HMIN323: Informatique graphique

Rendu TP9 : Qualité de surface et lissage

Tianning MA

M2 IMAGINA

06/12/2020

Table de matière

1. Introduction

Ce compte rendu est dédié au TP9 lissage.

2. Fonctionnement des touches

Touche	Fonctionnement		
P	Changement PolygonMode	Left	Rotation (y) vers la gauche
SPACE	Switch on/off rotation (y) automatique	Right	Rotation (y) vers la droite

3. Rendu et explication des exercices

3.1 Question 1 Lissage par l'opérateur Laplacien uniforme

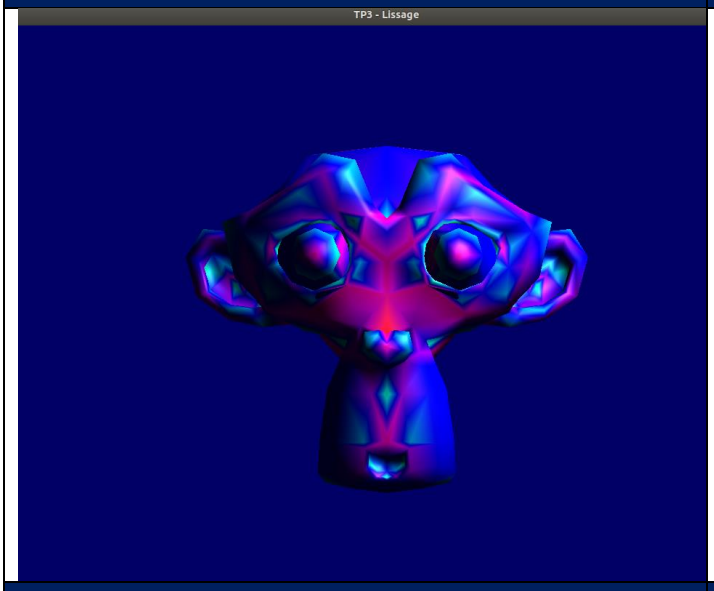
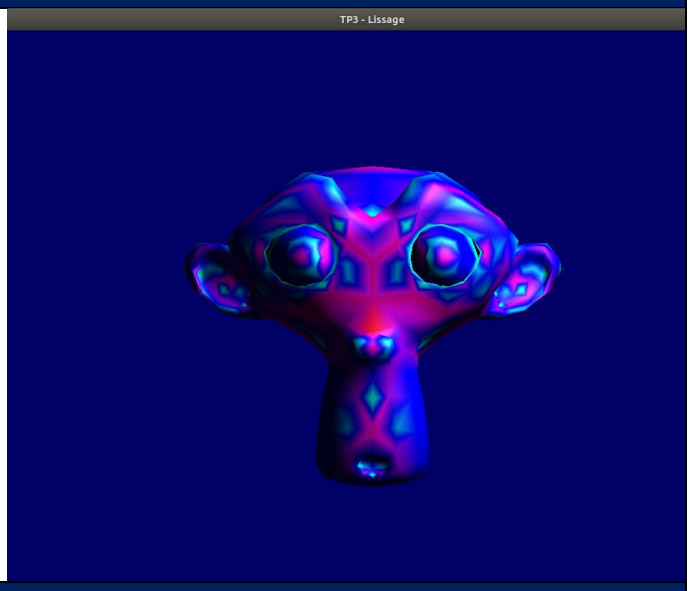
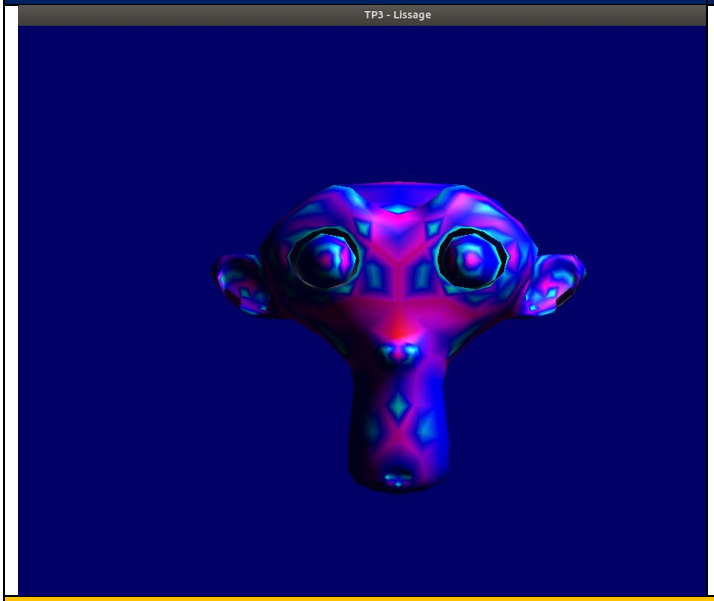
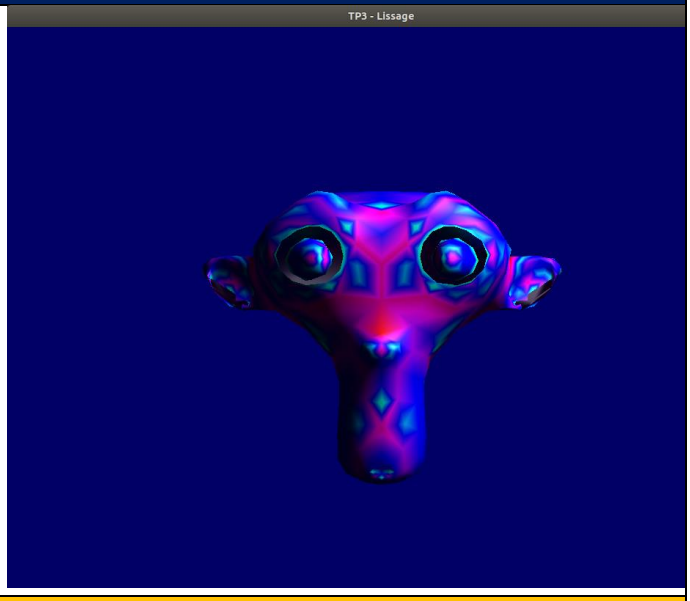
Voici la fonction calc_unjiform_mean_curvature que j'ai implémenté, cette fonction consiste à calculer la courbure moyenne aproximative en utilisant l'opérateur uniforme de Laplace.

```
/****************************** Lissage *********************************/

//Fonction calculer l'opérateur uniforme de laplace
std::vector<glm::vec3> calc_uniform_mean_curvature(const std::vector<glm::vec3> & vertices,
                                                   const std::vector<std::vector<unsigned short>> & triangles){
    std::vector<glm::vec3> result;
    std::vector<std::vector<unsigned short> > one_ring;
    collect_one_ring( vertices, triangles, one_ring );

    for(unsigned int i = 0; i < vertices.size(); i++){
        glm::vec3 Lu(0.0,0.0,0.0);

        for(unsigned int j = 0; j < one_ring[i].size(); j++){
            Lu+=vertices[one_ring[i][j]];
        }
        Lu /=one_ring[i].size();
        Lu -= vertices[i];

        result.push_back(Lu);
    }

    return result;
}
```

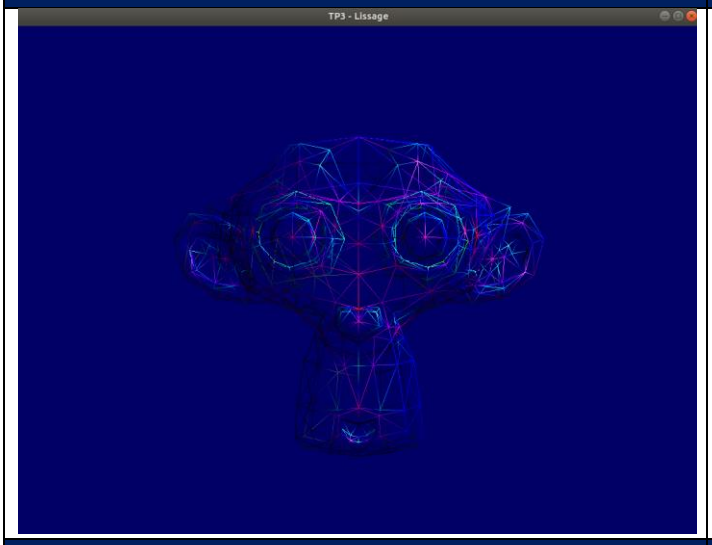
J’ai utilisé d’abord la fonction collect_one_ring pour collecter les informations sur les 1-voisinage, puis calculer le vecteur pour chaque sommet et les sauvegarder dans result (la sortie).

```
//********************** Lissage ******************************/

   for(unsigned int iter = 0; iter < maxIter; iter++){
       std::vector<glm::vec3> uniform_mean_curvature = calc_uniform_mean_curvature(indexed_vertices,triangles);
       std::vector<glm::vec3> indexed_vertices_lisse_uniform;
       for(unsigned int i = 0; i < indexed_vertices.size(); i++){
       uniform_mean_curvature[i].x *= 0.5;
       uniform_mean_curvature[i].y *= 0.5;
       uniform_mean_curvature[i].z *= 0.5;
       indexed_vertices[i] += uniform_mean_curvature[i];
       }
   }
```

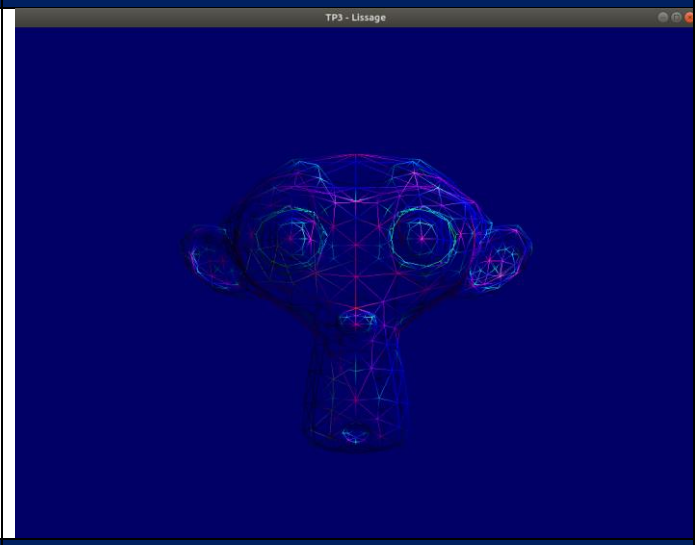
Ensuite, pour lisser le maillage, j’ai fait une boucle pour maxIter itération, à chaque itération, j’ai fait ramener le sommet vers la direction de vecteur calculé 0.5 (poid). Donc le maillage sera lissé et vous trouverez les images résultantes comme ci-dessous.

<table>
<tr><td>Iter = 0</td><td>Iter = 2</td></tr>
<tr><td>  </td><td>  </td></tr>
<tr><td>Iter = 4</td><td>Iter = 6</td></tr>
<tr><td>  </td><td>  </td></tr>
<tr><td colspan="2">PolygonMode(GL_FRONT_AND_BACK, GL_LINE)</td></tr>
</table>

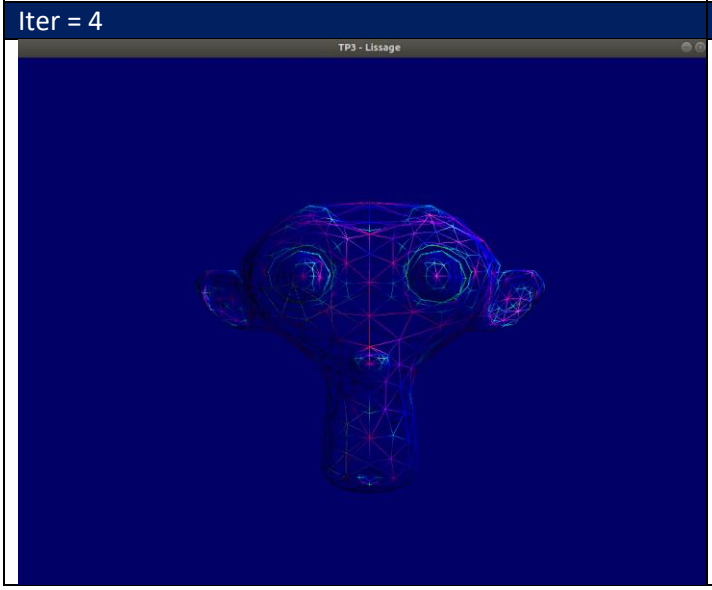
Iter = 0



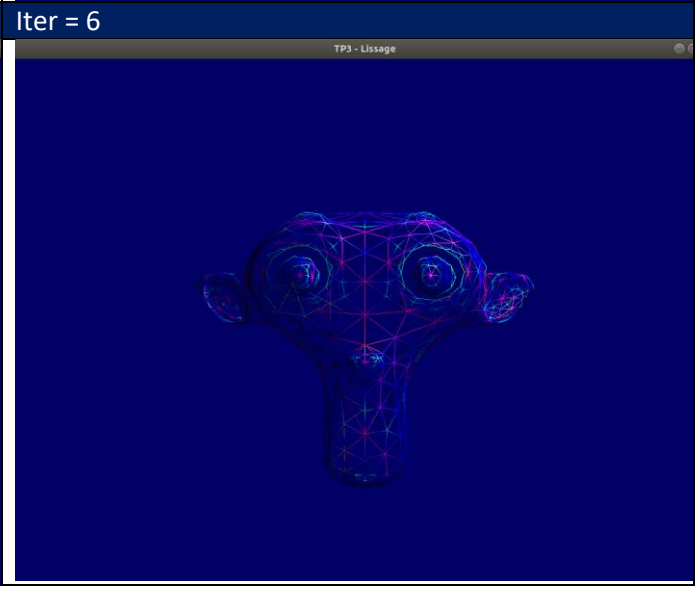
Iter = 2



Iter = 4

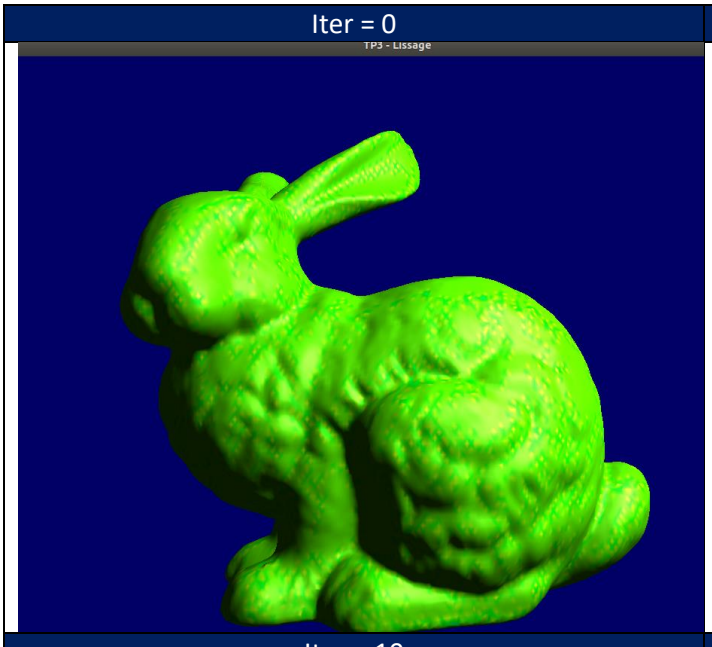


Iter = 6

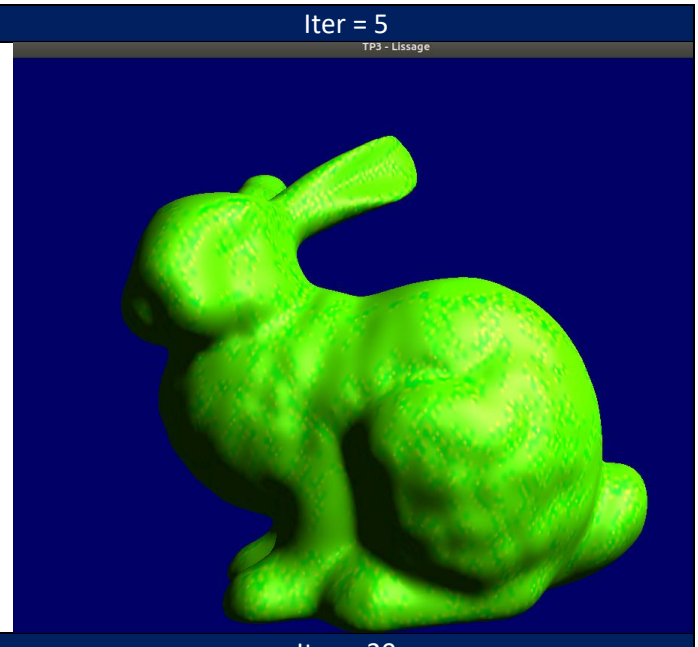


Testes sur un autre exemple :

Iter = 0

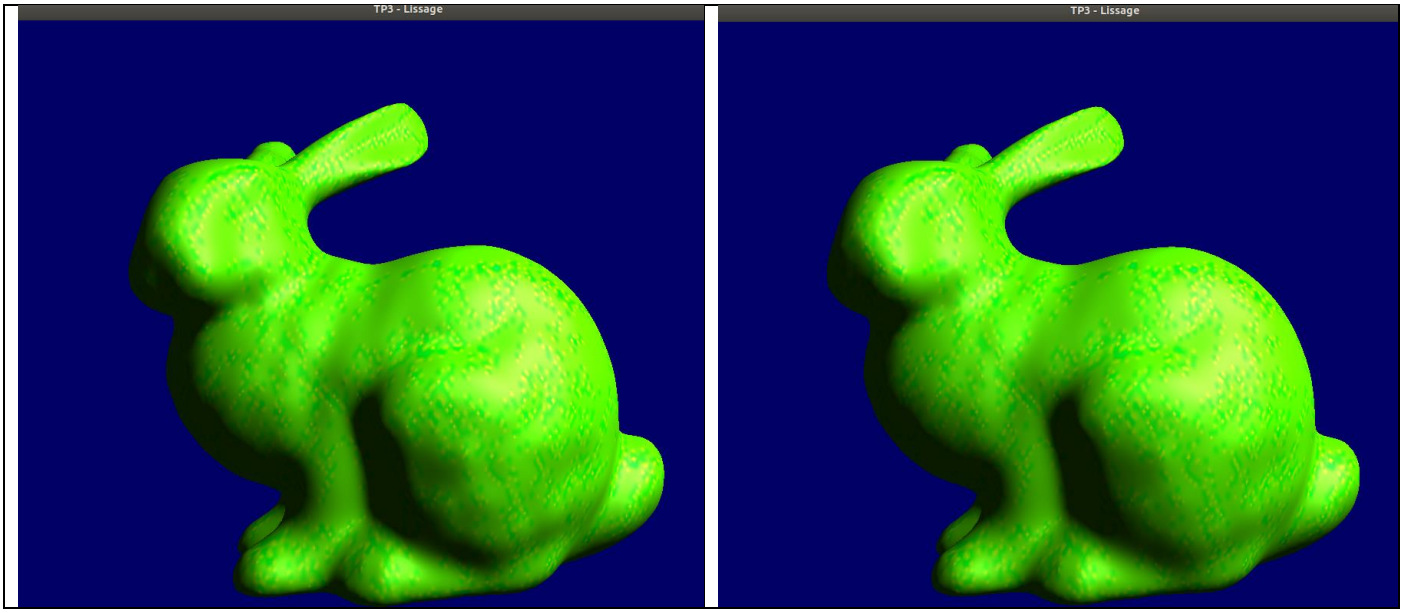


Iter = 5



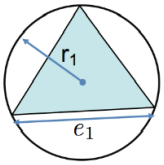
Iter = 10

Iter = 20



3.2 Question 2 Qualité des triangles et Lissage par l'opérateur Laplace-Beltrami

3.2.1 Calcul de la qualité des triangles



La méthode de calcul de la qualité des triangles consiste à comparer le rapport entre le rayon du cercle circonscrit de triangle et la plus petite longueur d'arête.

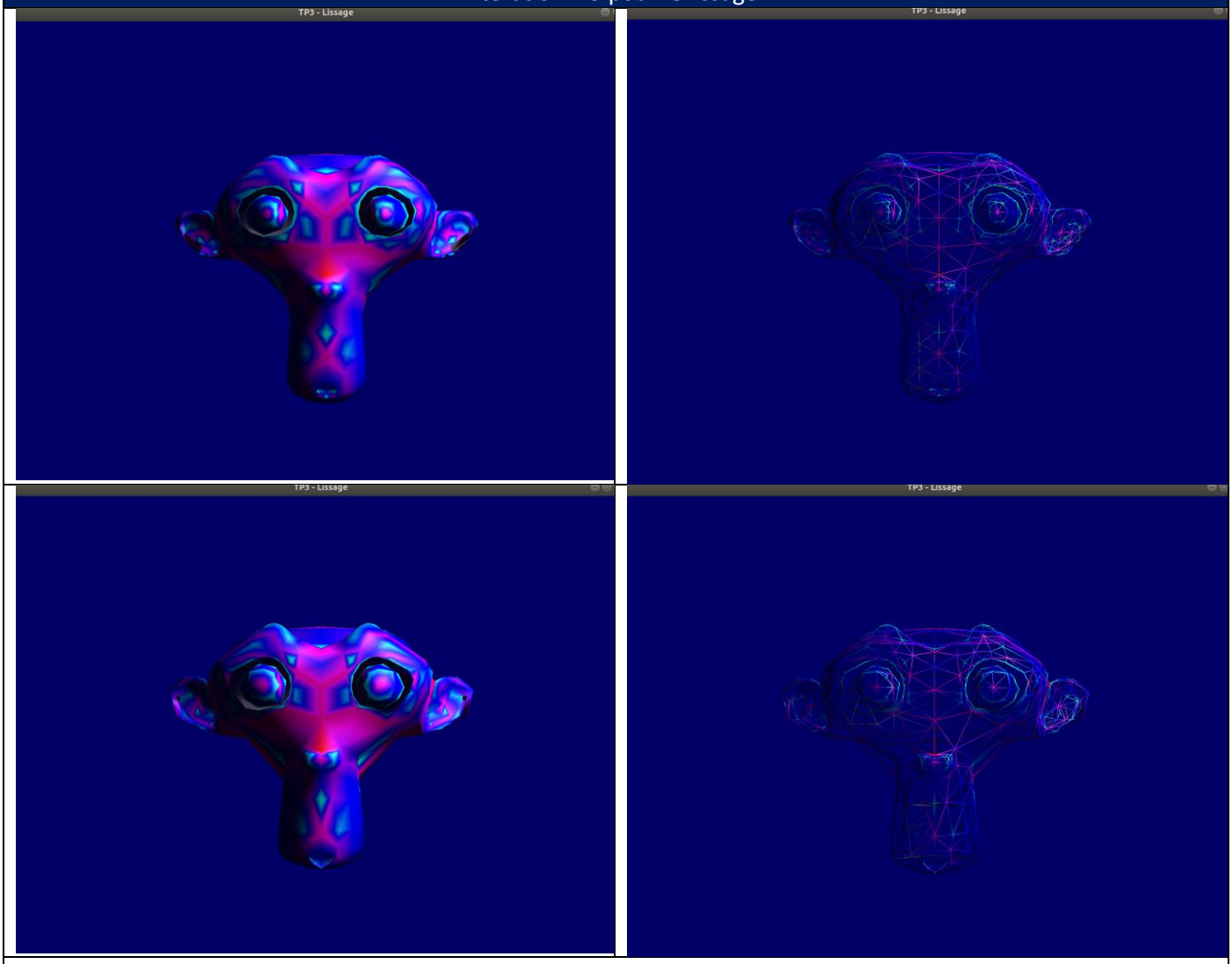
Plus ce rapport est petit, plus les triangles prochent du triangle équilatéral. Donc le maillage est plus idéal.

Mon implémentation de cette méthode est comme -suivante :

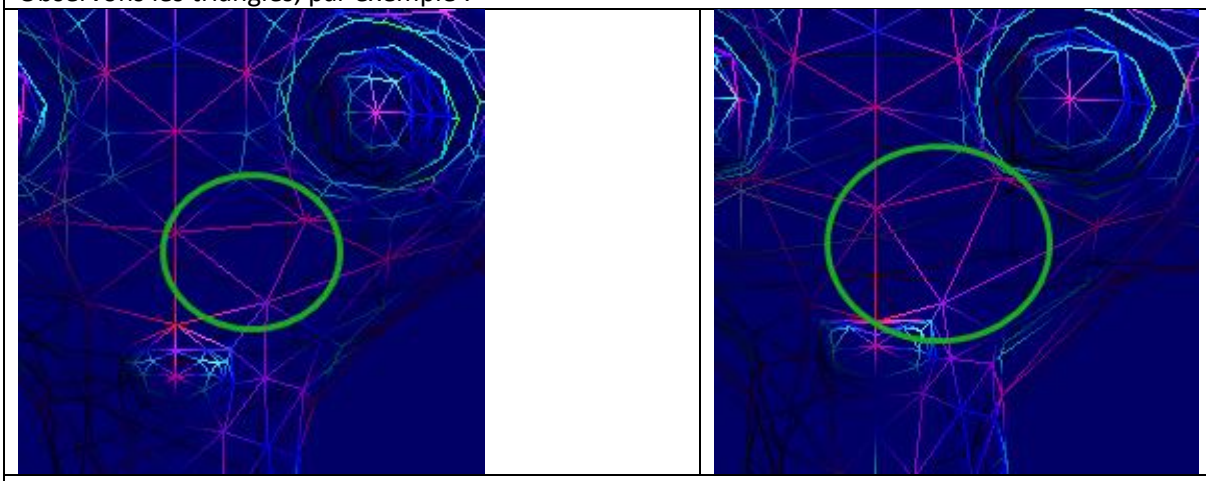
```
float norm(glm::vec3 v){
    float x = v.x * v.x;
    float y = v.y * v.y;
    float z = v.z * v.z;
    return sqrt(x+y+z);
}
float min(float a, float b, float c ){
    if(a <= b && a <= c) return a;
    else if(b < a && b <= c) return b;
    else return c;
}

std::vector<float> calc_triangle_quality(const std::vector<glm::vec3> & vertices,
                                         const std::vector<std::vector<unsigned short>> & triangles){

    std::vector<float> result;
    for(unsigned int i = 0; i < triangles.size(); i++){
        float a(0.0), b(0.0), c(0.0);

        a = norm(vertices[triangles[i][0]]);
        b = norm(vertices[triangles[i][1]]);
        c = norm(vertices[triangles[i][2]]);
        glm::vec3 tmp = glm::cross(vertices[triangles[i][0]], vertices[triangles[i][1]]);

        float nominator = a * b * c * 2.0;
        float denominator = 4 * norm(tmp);
        float r = nominator / denominator;
        //********* Stabilité numérique **********
        //éliminer le cas où le dénominateur est petit ou négatif
        if(denominator <= 0.1){ r = 10;}
        float min_longueur = min(a,b,c);
        float rapport = r / min_longueur;

        result.push_back(rapport);
    }
    return result;
}
```

3.2.2 Opérateur Laplace-Beltrami

Iteration = 5 pour le lissage



Observons les triangles, par exemple :



Observation & Réflexion :
Pour l'itération de lissage = 5, on peut constater que la forme de maillage sur l'opérateur Laplace-Beltrami semble plus proche de maillage d'entrée (donc, la forme semble plus conservée). De plus, concernant la forme des triangles, on s'aperçoit aussi que celui qui a appliqué l'opérateur Laplace-Beltrami a des triangles plus proche des triangles équilatéraux. Donc la qualité des triangles est normalement meillure que la méthode précédente.

Qualité des triangles - Uniform Laplacian

```
Qualité triangle Lissage Laplace-uniforme : 6807.67
Qualité moyenne par triangle Laplace-uniforme : 7.03272
```

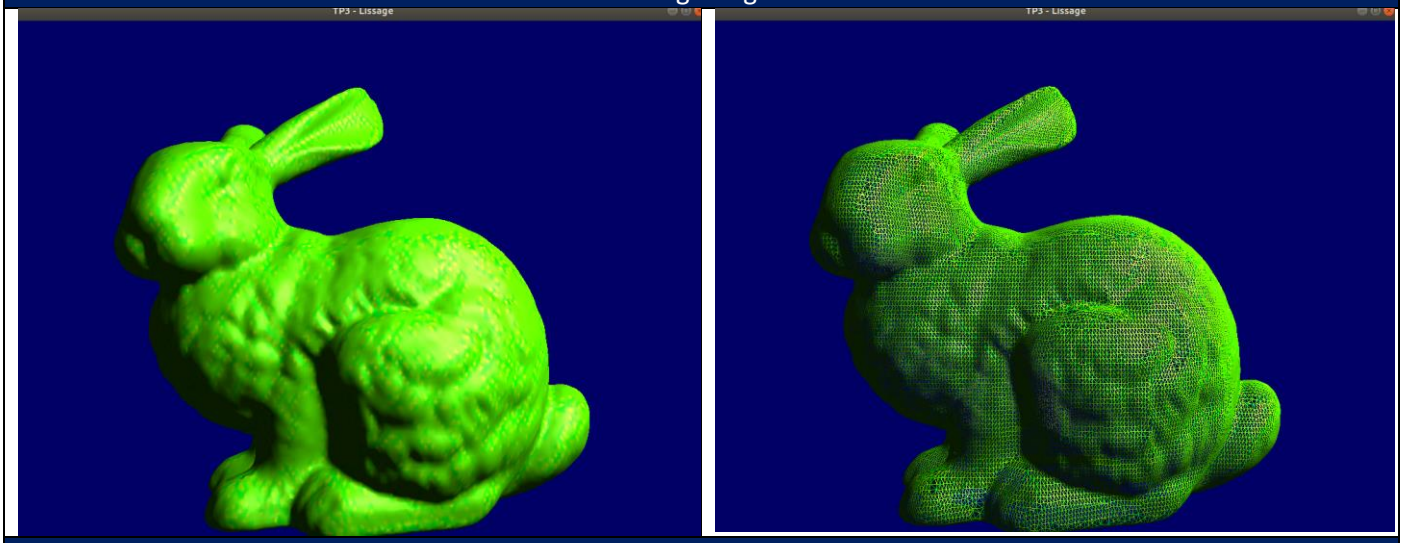
Qualité des triangles – Laplace Beltrami

```
Qualité triangle Lissage Laplace-Beltrami : 6620.8
Qualité moyenne par triangle Lissage Laplace-Beltrami : 6.83966
```

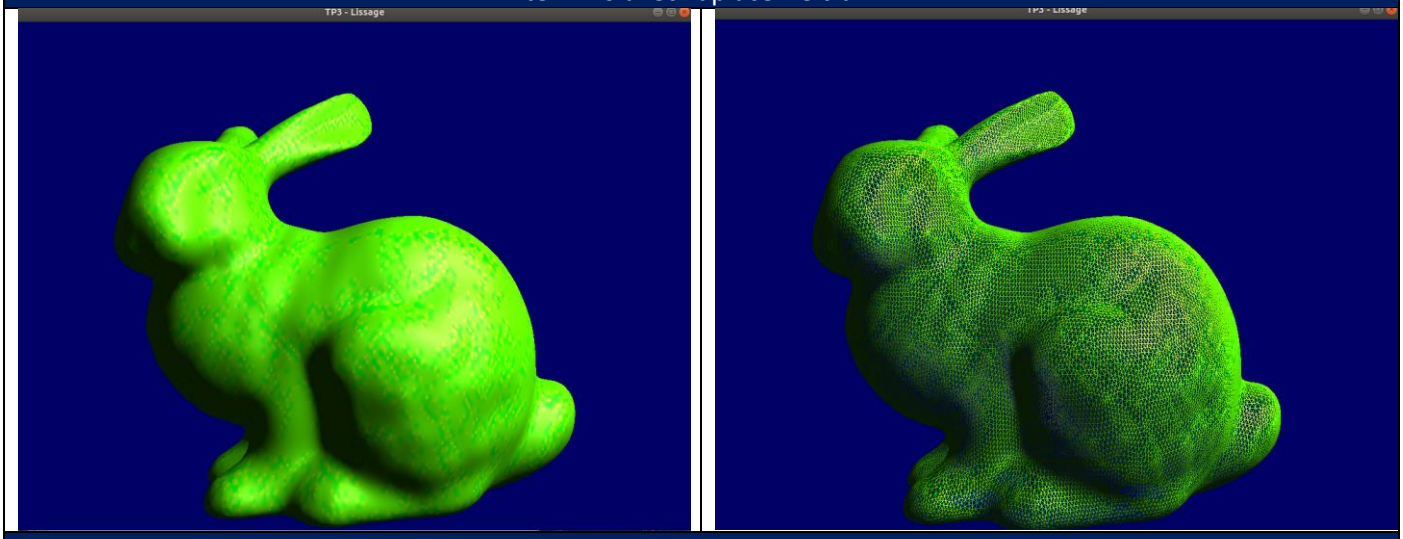
Après avoir utilisé la fonction de calcul de la qualité des triangles, j'ai obétnu la résultat comme ci-dessus.
Ici, j'ai utilisé le rapport entre le rayon du cercle circonscrit de triangle et la plus petite longueur d'arête pour calculer la qualité des triangles. On constate que cette valeur (le rapport) de la méthode LaplaceBeltrami est plus petite que celle de la méthode laplacien uniforme. elle a plus de triangle est proche du triangle équilatéral. Donc le maillage lissé est la meillure qualité.

Testes sur un autre exemple :

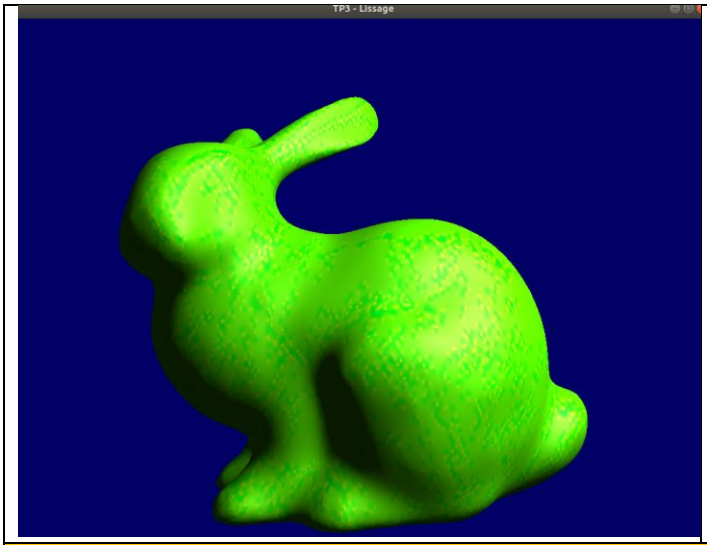
Maillage original

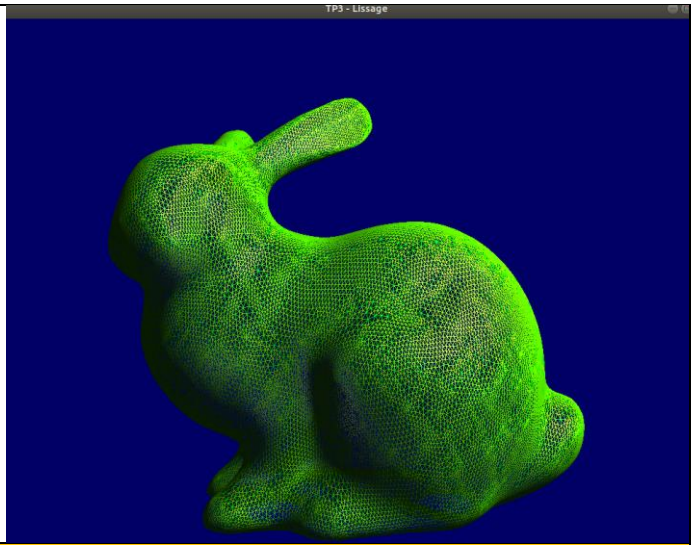


Iter = 20 avec Laplace-Beltrami



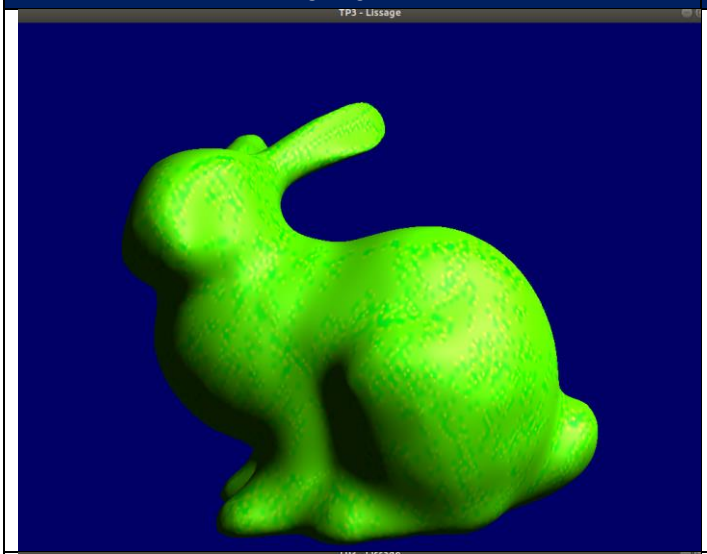
Iter = 50 avec Laplace-Beltrami

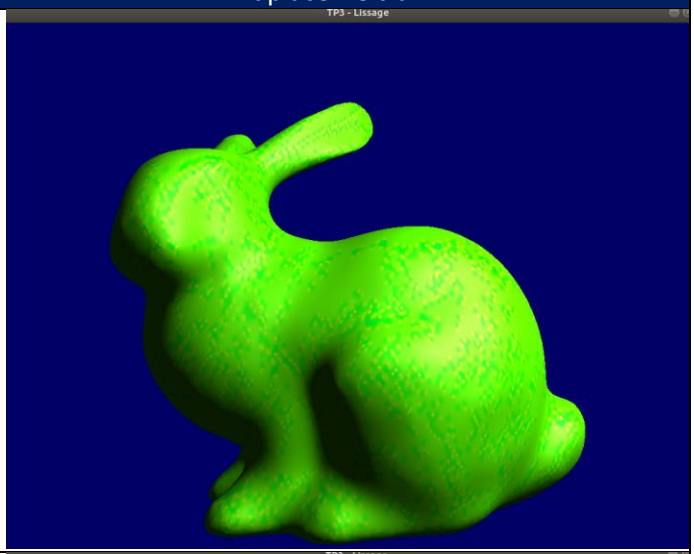


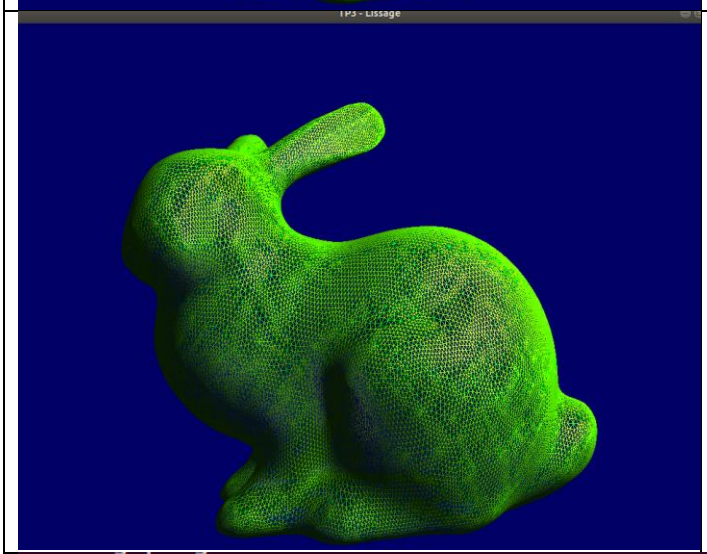


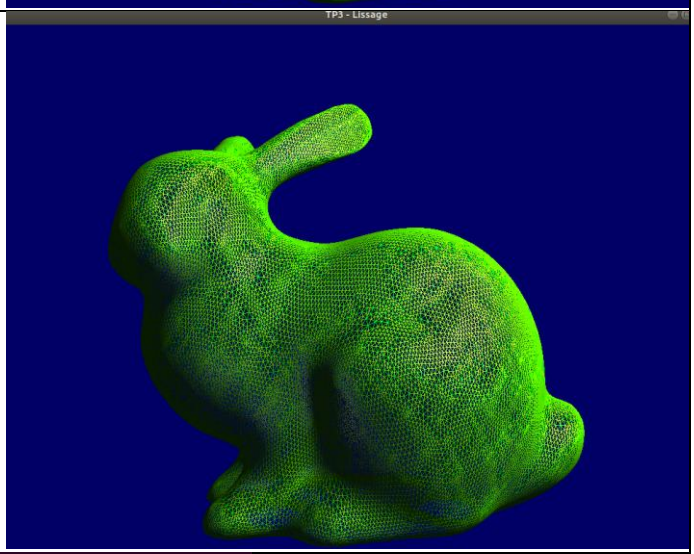
Comparaison avec la méthode Laplacian Uniforme

Uniform	Laplace-Beltrami










```
Qualité triangle Lissage Laplace-uniforme : 7.32231e+06
Qualité moyenne par triangle Laplace-uniforme : 105.106
```

```
Qualité triangle Lissage Laplace-Beltrami : 7.31874e+06
Qualité moyenne par triangle Lissage Laplace-Beltrami : 105.055
```

On constate que visuellement les deux maillages n'ont pas beaucoup de différence, en effet, si on regarde plus précisement, on voit sur la première méthode, les triangles ont l'air déjà assez « idéal » (plus proche de traingle équilitérale). Si on regarde les valeurs d'évaluation par la méthode précédente, on voit que la méthode de laplace-Beltrami a quand même meillure que la méthode laplacien uniforme. (Même si la différence entre les deux est pas énorme).

3.3 Question 3 La courbure gaussienne

Comme vu en cours, la méthode facile pour approcher la courbure gaussienne pour un sommet est d'utiliser le défaut d'angle – la somme des angles autour d'un sommet.

Pour calculer les angles autour d'un sommet, j'ai utilisé la théorème d'Al-Kashi :

$$\alpha = \arccos\left(\frac{\overrightarrow{OA} \cdot \overrightarrow{OB}}{\|\overrightarrow{OA}\| \times \|\overrightarrow{OB}\|}\right).$$

```
/************************Gaussian Curvature*******************************/
std::vector<float> calc_gauss_curvature(const std::vector<glm::vec3> & vertices,
                                        const std::vector<std::vector<unsigned short>> & triangles){
    std::vector<float> result;
    std::vector<std::vector<unsigned short> > one_ring;
    collect_one_ring( vertices, triangles, one_ring );

    for(unsigned int i = 0; i < vertices.size(); i++){
        float angle_total(0.0);
        for(unsigned int j = 0; j < one_ring[i].size() - 1; j++){
            float angle(0.0);

            glm::vec3 OA = vertices[one_ring[i][j]] - vertices[i];
            glm::vec3 OB = vertices[one_ring[i][j+1]] - vertices[i];
            float nominator = glm::dot(OA, OB);
            float denominator = norm(OA) * norm(OB);
            angle = acos(nominator / denominator) * 180.0 / PI;
            std::cout<<angle<<" ";
            angle_total += angle;

        }
        float res = 360.0 - angle_total;
        result.push_back(res);
    }


    return result;
}
```

Pour chaque sommet, on regarde les angles entre ce sommets et ses voisins. On calcule les angles par la théorème d'Al-Kashi, puis on accumule les angles pour tous les voisins. Enfin pour calculer la courbure gaussienne, il faut voir la différence entre 2PI et angle accumulé.

La fonction que j'ai implémenté va sortir un vector de flottant. Donc une collection des valeurs correspondantes aux valeurs de courbure pour chaque sommet.