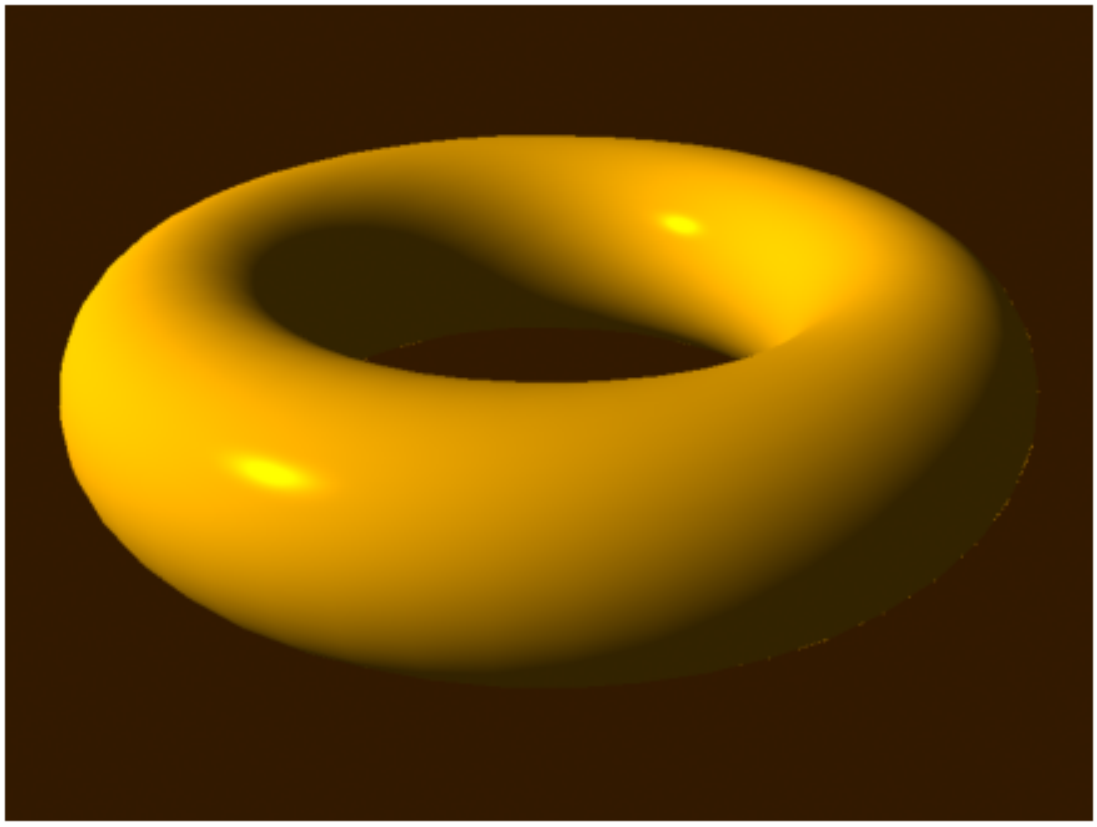
[39]:
```
def tracer_phong(S, consts):
    discretiser_points(S)
    discretiser_normales(S)
    projeter(S)
    mettre_echelle(S)
    fs = faces(S)
    fs.sort(key=lambda T: -distance2(centre_face(S, T), S.Obs))
    for T in fs:
        tracer_triangle_phong(S, T, consts)
    plt.axis('off')
    plt.imshow(S.canvas)
    plt.savefig('image.png')
```

1.4.3 4.3 Exemples

Nous pouvons maintenant reprendre nos exemples.

Tout d'abord, voici le tore.

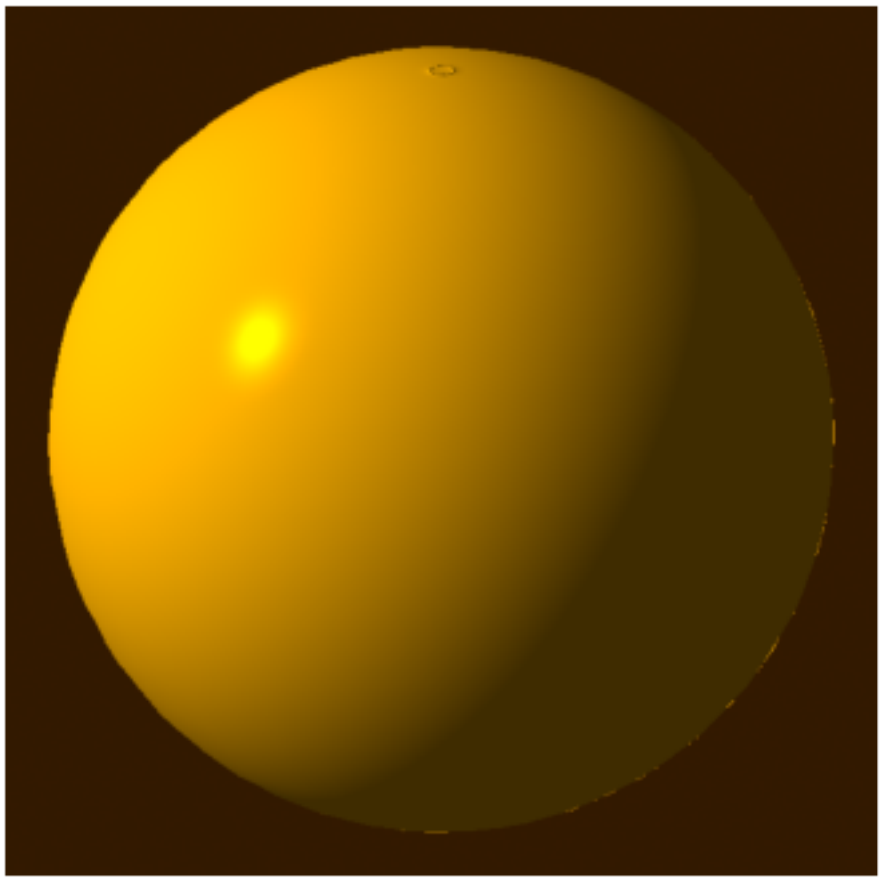
[40]:
```
S = Scene(surf01.F_tore, surf01.bornes_tore, (50, 50), (10, 5, 5), (1, -1, 1),␣
↪(800, 600))
consts = [0.2, 1, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```



Nous avons clairement (?) progressé en passant de l'algorithme de Gouraud à celui de Phong. Je reviendrai à la fin du notebook sur le point d'interrogation.

Voici la sphère.

[41]:
```
S = Scene(surf01.F_sphere, surf01.bornes_sphere, (50, 50), (10, 5, 5), (1, -1,␣
 ↪1), (600, 600))
consts = [0.25, 0.9, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```

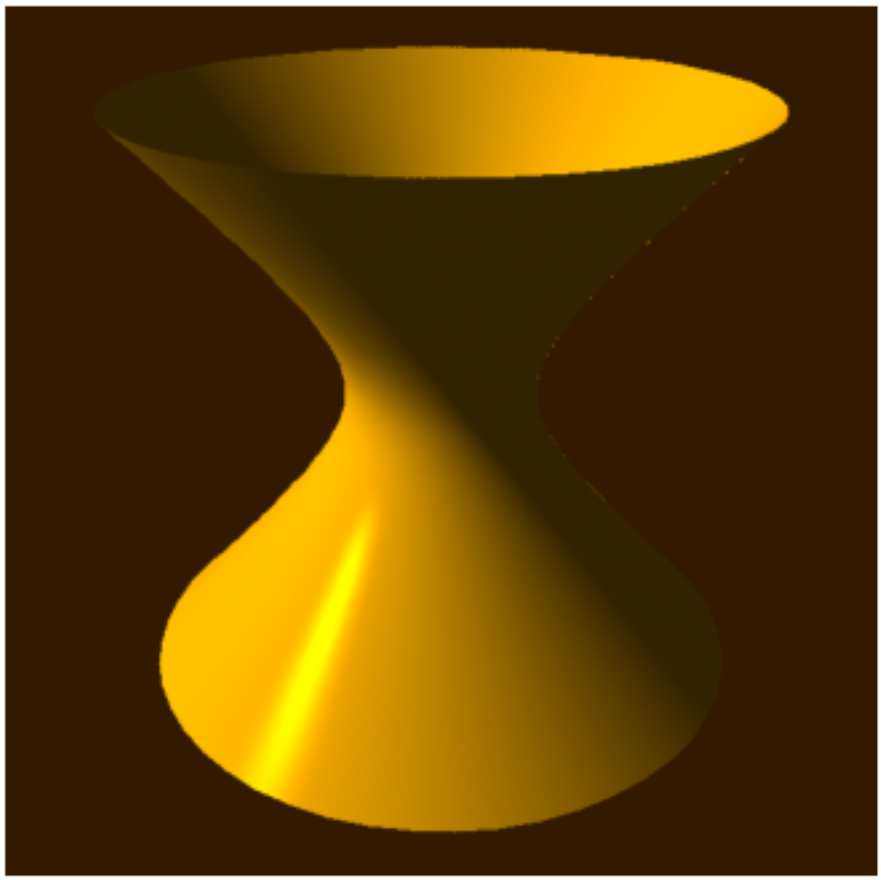


Remarquez un certain nombre de déauts :

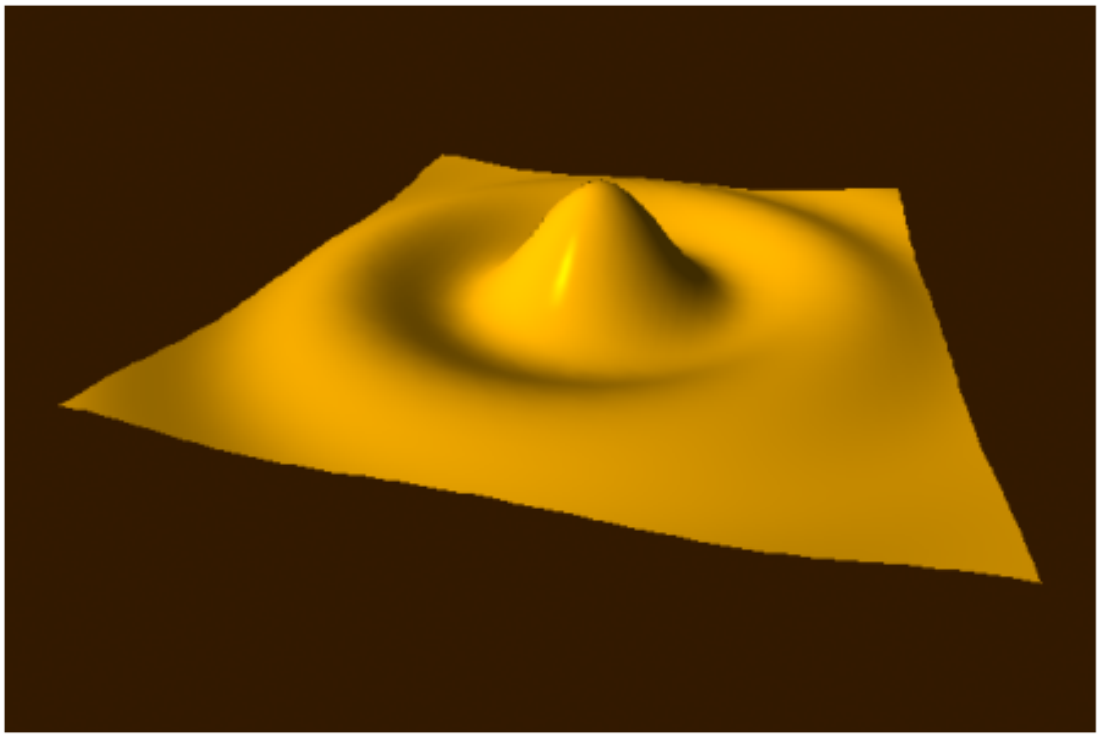
- Quelques points éclairés sur le bord droit de la sphère. Sauriez-vous arranger cela ?
- Un petit défaut au pôle nord, dû au fait que ce point est un point singulier de la surface. Les normales sont mal interpolées à cet endroit.

Sans commentaires, voici les autres exemples.

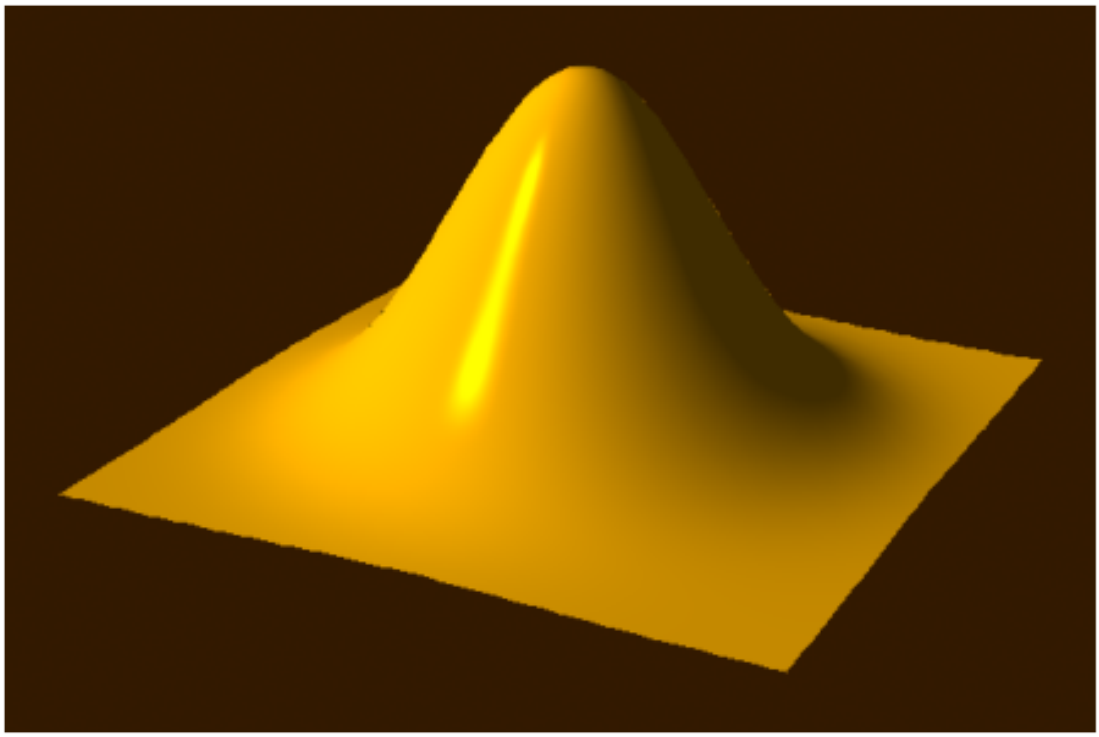
[42]:
```
S = Scene(surf01.F_hyper, surf01.bornes_hyper, (50, 50), (10, 5, 5), (1, -1, 1),␣
↪(600, 600))
consts = [0.2, 0.9, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```



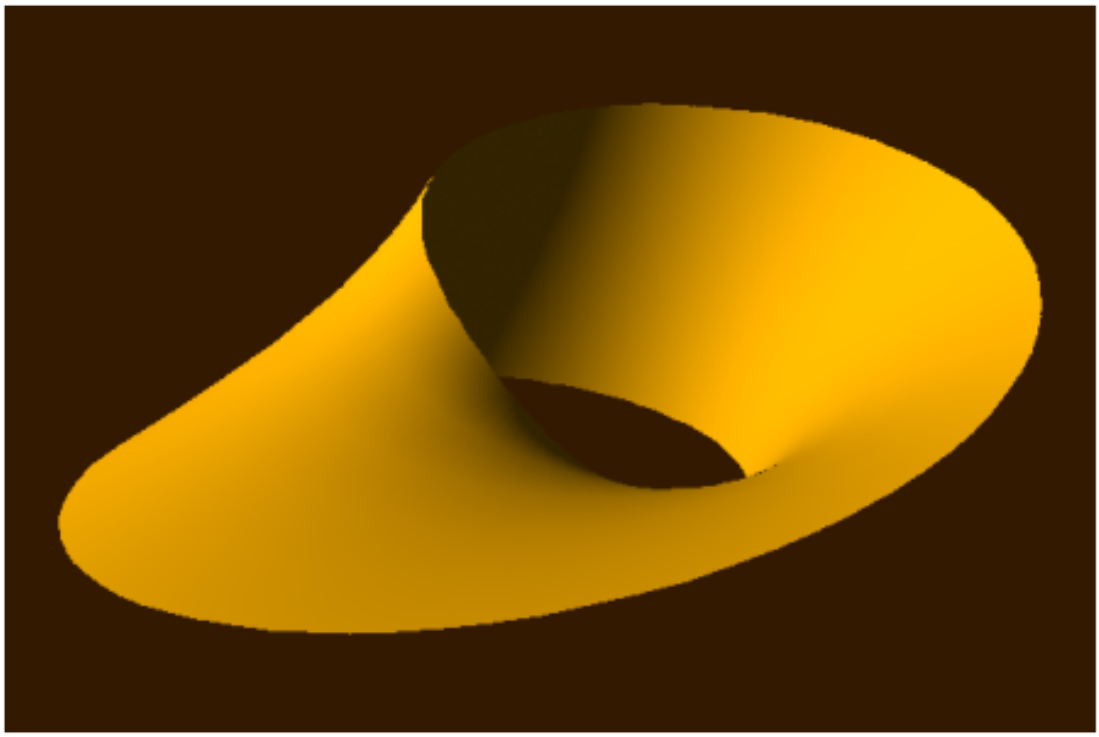
[43]:
```
S = Scene(surf01.F_sin, surf01.bornes_sin, (50, 50), (20, 5, 10), (1, -1, 1),␣
↪(600, 400))
consts = [0.25, 0.9, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```



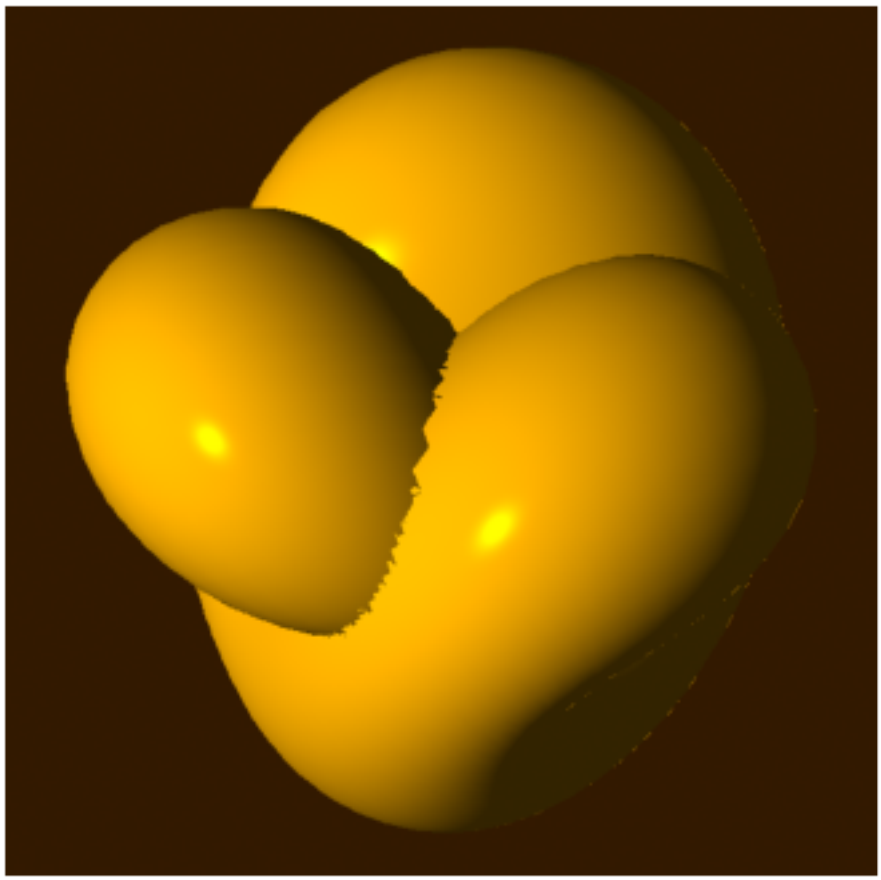
[44]:
```
S = Scene(surf01.F_exp, surf01.bornes_exp, (50, 50), (10, 5, 5), (1, -1, 1),␣
↪(600, 400))
consts = [0.25, 0.9, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```



[45]:
```
S = Scene(F_mobius, bornes_mobius, (5, 100), (10, 5, 5), (1, -1, 1), (600, 400))
consts = [0.2, 0.9, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```



[46]:
```
S = Scene(F_boy, bornes_boy, (120, 200), (20, 3, 20), (1, -1, 1), (600, 600))
consts = [0.2, 0.9, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
tracer_phong(S, consts)
```



1.5 5. Une comparaison entre Gouraud et Phong

Ce que nous avons fait ne permet pas de bien comprendre en quoi l'algorithme de Phong donne un meilleur rendu que l'algorithme de Gouraud. Pourquoi ? Parce que dans la section précédente nous avons fait DEUX améliorations : une pour le lissage et une pour le modèle d'illumination. Reprenons ci-dessous l'algorithme de Gouraud pour qu'il prenne en compte la réflexion spéculaire. Il n'y a que 4 fonctions à légèrement modifier, je vous laisse regarder les détails de celles-ci.

La fonction `tracer_gouraud2` trace la surface décrite par la scène S avec les paramètres d'illumination `consts`.

[47]:
```
def tracer_gouraud2(S, consts):
    discretiser_points(S)
    discretiser_normales(S)
    projeter(S)
    mettre_echelle(S)
    fs = faces(S)
    fs.sort(key=lambda T: -distance2(centre_face(S, T), S.Obs))
    for T in fs:
```

```
            tracer_triangle_gouraud2(S, T, consts)
    plt.axis('off')
    plt.imshow(S.canvas)
```

Ci-dessous, trois fonctions déjà vues plus haut et légèrement modifiées. J'ai modifié les noms en rajoutant un « 2 » à la fin.

[48]:
```
def couleurs_sommets_triangle2(S, T, consts):
    c1 = couleur_sommet2(S, T[0][0], T[0][1], consts)
    c2 = couleur_sommet2(S, T[1][0], T[1][1], consts)
    c3 = couleur_sommet2(S, T[2][0], T[2][1], consts)
    return (c1, c2, c3)
```

[49]:
```
def couleur_sommet2(S, u, v, consts):
    n = S.mat_norm[u][v]
    M = S.mat_3d[u][v]
    c = illumination(M, S, n, consts)
    return c
```

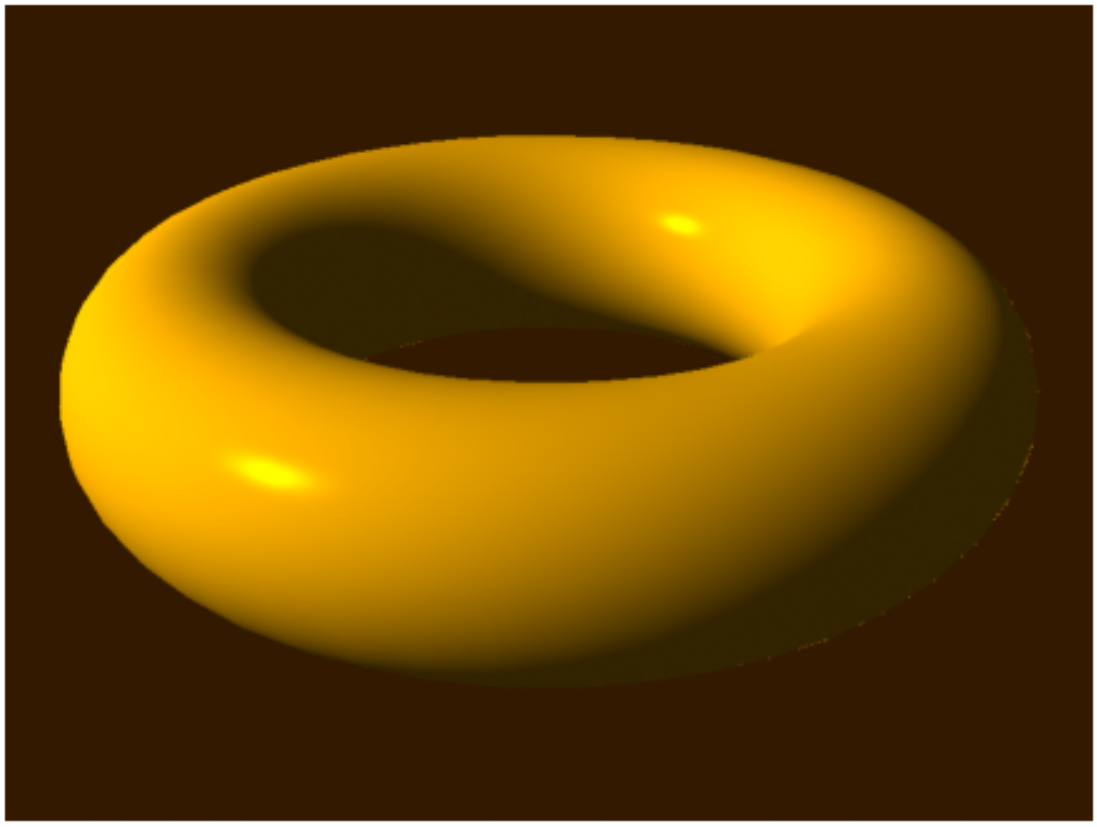
[50]:
```
def tracer_triangle_gouraud2(S, T, consts):
    A = S.mat_proj[T[0][0]][T[0][1]]
    B = S.mat_proj[T[1][0]][T[1][1]]
    C = S.mat_proj[T[2][0]][T[2][1]]
    T1 = [A, B, C]
    imin, imax, jmin, jmax = bornes(T1)
    c1, c2, c3 = couleurs_sommets_triangle2(S, T, consts)
    for i in range(imin, imax + 1):
        for j in range(jmin, jmax + 1):
            a, b, c = coef_barycentre((i, j), T1)
            if a != None and a >= 0 and b >= 0 and c >= 0:
                S.canvas[S.nj - j][i] = add(mul(a, c1), add(mul(b, c2), mul(c,␣
 ↪c3)))
```

Prenons l'exemple du tore, qui illustre parfaitment le problème de l'algorithme de Gouraud. Traçons un tore avec Gouraud et Phong, tous les autres paramères étant identiques.

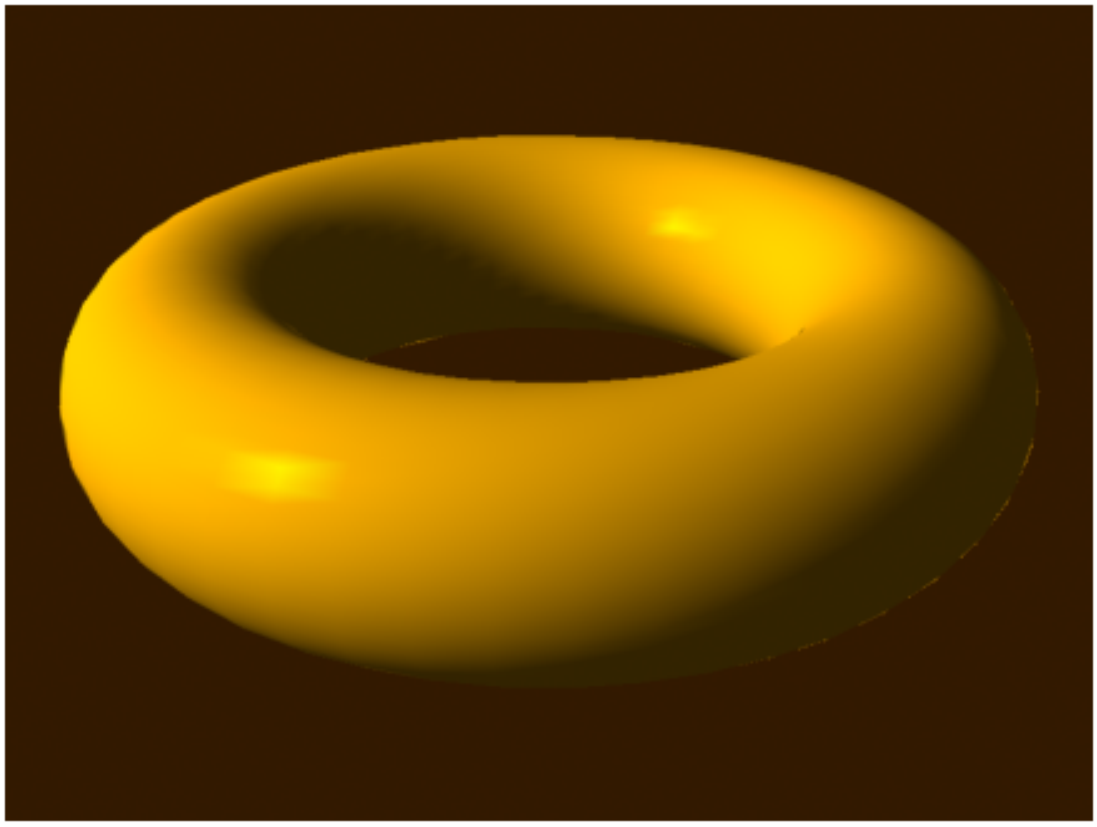
[51]:
```
S = Scene(surf01.F_tore, surf01.bornes_tore, (50, 50), (10, 5, 5), (1, -1, 1),␣
 ↪(800, 600))
consts = [0.2, 1, 0.6, 100, (1, 0.7, 0), (1, 0.7, 0), (1, 0.7, 0)]
```

Petit rappel du rendu avec Phong, puis le résultat avec Gouraud + réflexion spéculaire.

[52]:
```
tracer_phong(S, consts)
```



[53]:
```
tracer_gouraud2(S, consts)
```



L'interpolation des couleurs dans l'algorithme de Gouraud ne permet pas de prendre en compte les parties « brillantes » de la surface. L'algorithme de Phong, en revanche, effectue un rendu plus réaliste. Définitivement, ce dernier algorithme donne de meilleurs résultats pour la réflexion spéculaire.

Vous aurez cependant remarqué que sur l'exemple ci-dessus le tracé avec Phong prend environ deux fois plus de temps que celui avec Gouraud. L'algorithme de Gouraud n'est donc pas à rejeter si la rapidité est à privilégier.

1.6 6. Conclusions

1.6.1 6.1 Améliorations possibles

Voici une liste des améliorations évidentes que l'on pourrait apporter :

- Il est facile de prendre en compte plusieurs sources de lumière. Évidemment, le temps de calcul sera à peu près proportionnel au nombre de sources.
- Plus difficile, mais faisable : on pourrait régler le problème des sufaces qui se « recoupent ». La surface de Boy en est un exemple typique.
- Beaucoup plus ennuyeux est le problème des ombres. L'algorithme du peintre néglige totalement cet aspect : si une face est visible de l'observateur mais qu'elle est cachée de la source

de lumière par une autre face, cette face sera quand même vue éclairée. Ce défaut est difficile à corriger. Une piste possible serait de tout recalculer en plaçant l'observateur à la source lumineuse, et de combiner les deux calculs. Gros, gros travail.

1.6.2 6.2 Que c'est lent !

« Que c'est lent ! On peut faire la même chose en 1 seconde avec Surfplot Pro 4D 14.2 … »

J'entends souvent ce genre de réflexion à propos de certains de mes notebooks. Mon interlocuteur me cite alors telle ou telle merveilleuse application qui fait la même chose en une seconde. Ou alors il me parle des jeux vidéo dernier cri qui font du raytracing en temps réel, etc, etc. Il est absolument nécessaire ici de faire quelques remarques.

- Si vous voulez faire de magnifiques images et que vous n'avez pas envie de savoir comment tout cela fonctionne, utilisez Blender, Yafray, Art of Illusion, POV, etc. Un modeleur professionnel est un programe de plusieurs dizaines de milliers de lignes de programme, développé et amélioré pendant des années. Ce notebook contient environ 500 lignes de code, écrites en quelques heures.
- Le très médiatisé (car très rentable commercialement) raytracing en temps réel est le fruit de **décennies de recherches**, tant du point de vue algorithmique que du point de vue du matériel.
- Rien de ce qui précède n'est « optimisé » (pour autant que ce mot veuille dire quelque chose). Ce notebook **explique** comment tracer des surfaces avec un certain algorithme. Pour que les explications soient possibles, il est nécessaire de **décomposer le code** en petits morceaux facilement compréhensibles, ce qui empêche d'écrire du code plus efficace. Le code écrit est un **code jouet** qui ne cherche absolument pas à être un code professionnel ou commercialement exploitable.
- Le but de ce genre de notebook est de répondre aux deux questions :
  - Comment ça marche ?
  - Pourquoi ça marche ?
- Le choix du langage Python est en partie (mais seulement en partie) la cause de la lenteur. Si vous voulez allez plus vite :
  - Il existe un excellent compilateur Python appelé **Pypy** (`https://www.pypy.org`), qui permet de gagner en gros un facteur 5 sur les temps d'exécution. Et pourtant ce sera toujours du Python.
  - Utilisez `numpy` pour tous les calculs vectoriels et matriciels. Je ne l'ai pas fait afin de montrer ce qu'il est possible de faire avec du **pur Python**, sans aucune boîte noire.
  - Ou alors réécrivez le tout en C. Vous gagnerez un facteur 20 (si votre code est bien écrit) à l'exécution, et mettrez sans doute 5 fois plus de temps pour écrire le code, à quoi il faudra probablement rajouter le même temps pour déboguer le programme.

Bref, sachons rester raisonnables et modestes, et surtout ne perdons pas de vue nos objectifs …