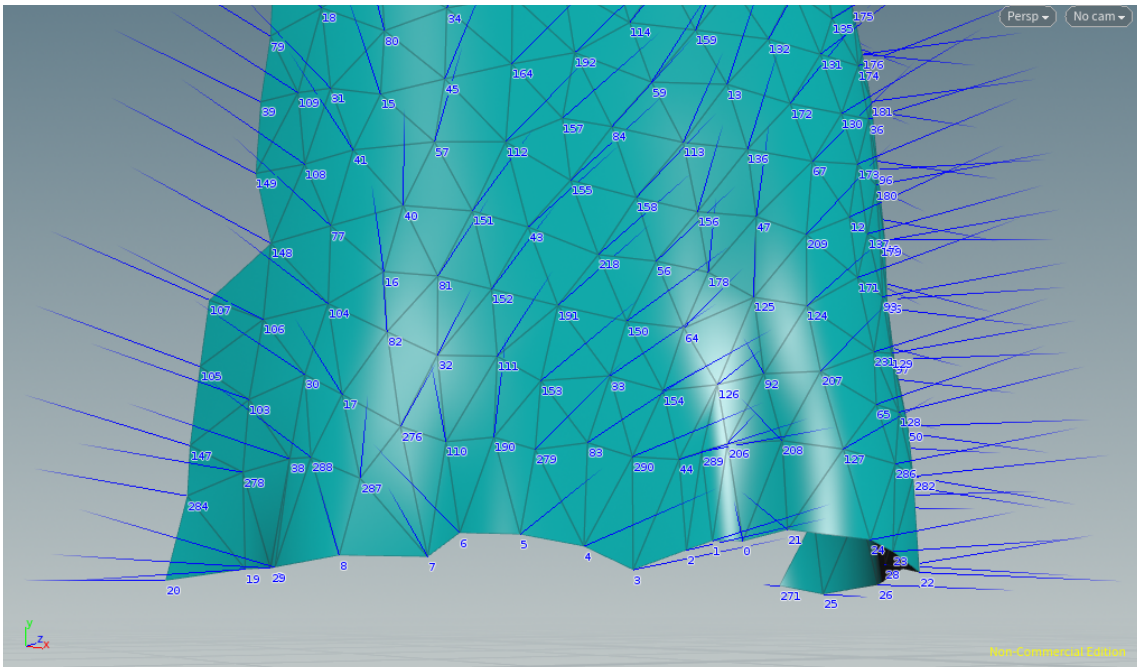
Projet Modélisation Surfacique - Atteindre la developpabilité d'une surface polygonale par déformation minimale

Ahamed Mouhamadou, Jean-Baptiste Gaeng

January 20, 2019



Summary

1 Rappel du problème à résoudre

Notre sujet s'intéresse aux surfaces développables. Plus précisément, on cherche à améliorer la développabilité d'une surface triangulée afin de la transformer en surface quasi développable. Une surface développable est une surface qui peut être dépliée sur un plan sans induire de déformations ni de compressions. Une définition plus précise est qu'une surface développable est caractérisée par une courbure de Gauss nulle en tout sommet de la surface.

2 Implémentation de l'algorithme de l'article

Tout au long du projet, nous avons utilisé l'article "Achieving developability of a polygonal surface by minimum deformation : a study of global and local optimization approaches" de C.L Wang et Kai Thang (2004).

Comme le titre de l'article l'indique, l'article présente deux approches afin de résoudre le problème énoncé plus haut. D'une part, il présente une approche globale consistant à minimiser une fonction d'erreur en considérant toute la surface. D'autre part, comme cette méthode globale est très lente, demande beaucoup de puissance de calcul et n'est pas implémentable pour les cas pratiques usuels, l'article présente une deuxième méthode locale. Cette deuxième approche consiste à reformuler le problème à un problème d'optimisation locale et de mettre à jour itérativement la position de sommets bien choisis afin d'obtenir la surface quasi-développable.

2.1 Choix de la méthode locale contre la méthode globale

Nous avons choisi d'implémenter la méthode locale, pour les raisons citées plus haut. L'algorithme de développabilité locale peut être écrit de cette manière :

1. On calcule les g(q) pour tous les sommets de la grille à optimiser
2. On calcule les vecteurs normaux unitaires pour chaque sommet
3. On crée une file de priorité H qui va contenir les couples (Si , coût(Si)) en ordre croissant des coûts, avec le coût de Si qui vaut $(g(Si))^2$. Le sommet avec le coût max sera placé au sommet de H
4. Initialiser epsilon (seuil de précision), initialiser j à 0 (compteur d'itérations pour la boucle while), initialiser une variable delta à une valeur delta0 choisie arbitrairement et pas trop grande, initialiser un nombre d'itérations max Nmax
5. Boucle while

   Faire

   - Sélectionner le Si au sommet de H
   - Mettre à jour δ : $\delta = \delta - \frac{T(\delta)}{\dot{T}(\delta)}$
   - Déplacer le Si sélectionné précédemment de delta selon la direction de la normale n
   - Actualiser g(qi) et les g(qj) des sommets adjacents car il y a mouvement relatif entre Si et les Sj voisins : on recalcule H et on le trie par ordre croissant des nouveaux coûts

   Tant que [g(Sommet courant) $>$ epsilon (coût trop élevé) ou qu'on a pas atteint le nombre max d'itérations autorisées (j$<$Nmax)
6. Mettre à jour les positions finales de tous les sommets
7. Mettre à jour les vecteurs normaux finaux
8. Retourner la géométrie optimisée

avec :

- g(qi) la fonction de développabilité d'un sommet qi donnée par :

$$g(q_i) = \begin{cases} 2\pi - \sum_k \theta_k & \text{si } q_i \in B \\ 0 & \text{si } q_i \notin B \end{cases}$$

  où les θ_k sont les angles autour du sommet q_i sur les faces autour de ce sommet (voir la figure 3 de l'article), et B les sommets appartenant au contour de la surface

- Comme on l'a dit, le problème global de départ est décomposé selon une combinaison d'optimisations locales sur les sommets triangulaires. Etant donné un sommet q de la surface, on veut trouver une nouvelle position de ce sommet $q* = q + \delta n_q$ avec $g(q*) = 0$. Ainsi, pour ce sommet q, le problème est reformulé par $min(\delta)^2$ sous $T(\delta) = 0$ avec $T(\delta)$ la fonction définie par :

$$T(\delta) = (g(q + \delta n_q))^2 + \sum_j (g(q_j))^2$$

2.2 Bibliothèques utilisées

Pour implémenter l'algorithme, nous avons utilisé la bibliothèque Pymesh de Python. Celle-ci est relativement dure à installer à partir de rien mais une version compilée est disponible au sein d'un Docker, c'est la solution pour laquelle nous avons optée. Après avoir installé Docker sur nos machines, nous avons téléchargé l'image Docker contenant Pymesh disponible à l'adresse suivante : https://hub.docker.com/r/qnzhou/pymesh/

Au sein du Docker contenant la bibliothèque Pymesh, nous utilisons la ligne de commande suivante afin d'importer nos modèles 3D et notre script Python :

```
A la racine du projet :

$ sudo docker run -it --rm -v `pwd`:/models pymesh/pymesh bash

Puis ensuite une fois dans le Docker:

$ python /models/main.py
```

2.3 Visualisation de la courbure de Gausse d'une surface

On rappelle qu'une caractéristique d'une surface dévelopable est que la courbure de Gauss est nulle en tout sommet de cette surface. Ainsi, nous avons voulu pouvoir, pour les différentes surfaces générées par l'algorithme, pouvoir visualiser la courbure de Gauss des surfaces.

Pour ce faire, nous avons utilisé un logiciel appelé Houdini. C'est un logiciel permettant de créer des effets spéciaux mais ce logiciel possède un puissant environnement 3D permettant de créer et de visualiser des surfaces avec beaucoup de fonctions.

Après avoir importé notre surface dans la scène 3D de Houdini, nous avons implémenté le calcul de la courbure moyenne et de la courbure de Gauss. Pour ce faire, on calcule d'abord la matrice Hessienne de la surface 3D. La courbure moyenne est ensuite égale à la moitié de la trace de la matrice Hessienne, tandis que la courbure de Gauss est le déterminant de la matrice Hessienne.

Voici le code en VEX permettant de calculer ces courbures (langage propre au logiciel Houdini) :

```
// compute trace of hessian matrix
float dxx = (normalize(volumegradient(@OpInput1, 0, @P + @dPdx * 0.5)).x - normalize(volumegradient(@OpInput1, 0, @P - @dPdx * 0.5)).x) / length(@dPdx);
float dyy = (normalize(volumegradient(@OpInput1, 0, @P + @dPdy * 0.5)).y - normalize(volumegradient(@OpInput1, 0, @P - @dPdy * 0.5)).y) / length(@dPdy);
```

```
float dzz = (normalize(volumegradient(@OpInput1, 0, @P + @dPdz * 0.5)).z − normalize(
    ↪ volumegradient(@OpInput1, 0, @P − @dPdz * 0.5)).z) / length(@dPdz);

// calculate mean curvature
//@curvature = (dxx + dyy + dzz) * 0.5;

// calculate gaussian curvature
@curvature = (dxx * dyy * dzz);
```

Il est alors possible de visualiser, pour une surface donnée, la courbure de Gauss comme sur la figure suivante :

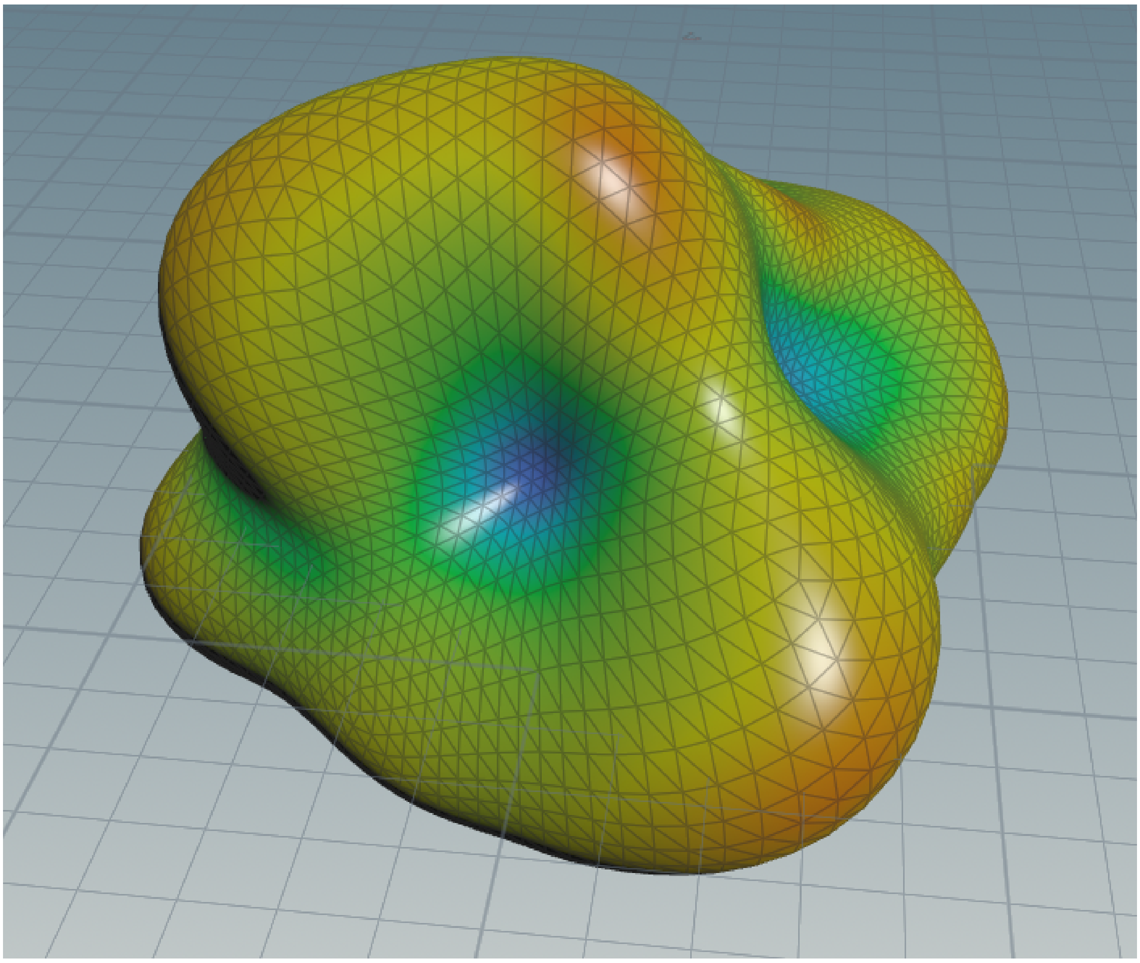


Figure 1: Un exemple de visualisation de la courbure de Gauss pour une sphère déformée. Une rampe de couleur est utilisée pour la visualisation, allant du bleu pour les parties convexes au rouge pour les parties concaves

3 Résultats obtenus

Nous allons expliquer dans cette partie les résultats que l'on obtient après avoir implémenter l'algorithme décrit dans les sections précédentes. Le code source Python est disponible en Annexe.

3.1 Visualisation des coubures de Gauss

A l'heure où ce rapport est écrit, il persiste toujours un petit problème qui nous empêche de bien visualiser la courbure de Gauss pour les surfaces fournies par les professeurs. Les variations d'altitudes sur ces surfaces sont très faibles, et nous n'arrivons pas à trouver une rampe de couleur permettant de visualiser ces petites variations. C'est pour cela que dans la suite, nous avons renvoyé également au cours de l'algorithme l'évolution de la courbure de Gauss pour les sommets dont nous modifions la position, afin de prouver tout de même que notre algorithme fonctionne correctement.

3.2 Un exemple d'exécution de l'algorithme sur une petite surface test

Pendant le projet, afin de débugger notre algorithme et de superviser son fonctionnement, nous avons créer des surfaces très simples sous Maya.
Voici un extrait de la trace obtenue pour une petite surface plane déformée :

```
j = 0
idx_vertex_max before = 5 , g_2_vertex_max before= 0.8135342703763443
Gaussian curvature before = 7.5670141883280815
T(delta) = 0.8420546585973618 , T_point(delta) = 14.86349542474945
delta = -0.056651532566144754
vertex_max position = [-0.166667 0.227113 0.166667]
delta * vertex_max_normal = [0.009013622079433684, -0.05387284064146076, -0.015028898839298895]
new position = [-0.15765338 0.17324016 0.1516381 ]
Gaussian curvature after = 4.108122245385017
idx_vertex_max after = 5 , g_2_vertex_max after = 0.2285460082218915


j = 1
idx_vertex_max before = 5 , g_2_vertex_max before= 0.2285460082218915
Gaussian curvature before = 4.108122245385017
T(delta) = 0.02205632176846878 , T_point(delta) = 1.6407587285265333
delta = -0.07009429010568638
vertex_max position = [-0.15765338 0.17324016 0.1516381 ]
delta * vertex_max_normal = [0.011152450998588057, -0.06665624652485792, -0.018595083795497715]
new position = [-0.14650093 0.10658391 0.13304302]
Gaussian curvature after = 0.8331819398851654
idx_vertex_max after = 6 , g_2_vertex_max after = 0.020111752479892078
```

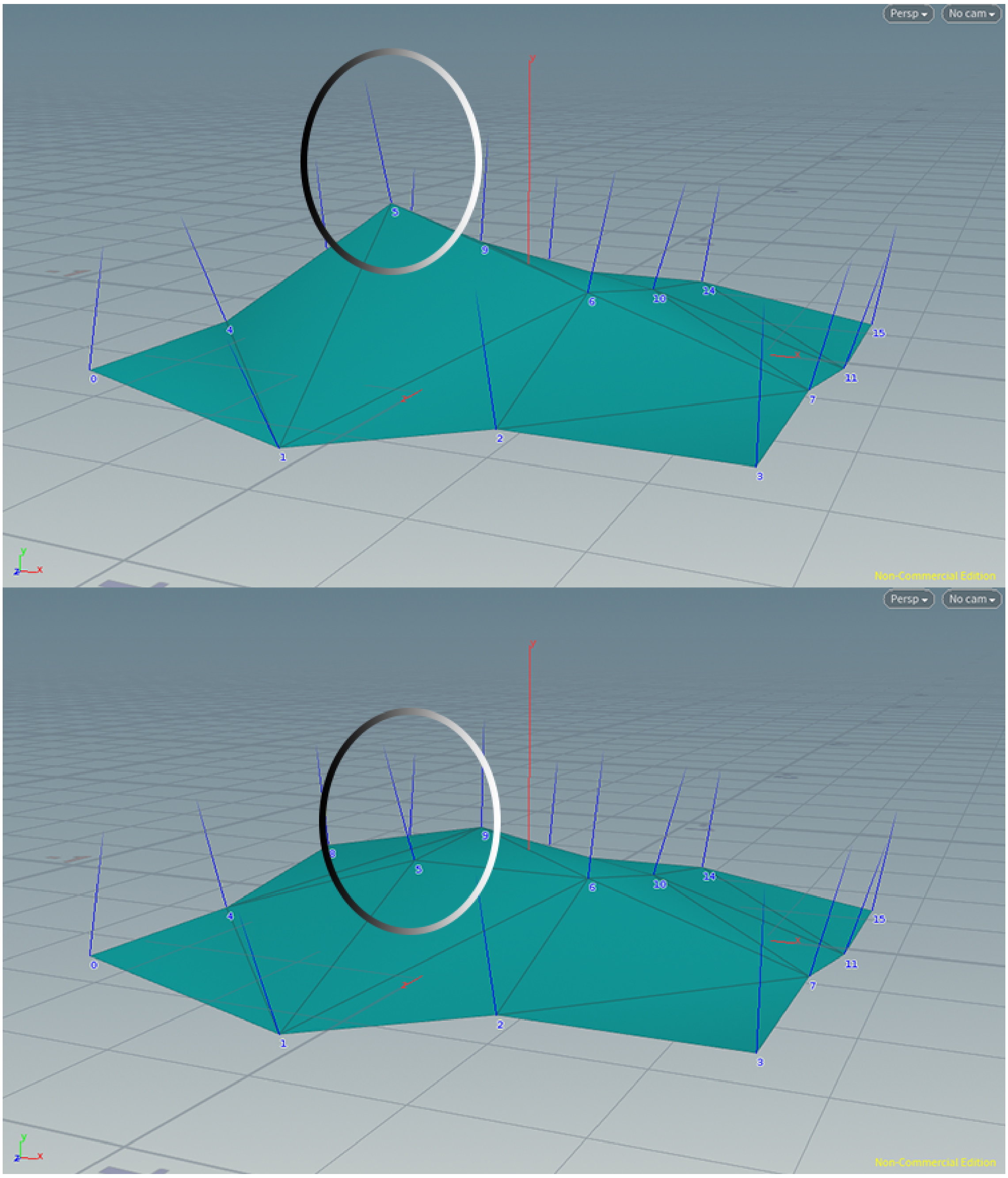


Figure 2: Itération 0 et 1 de la boucle while de l'algorithme pour test-plane.obj. Avec l'aide de la trace, on observe que la courbure de Gauss autour du sommet n°5 diminue - elle passe de -7,50 à l'itération 0 à 0,83 à l'itération 1 - on s'approche donc bien d'une surface developpable.

Dans cet exemple très simple, au début de l'algorithme, c'est le sommet d'index 5 qui a le coût $g(...)^2$ le plus important. Le sommet 5 est bien déplacé le long de sa normale et la courbure de Gauss autour de ce sommet devient quasiment nulle, on s'approche donc bien d'une surface développable. Une fois le sommet 5 déplacé, on atteint les conditions de terminaisons de la boucle while qu'on a fixé et l'algorithme s'arrête. On peut maintenant passer à un exemple plus complexe.

3.3 Un exemple d'exécution de l'algorithme sur une surface de vêtement

Pour ce deuxième exemple, on a utilisé le modèle 3D mesh00040.off du dossier front-dress-anim fourni par les professeurs.

Voici un extrait de la trace renvoyée par l'algorithme pour le modèle de vêtement. Il s'agit des itérations 20 à 22 de la boucle while de l'algorithme :

```
j = 20
idx_vertex_max before = 126 , g_2_vertex_max before= 0.013727014787382922
Gaussian curvature before = -28.488102598576674
T(delta) = 0.03166884408913701 , T_point(delta) = 2.6440287808272824
delta = -0.017842449415774O2
vertex_max position = [0.154739 1.52292 0.2338 ]
delta * vertex_max_normal = [0.007209564461611903, -0.00032921273539923397, -0.016317683671500938]
new position = [0.16194856 1.52259079 0.21748232]
Gaussian curvature after = 74.90545320638711
idx_vertex_max after = 126 , g_2_vertex_max after = 0.10356710370041404


j = 21
idx_vertex_max before = 126 , g_2_vertex_max before= 0.10356710370041404
Gaussian curvature before = 74.90545320638711
T(delta) = 0.9475650909910948 , T_point(delta) = -57.60679191872684
delta = -0.00139360615562608
vertex_max position = [0.16194856 1.52259079 0.21748232]
delta * vertex_max_normal = [0.0005631117779267928, -2.5713560053994926e-05, -0.0012745124775334206]
new position = [0.16251168 1.52256507 0.2162078 ]
Gaussian curvature after = 85.28033917244414
idx_vertex_max after = 126 , g_2_vertex_max after = 0.1353575336856797


j = 22
idx_vertex_max before = 126 , g_2_vertex_max before= 0.1353575336856797
Gaussian curvature before = 85.28033917244414
T(delta) = 0.1983340883418707 , T_point(delta) = -16.119714707514014
delta = 0.010910215093109767
vertex_max position = [0.16251168 1.52256507 0.2162078 ]
delta * vertex_max_normal = [-0.004408469777377465, 0.000201305418942159, 0.009977858674493951]
new position = [0.15810321 1.52276638 0.22618566]
Gaussian curvature after = 10.942948783486555
idx_vertex_max after = 99 , g_2_vertex_max after = 0.010185513415500058
```

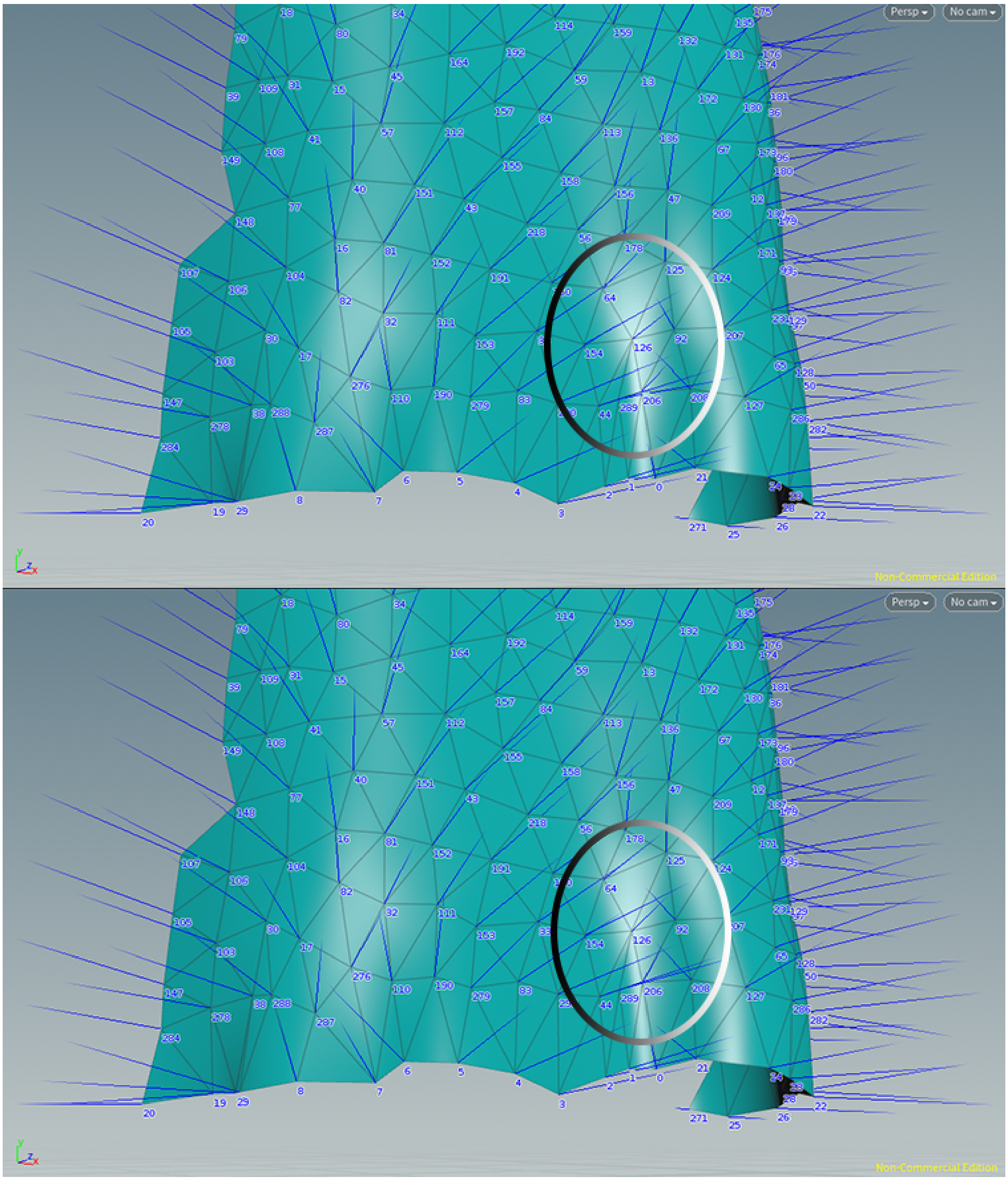


Figure 3: Itération 20 et 22 de la boucle while de l'algorithme pour mesh00040.off. Avec l'aide de la figure précédente, on observe que la courbure de Gauss autour du sommet diminue - elle passe de -28 à l'itération 20 à 10 à l'itération 22 - on s'approche donc bien d'une surface developpable.

Nous pouvons observer sur les figures précédentes qu'entre les itérations 20 à 22, c'est le sommet d'indice 126 qui a le plus grand coût $g(...)^2$. Sa position est modifiée plusieurs fois le temps que sa position converge vers une courbure de Gauss assez faible. C'est ensuite un autre sommet de la surface qui a un coût $g(...)^2$ plus important, et ainsi de suite.

En faisant tourner l'ensemble de l'algorithme sur cet exemple et avec les paramètres du script en annexe, la trace renvoyée par l'algorithme montre qu'à chaque itération de la boucle while, l'algorithme modifie bien la position du sommet dont la fonction $g(...)^2$ est la plus importante. La position de ce sommet varie de delta le long de sa normale tel que définit dans l'algorithme. Pour les sommets dont la position est modifiée, on constate bien que la courbure de Gauss autour de ce sommet s'approche de 0 : la surface s'approche donc d'une surface développable.

Une vidéo du déroulement de l'algorithme au cours du temps est incluse dans le dossier du projet.

4 Améliorations possibles de l'algorithme

Il est encore possible d'améliorer l'algorithme de développabilité locale sur plusieurs points. Tout d'abord, il y a certains cas où la courbure ne Gauss ne s'améliore pas lorsque l'on modifie la position du sommet, c'est à dire qu'au lieu de s'approcher de zéro, la courbure de Gauss s'en éloigne un peu plus qu'initialement. Cela peut venir de problème de convergence lié à la mise à jour de delta au fil de l'algorithme. De plus, l'article que nous avons utilisé demande de "corriger le sommet" lorsque la fonction $T\_point(\delta_0)$ s'approche trop de 0, mais n'est pas clair sur ce qui doit être fait concrètement. Afin de gérer ces cas de divergences, nous avons faits des tests au sein de la boucle while afin d'éviter que le delta ne soit trop grand. Nous supposons qu'il doit être possible de trouver des tests plus efficaces pour gérer ces cas particuliers.

5 ANNEXE : Code source Python

```
#!usr/bin/env python

"""
Developability Algorithm
"""

from math import *
import pymesh
import numpy as np

def g(mesh, idx_vertice, border_vertices_list):
    """
    Compute the vertex developability detect function g(q) at each vertex q on the given mesh
        ↪ patches
    idx_vertice is the index of the vertex q
    """
    mesh.enable_connectivity()
    idx_vertice = int(idx_vertice)
    if idx_vertice not in border_vertices_list:
        list_idx_adjacent_faces = mesh.get_vertex_adjacent_faces(idx_vertice)
        somme_angles = 0
        for idx_face in list_idx_adjacent_faces:
            face = mesh.faces[idx_face]
            if face[0] == idx_vertice:
                edge1 = mesh.vertices[face[1]] - mesh.vertices[face[0]]
                edge2 = mesh.vertices[face[2]] - mesh.vertices[face[0]]
            elif face[1] == idx_vertice:
                edge1 = mesh.vertices[face[0]] - mesh.vertices[face[1]]
                edge2 = mesh.vertices[face[2]] - mesh.vertices[face[1]]
            else:
                edge1 = mesh.vertices[face[0]] - mesh.vertices[face[2]]
                edge2 = mesh.vertices[face[1]] - mesh.vertices[face[2]]
            dot_product = np.dot(edge1, edge2)/(np.linalg.norm(edge1)*np.linalg.norm(edge2))
            if (dot_product > 1):
                dot = 1.0
            elif (dot_product < -1):
                dot = -1.0
            else:
                dot = dot_product
            angle = acos(dot)
            somme_angles += angle
        return(2.0*pi - somme_angles)
    else:
        return(0)


def T(mesh, mesh_normal, border_vertices_list, delta, idx_vertice):
    """
    Compute the function T(delta)
    """
    mesh.enable_connectivity()
    somme = 0
    idx_vertice = int(idx_vertice)
    list_idx_adjacent_vertices = mesh.get_vertex_adjacent_vertices(idx_vertice)
    for qi in list_idx_adjacent_vertices:
        somme += g(mesh, qi, border_vertices_list) ** 2
```

```python

    #We have to copy the vertices to modify it
    vertices_copy = mesh.vertices.copy()
    delta_normal = [delta * mesh_normal[3*idx_vertice], delta * mesh_normal[3*idx_vertice + 1], delta * mesh_normal[3*idx_vertice+2]]
    vertices_copy[idx_vertice] = mesh.vertices[idx_vertice] + delta_normal
    new_mesh = pymesh.form_mesh(vertices_copy, mesh.faces)
    somme += g(new_mesh, idx_vertice, border_vertices_list) **2
    return(somme)

def T_point(mesh, mesh_normal, border_vertices_list, delta, idx_vertice, h):
    """
    Compute the derivative of T with the formula : (T(delta+h) - T(delta))/h
    """
    return (T(mesh, mesh_normal, border_vertices_list, delta + h, idx_vertice)-T(mesh, mesh_normal, border_vertices_list, delta, idx_vertice))/h




def main():
    mesh = pymesh.load_mesh("/models/models/front_dress_anim/meshes/mesh_00040.off")
    # mesh = pymesh.load_mesh("/models/models/test_plane.obj")
    mesh.enable_connectivity()

    N_max = 1000
    epsilon = 0.001

    # 0 Find the edge border of the mesh

    edge_list = []
    border_vertices_list=[]

    for [x,y,z] in mesh.faces:
        edge_list.append((x,y))
        edge_list.append((y,x))
        edge_list.append((y,z))
        edge_list.append((z,y))
        edge_list.append((z,x))
        edge_list.append((x,z))

    compte = {}.fromkeys(set(edge_list),0)
    for value in edge_list:
        compte[value] += 1

    for key in compte.keys():
        if compte.get(key) == 1:
            border_vertices_list.append(key[0])
            border_vertices_list.append(key[1])

    border_vertices_list = set(border_vertices_list)


    # 1 Compute the vertex developability detect function g(q)
    # at each vertex q on the given mesh patches

    list_g = np.array(len(mesh.vertices)*[0.0])

```

```
    for qi in range (len(mesh.vertices)):
        list_g[qi] = g(mesh, qi, border_vertices_list)

    # 2 Compute the unit normal n of each vertex q on O
    mesh.add_attribute("vertex_normal")
    mesh_normal = mesh.get_attribute("vertex_normal")

    # 3 Place all vertices in a maximum heap H keyed on the [g(...)^2]
    # measure - the vertex with the maximum [g(...)^2] is placed at the top
    # of H
    g_2 = []
    for i in range(len(list_g)):
        g_2.append(list_g[i]*list_g[i])

    #4 -> #11
    j = 0
    delta_zero = 0.000001
    delta = delta_zero
    h = 0.01

    g_2_vertex_max = max(g_2)
    idx_vertex_max = g_2.index(g_2_vertex_max)

    while(j< N_max and g_2_vertex_max > epsilon):
        print("j = ", j)

        #Updating the normal vectors
        mesh.add_attribute("vertex_normal")
        mesh_normal = mesh.get_attribute("vertex_normal")

        mesh.add_attribute("vertex_gaussian_curvature")
        mesh_gauss = mesh.get_attribute("vertex_gaussian_curvature")
        print("idx_vertex_max before =", idx_vertex_max, ", g_2_vertex_max before=", g_2_vertex_max)
        print("Gaussian curvature before =", mesh_gauss[idx_vertex_max])

        # Moving vertex_max by delta*n(vertex_max) according to equation 17
        T_delta = T(mesh, mesh_normal, border_vertices_list, delta, idx_vertex_max)
        T_point_delta = T_point(mesh, mesh_normal, border_vertices_list, delta, idx_vertex_max, h)
        print("T(delta) =", T_delta, ", T_point(delta) =", T_point_delta)
        if ((abs(T_point_delta) > 0.1) and (abs(T_delta/T_point_delta) < 0.1)):
            delta = delta - T_delta/T_point_delta
            print("delta =", delta)

            #We have to copy the vertices to modify it
            vertices_copy = mesh.vertices.copy()
            idx_vertex_max = int(idx_vertex_max)
            delta_normal = [delta * mesh_normal[3*idx_vertex_max], delta * mesh_normal[3*idx_vertex_max + 1], delta * mesh_normal[3*idx_vertex_max+2]]
            vertices_copy[idx_vertex_max] = mesh.vertices[idx_vertex_max] + delta_normal
            print("vertex_max position =", mesh.vertices[idx_vertex_max])
            print("delta * vertex_max_normal =", delta_normal)
            print("new position =", vertices_copy[idx_vertex_max])

            #Updating the mesh
            mesh = pymesh.form_mesh(vertices_copy, mesh.faces)
            mesh.enable_connectivity()
```

```python

            #Compute the new g_2 for the vertex and its adjacent vertices
            g_2[idx_vertex_max] = g(mesh, idx_vertex_max, border_vertices_list) **2
            list_idx_adjacent_vertices = mesh.get_vertex_adjacent_vertices(idx_vertex_max )
            for qi in list_idx_adjacent_vertices:
                g_2[qi] = g(mesh, qi, border_vertices_list) ** 2

        else:
            #If T_point is too close to 0, we switch vertex
            g_2[idx_vertex_max] = 0.0

        mesh.add_attribute("vertex_gaussian_curvature")
        mesh_gauss = mesh.get_attribute("vertex_gaussian_curvature")
        print("Gaussian curvature after =", mesh_gauss[idx_vertex_max])

        #Updating the g_2 max and its vertex
        g_2_vertex_max = max(g_2)
        idx_vertex_max = g_2.index(g_2_vertex_max)
        print("idx_vertex_max after =", idx_vertex_max, ", g_2_vertex_max after =", g_2_vertex_max)
        print("\n")

        name = "test_mesh_" + str(j) + ".obj"
        pymesh.save_mesh("./test/" + name, mesh)
        j+=1

if __name__ == "__main__":
    main()
```