TP1 - Introduction à la modélisation 3D

1 Création d'un maillage triangulaire de sphère 3D

Dans le fichier `tp.cpp`, compléter la fonction `void setUnitSphere(Mesh &o_mesh, int nX, int nY)`, qui créera un maillage triangulaire de sphère 3D.

1. Créer des sommets discrétisant la sphère avec un nombre nX de méridiens et nY de parallèles. Indications : un point 3D sur la sphère peut être obtenu à l'aide de la paramétrisation sphérique en fonction de deux angles $(\theta, \varphi) \in [0, 2\pi] \times [-\pi/2, \pi/2]$:
   - $x = cos(\theta) * cos(\varphi)$
   - $y = sin(\theta) * cos(\varphi)$
   - $z = sin(\varphi)$
2. Générer la topologie : créer la liste des triangles du maillage.
3. Ajouter la fonctionnalité suivante : l'appui de la touche « + »augmente le nombre de méridiens et de parallèles de 1. De la même manière, l'appui sur la touche « - »diminue de 1 le nombre de méridiens et de parallèles.
4. Ajouter les normales aux sommets.

1. Pour créer les sommets, j'ai procédé en trois temps :
   - Création des angles θ selon `nX`,
   - Création des angles φ selon `nY`,
   - Création des points.

   J'ai créé les deux fonctions suivantes pour les deux premiers points :

```cpp
#include <math.h> // < Nécessaire pour M_PI

std::vector<double> initTetas(size_t nX) {
    std::vector<double> tetas(nX);
    for (size_t i = 0; i < nX; ++i)
        tetas[i] = ((2*M_PI)/(nX-1)) * i;
    return tetas;
}

std::vector<double> initPhis(size_t nY) {
    std::vector<double> phis(nY);
    for (size_t i = 0; i < nY; ++i)
        phis[i] = ((M_PI/(nY-1))*i) - (M_PI/2);
    return phis;
}
```

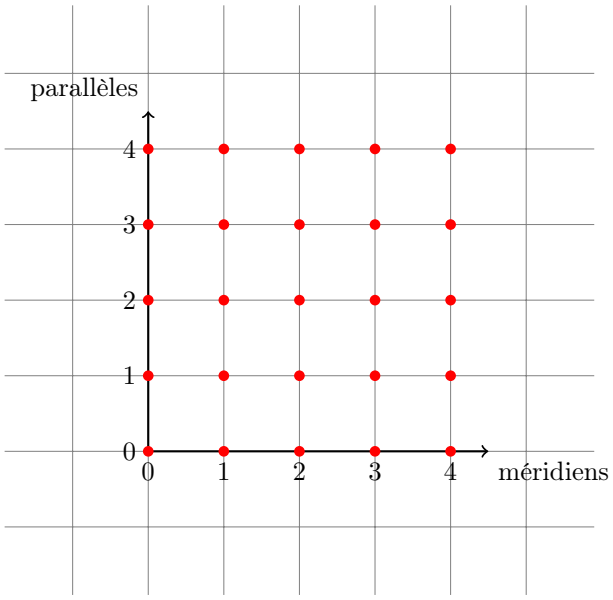
   Une fois ces angles créées, j'ai procédé à la création de tous mes sommets, parallèle par parallèle :

```cpp
double x(double teta, double phi) { return cos(teta) * cos(phi); }
double y(double teta, double phi) { return sin(teta) * cos(phi); }
double z(double phi) { return sin(phi); }
```

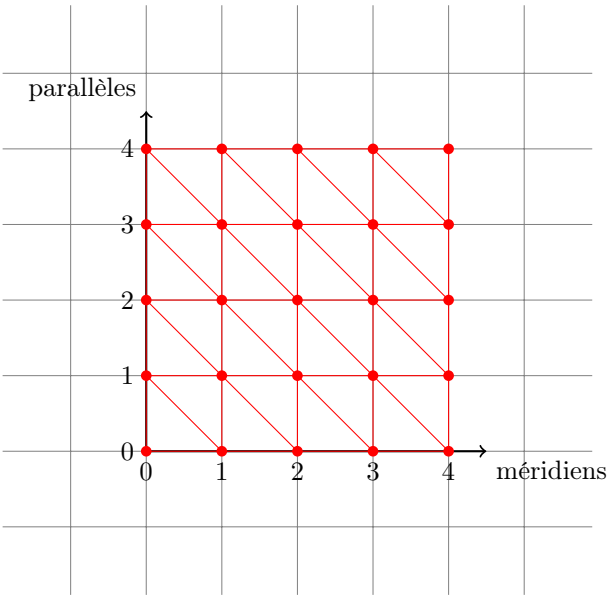
```
void generateVertices(Mesh &o_mesh, size_t nX, size_t nY) {
    const std::vector<double> tetas = initTetas(nX);
    const std::vector<double> phis  = initPhis(nY);

    for (size_t i = 0; i < nX; ++i) {
        for (size_t j = 0; j < nY; ++j) {
            o_mesh.vertices.push_back(Vec3(
                x(tetas[i], phis[j]),
                y(tetas[i], phis[j]),
                z(phis[j])
            ));
        }
    }
}
```

2. Pour générer ma topologie, j'ai simplement vu ma sphère comme une grille : chaque parallèle correspond à un point sur l'axe des ordonnées, et chaque méridien à un point sur l'axe des abscisses :



Le but était de créer des triangles, donc en prenant les 4 points qui forment un carré, on a deux triangles :



Pour ce faire, il suffit donc de prendre les 4 points, en sachant que le premier est à la distance nY du troisième dans le tableau des points, qui sont tous deux à distance 1 de celui juste au dessus :

```
void generateTriangles(Mesh &o_mesh, size_t nX, size_t nY) {
    for (size_t i = 0; i < nX - 1; ++i) {
        for (size_t j = 0; j < nY - 1; ++j) {
            size_t x = (i * nY) + j;
            size_t y = x + 1;
            size_t z = x + nY;
            size_t t = z + 1;

            o_mesh.triangles.push_back(Triangle(x, y, z));
            o_mesh.triangles.push_back(Triangle(z, y, t));
        }
    }
}
```

3. Ici, openGL travaille pour nous : il suffit de modifier la fonction key pour ajouter les cas « + »et « - ». Nous devons cependant instancier deux variables globales pour garder le nombre de parallèles et méridiens à chaque itération. Il ne faut pas oublier de remettre à zéro les vecteurs de notre mesh, pour ne pas créer des choses bizarres :

```
// Variables globales pour les parallèles et méridiens
int g_nX, g_nY;

// Modification de la fonction key :
void key (unsigned char keyPressed, int x, int y) {
    switch (keyPressed) {
    // ...
    case '+' :
        setUnitSphere(unit_sphere, g_nX + 1, g_nY + 1);
        break;

    case '-' :
```

```
        if (!(gnX == 0 || gnY == 0))
            setUnitSphere(unit_sphere, g_nX - 1, g_nY - 1);
        else
            std::cerr << "Vous essayez de supprimer la sphère !" << std::endl;
        break;
    // ...
    }

// Notre fonction :
void setUnitSphere(Mesh &o_mesh, size_t nX = 20, size_t nY = 20) {
    o_mesh.vertices.clear(); o_mesh.triangles.clear(); o_mesh.normals.clear();
    // ...
}
```

4. Ici, on peut tricher. En effet, le cercle est de rayon 1, et les normales sont les vecteurs du centre $O = (0, 0)$ et de chaque point, les normales sont donc directement les coordonnées des points :

```
void generateNormals(Mesh &o_mesh) {
    for (const Vec3& vertex : o_mesh.vertices) {
        o_mesh.normals.push_back(vertex);
    }
}
```

Avec ça, notre fonction setUnitSphere finale ressemble à ceci :

```
void setUnitSphere(Mesh &o_mesh, int nX = 20, int nY = 20) {
    o_mesh.vertices.clear(); o_mesh.triangles.clear(); o_mesh.normals.clear();
    g_nX = nX;
    g_nY = nY;

    generateVertices(o_mesh, nX, nY);
    generateTriangles(o_mesh, nX, nY);
    generateNormals(o_mesh);
}
```