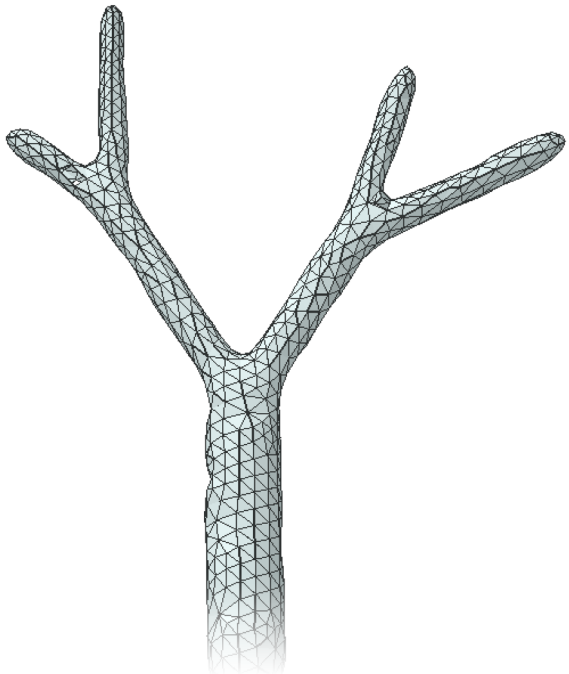
Génération et Maillages Triangulaires d'Arbres par Surfaces Implicites

[HMIN-323] – Mini-Projet

Objectifs et Rendu

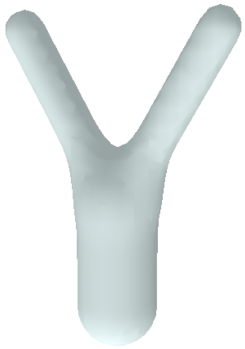
Le but de ce projet est de construire, à partir des différents éléments vus en cours et implémentés en TP, un petit pipeline complet de génération d'objets 3D, en l'occurrence d'arbres. En utilisant essentiellement les fonctions qui étaient demandées lors des TP, plus quelques extensions détaillées dans cet énoncé, il sera possible d'exécuter les différentes étapes du pipeline. Les versions (partiellement) corrigées des fichiers du TP sont mises a disposition sur le moodle de l'UE (projet_template.zip), mais il vous faudra les compléter avec vos propres fonctions codées pour les TP, ainsi qu'avec celles à ajouter au fil de cet énoncé. Un fichier projet.py vous permet de tester les fonctions a modifier pour les différentes questions. Il vous est demandé de rédiger un document au format Word/OpenOffice/PDF détaillant l'exécution de votre code et illustrant les différentes étapes par des screenshots réalisés sur des exemples test (n'hesitez pas a créer vous meme des tests). Le tout sera à envoyer sous forme d'archive zip ou tgz à **frederic.boudon@cirad.fr** avant le 7 janvier 2019. Le projet se compose de trois parties qu'il est possible de traiter plus ou moins indépendamment de sorte que même si tous les composants ne sont pas complets il sera possible de lancer une génération permettant d'évaluer votre implémentation.



PARTIE I:

SURFACE IMPLICITE À PARTIR D'UN SQUELETTE

L'objectif de cette partie est de produire un maillage triangulaire à partir de la définition d'un squelette formé de segments. Pour cela, nous utiliserons une surface implicite, reposant sur des primitives à squelette et des fonctions de potentiel à support compact.



1 Evaluation d'une fonction potentiel d'un squelette

1.1 Fonction potentiel d'un segment

La première étape consiste donc à générer une surface implicite à partir d'un segment de droite. On considèrera qu'un segment est défini par la donnée de deux points $S = (x_S, y_S, z_S)$ et $E = (x_E, y_E, z_E)$. La fonction potentiel elle, aura besoin d'un paramètre de rayon d'influence R. Si l'on utilise la même fonction potentiel vue en TP :

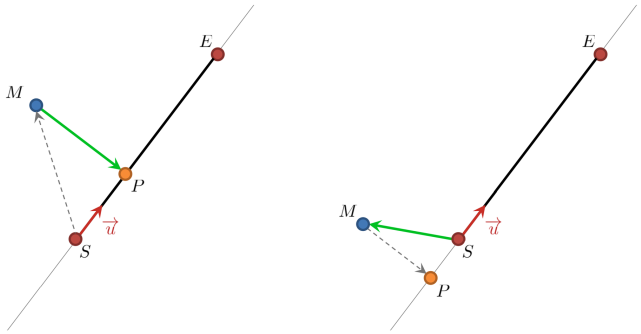
$$f(d) = \frac{1}{R^8}\left(d^2 - R^2\right)^4 \tag{1}$$

le problème revient à savoir calculer la distance d'un point M de l'espace au segment SE. Dans l'espace, la distance à un segment de droite se calcule simplement en utilisant le produit scalaire et la projection sur la droite en question. En notant $\overrightarrow{u} = \frac{\overrightarrow{SE}}{\|SE\|}$ le vecteur directeur unitaire de la droite, le point P, projeté orthogonal de M sur (SE) s'écrit:

$$P = S + (\overrightarrow{SM} \cdot \overrightarrow{u})\overrightarrow{u} \tag{2}$$

La longueur PM donne la distance de M à la droite (SE), mais la distance au segment $[SE]$ est quant à elle donnée par la formule suivante:

$$d(M, [SE]) = \begin{cases} PM & \text{si} \quad 0 < \overrightarrow{SM} \cdot \overrightarrow{u} < SE \\ \min\left(SM, EM\right) & \text{sinon} \end{cases} \tag{3}$$



Question 1

Ecrire la fonction Python retournant un évaluateur de la distance d'un tableau de points donnés par des coordonnées (x,y,z) à un segment défini par les coordonnées de S et E. Pour cela vous pouvez par exemple compléter le canevas de code suivant. A noter qu'il serait preferable de fournir une implementation en utilisant numpy. Des fonction pour calculer la multiplication et la norme d'un ensemble de vecteurs est donné en début de fichier.

```
def segment_distance(start_point, end_point):
    length = np.linalg.norm(end_point-start_point)
    directional_vector = (end_point-start_point)/length

    def distance(points):
        point_vectors = # Vectors from start_point to the points

        start_distances = mnorm(point_vectors) # Distances of points to start_point
        end_distances = # Distances of points to end_point
        distances = np.minimum(start_distances,end_distances)

        dot_products = # point_vectors . directional_vector (use np.dot)
        projected_points = # Coordinates of projections, using dot_products
        line_distances = # Distances of points to projected_points
        segment_points = np.where((0<dot_products) & (dot_products<length))
        distances[segment_points] = line_distances[segment_points]
        return distances

    return distance
```

Question 2

A l'aide de la fonction précédente, et en vous inspirant de la classe **SphericalImplicitSurface** du TP, compléter la classe implémentant une surface implicite par primitive à squelette. Pour cela remplir le constructeur de la classe qui instancie ses attributs, et sa méthode **potential_evaluator** qui renvoie sa fonction évaluateur de potentiel.

```
class SegmentImplicitSurface(ImplicitSurface):
    def __init__(self, start_point=[0,0,0], end_point=[0,0,1], radius=1.):
        super(ImplicitSurface, self).__init__()
        # Instantiate attributes of the class to store the center and
        # necessary parameters (e.g. radius for Stolte).
        self.distance = segment_distance(np.array(start_point), np.array(end_point))
```

```
        self.radius = radius

    def potential_evaluator(self):
```

1.2 Définition d'une classe squelette

Dans la suite, on désignera par **squelette** une collection de segments (définis chacun par les coordonnées de leurs deux extrémités) auxquels un paramètre de taille est associé. En Python, cela s'implémenterait par exemple avec les 2 classes suivantes:

```
class Segment(object):
    def __init__(self, start, end, size):
        self.start = np.array(start)
        self.end = np.array(end)
        self.size = size

class Skeleton(object):
    def __init__(self, segments = []):
        self.segments = list(segments)

    def add_segment(self, p1, p2, size):
        # append a new segment to self.segments
```

Question 3

Compléter la classe **Skeleton** à l'aide de deux méthodes:

- **add_segment** qui prend en argument un couple de points et une taille (dont la valeur par défaut peut être 1) et qui ajoute le segment correspondant aux dictionnaires de l'objet.
- **implicit_surface** qui renvoie la surface implicite de mélange de votre choix obtenue d'après la liste des surfaces implicites de segment générées à partir de chacun des segments du squelette.

Tester l'implémentation de la classe en générant la surface implicite associée au squelette suivant:

$$\mathcal{Y} = \left\{ \left([(0,0,0),(0,0,1)] : 1 \right), \left([(0,0,1),(0,-1,1+\sqrt{3})] : \frac{1}{2} \right), \left([(0,0,1),(0,1,1+\sqrt{3})] : \frac{1}{2} \right) \right\}$$

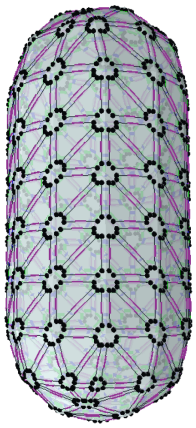
Visualiser le résultat à l'aide des fonctions du TP; vous devriez obtenir une surface similaire à celle donnée au début de cette partie.

2 G-Map triangulaire à partir d'un squelette

Le maillage triangulaire donné par la fonction **surface_mesh** de la classe **ImplicitSurface** n'est généralement pas d'une qualité extraordinaire, et ne se trouve pas représenter de façon à faciliter la manipulation de sa géométrie et de sa topologie. Il serait donc intéressant de le convertir en une structure de données plus adaptée...

Question 4

A l'aide des différentes fonctions vues en TP, créer une fonction prenant en argument un objet de la classe **ImplicitSurface** (ou de l'une de ses classes dérivées) et retournant une instance de la classe **GMap** qui représente le maillage obtenu en par l'application des Marching Cubes. Illustrer le résultat sur un squelette (relativement) simple en utilisant la fonction **dart_display** sur la G-Map ainsi créée.



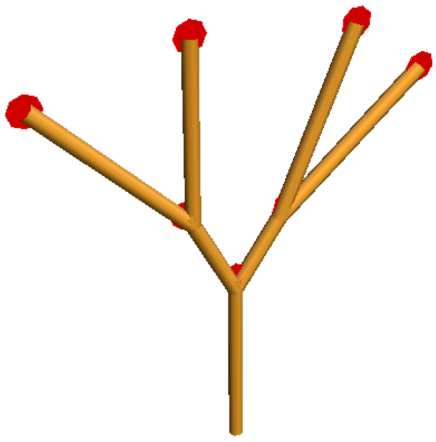
PARTIE II:

GÉNÉRATION PROCÉDURALE DE SQUELETTES D'ARBRES

Dans cette partie, l'idée est finaliser le pipeline en générant les squelettes sur lesquels se basent les surfaces implicites par génération procédurale à partir de L-Systems.

3 L-Systems pour la génération d'arbres

Les L-Systems permettent de générer simplement des structures arborescentes réalistes, avec un petite dose de stochasticité. L'idée ici est de constituer un squelette d'arbre sous forme de L-Systems de façon à pouvoir construire le maillage d'une surface implicite sans avoir à rentrer à la main les coordonnées des points. Cela aura d'ailleurs pour effet de créer des structures moins stéréotypées et présentant des formes plus naturelles.



Question 5

En se basant sur le fichier implicittree.lpy, utilisez L-Py pour générer une structure arborescente de votre choix et en faire le rendu en surface implicite.

L'idée est que pour chaque segment de la structure produite, il faut déterminer les positions début et fin et le rayon de chaque cylindre. Pour cela, vous faut utiliser les modules **`?P(p)`** . Par exemple, un segment sera défini dans votre lsystem par **`S(l,r)`** qui sera décomposé en **`?P(p) _(r) F(l) ?P(p2)`**.

PARTIE III:

OPTIMISATION DE MAILLAGE TRIANGULAIRE

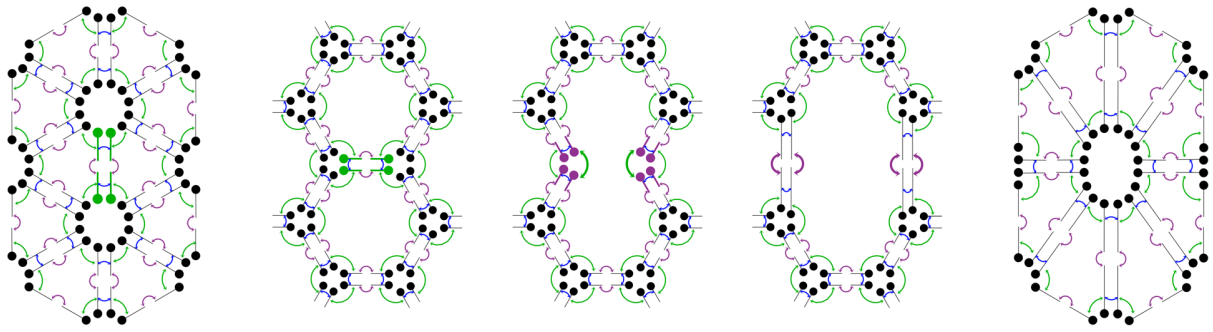
La méthode des Marching Cubes présente l'avantage de pouvoir générer très facilement un maillage triangulaire. En revanche peu de considérations sont accordées à la qualité des éléments de ce maillage, et la surface présentera bien souvent, du fait même de l'algorithme employé, des arêtes très courtes ou des triangles très allongés. Par ailleurs, une bonne résolution de la grille de discrétisation employée est généralement nécessaire pour obtenir une surface correcte, ce qui a tendance à produire des maillages très volumineux. Pour améliorer la situation, il va bien souvent être nécessaire d'optimiser le maillage.

4 Décimation de maillage : une implémentation en G-Maps

La première étape permettant à la fois de se débarrasser des éléments trop petits et de réduire la taille du maillage consiste à effectuer une décimation (respectant autant que possible la surface modélisée). Dans l'esprit de ce qui a été fait en TP, l'objectif est de réaliser une décimation itérative par une série d'opération topologiques unitaires, qui dans le cas de la décimation seront des effondrements d'arêtes ou "edge collapses".

4.1 Edge collapse et dualité

L'opération de collapse évoquée en cours n'est pas forcément évidente à implémenter, que ce soit en GMap ou dans une autre représentation, d'autant qu'elle impose des contraintes topologiques pour pouvoir être appliquée. En revanche, en passant par le dual, l'opération topologique peut devenir plus facile à réaliser.



En effet, dans un cas relativement régulier, le collapse d'une arête d'un maillage triangulaire va correspondre à deux transformations successives très simples du maillage dual:

- La suppression de l'arête (1-cellule) duale de celle que l'on souhaite effondrer
- La suppression des 2 sommets (0-cellule) qui étaient incidents à l'arête duale

Il y a bien sûr toujours des contraintes topologiques à respecter, mais la simplicité vient du fait que dans les deux cas, il est fait appel à la même opération : la suppression d'une cellule.

Question 6

Implémenter la suppression de cellule dans la classe **`GMap`** utilisée jusqu'ici. Pour cela il faudra compléter les trois fonctions suivantes : la suppression d'un brin, le test si une cellule est supprimable, et la suppression d'une cellule. Tester la suppression d'arêtes ou/puis de sommets sur le cube.

```
class GMap: (continued)

    def remove_dart(self, dart):
        """
        Remove a dart from the structure by removing it from
        the alphas dictionaries (keys and values) and transferring
        its position to another embedding dart if necessary.
        """

        # If dart is a key of the positions dictionary:
            # Store the position
            # Remove the dictionary item
            # Find another dart in the vertex orbit (if any)
            # Assign it the position

        # For each value i of alpha:
            # Set alpha_i(alpha_i(dart)) to itself
            # Remove the alphas[i] dictionary item


    def is_removable_cell(self, dart, degree):
        """
        Check whether the cell of a given degree represented by
        the specified dart can be safely removed from the object.
        """

        # Should return True if:
        # - degree is 1 or
        # - degree is 0 and all d in the vertex orbit verify:
        #    alpha_1(alpha_2(d)) = alpha_2(alpha_1(d))


    def remove_cell(self, dart, degree):
        """
        Remove all the darts forming the cell of a given degree
        represented bythe specified dart, if the removal can
        be performed.
        """

        # Verify that the cell can be removed

        # Re-link the opposite darts of the cell orbit
        # For each dart d of the cell orbit:
            # Compute n = alpha_degree(d)
            # If n is not in the orbit:
                # Compute n' = alpha_degree(alpha_degree+1(n))
                # While n' is in the orbit:
                    # n' <- alpha_degree(alpha_degree+1(n'))
                # Link n and n' by alpha_degree

         # Remove all darts in the cell orbit
         # Return the list of removed darts
```

A présent, l'idée est d'utiliser cette fonction **`remove_cell`** pour réaliser l'opération de collapse en passant par le dual du maillage d'origine. **Attention**, ici c'est uniquement la topologie du dual qui est utilisée, il faut bien veiller à ce que la géométrie du maillage ne soit pas affectée par l'aller-retour par le dual. Pour simplifier les choses, on donne la fonction (d'une grande simplicité) de dualisation suivante, qui permettra d'oublier pour un temps le plongement géométrique lorsque l'argument **`geometry`** est à **`False`**:

```
class GMap: (continued)

    def dual(self, geometry=True):
        """
        Compute the dual of the object.
        Create a new GMap object with the same darts but reversed alphas.
        Update positions of the dual 0-cells as the centers of the 2-cells
        """

        dual_gmap = GMap()
        for alpha_value in [0,1,2]:
            dual_gmap.alphas[alpha_value] = self.alphas[2-alpha_value]

        if geometry:
            for face_dart in self.elements(2):
                dual_gmap.set_position(face_dart,self.element_center(face_dart,2))

        return dual_gmap
```

Question 7

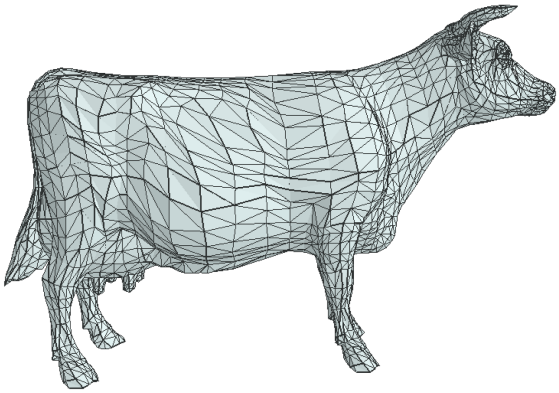
Utiliser les fonctions de dualisation et de suppression pour compléter la fonction d'effondrement d'arête sur une GMap représentant un maillage triangulaire. Quelques difficultés sont à prévoir, notamment pour l'indexation des cellules par un brin représentatif. En effet puisque l'on supprime des brins, il est par exemple possible que le brin associé à un sommet disparaisse lorsqu'on supprime une arête. Il va donc être nécessaire de maintenir des listes correspondant aux orbites des éléments qui vont être modifiés pour toujours pouvoir y accéder. Ensuite au niveau de la géométrie : si l'on ne veut pas qu'elle soit affectée par la dualisation, il faut la traiter en dehors, et donc réaffecter une position à chacun des éléments après l'opération topologique. Par ailleurs, il ne faut faire aucun changement avant d'avoir vérifié les conditions (arête incidente à deux faces, sans sommet de valence 3 adjacent) qui peuvent l'être sur le dual. Tester en effondrant une arête de l'octaèdre (dual du cube).

```
def triangular_gmap_dual_collapse_edge(gmap, dart):
    """
    Perform a topological collapse operation on a triangular GMap by
    removing an edge and two vertices in the dual structure, checking
    that all the topological constraints are respected.
    The resulting vertex is placed at the center of the original edge.
    """

    # Store the position of EACH DART in a dictionary
    # Store the position of the edge center

    try:
        # Compute the dual topology of the structure
        # Assert that the dual edge (same dart id) has two vertices

        # Store all the darts belonging to the vertices of the dual edge
        # Store all the darts belonging to the faces of the dual edge
```

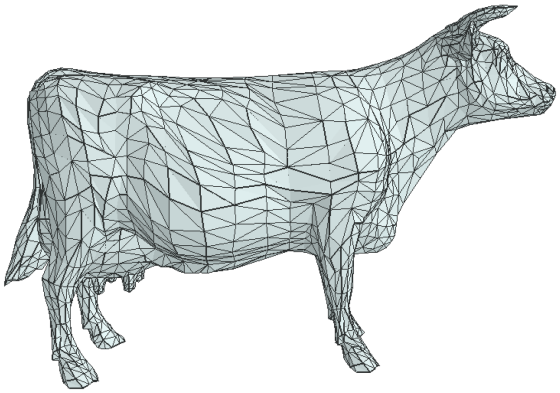
```
        # Make sure none of the vertex darts of the dual edge belongs to
        # a triangular face (dual of a vertex with valence 3)

        # Remove the dual edge from the dual
        # Eliminate the removed darts from the face and vertex dart lists

        # For each vertex (0-cell) represented in the vertex dart list:
            # Remove the vertex from the dual
            # Elimininate the removed darts from the face dart list

        # Update the primal alphas with the new alphas of the dual's dual
        # Set back the position of each primal vertex to its dart position
        # Erase the position of the dual face dart list if any
        # Set the postion of one representant in this list to the edge center

        return True
    except AssertionError:
        print "Impossible to collapse edge "+str(dart)
        return False
```

4.2 Décimation avec nombre cible de faces

Une fois implémentée cette opération élémentaire, on peut l'utiliser dans une procédure itérative visant à réduire le nombre d'éléments du maillage par collapses (optimaux) successifs. Une fonction réalisant cette optimisation est donnée dans le fichier **gmap_optimization.py**. Elle réalise les effondrements d'arêtes par ordre de longueur à condition que leur suppression n'entraine pas une erreur supérieure à un seuil donné sur la surface, ce jusqu'à atteindre un nombre cible de faces (ou que plus aucune arrête ne puisse être effondrée).



5804 Triangles [eccentricity : 0.13013, valence : 5.9959]



4070 Triangles [eccentricity : 0.14673, valence : 5.9941]

Question 8

Appliquer la fonction **gmap_edge_collapse_optimization** à un maillage triangulaire de votre choix (éventuellement issu d'une surface implicite). Essayer de commencer par des valeurs basses de l'erreur pour augmenter progressivement, et de la même façon de diminuer progressivement le nombre de triangles cible. Donner quelques exemples de décimations réussies avec les paramètres / appels associés.

5 Optimisation topologique et lissage

Le collapse n'est clairement pas une opération qui crée de la régularité dans un maillage, mais au contraire risque fort de rajouter des sommets irréguliers. En effet, sur une grille triangulaire parfaitement régulière (sommets uniformément de valence 6, triangle uniformément équilatéraux) le collapse d'une arête va créer deux sommets de valence 5 et un sommet de valence 8. Dans le cas général il va donc falloir combiner une décimation par collapse à une régularisation de la topologie, par exemple par flips successifs. De plus quelques passes de lissage permettront d'uniformiser quelque peu les longueurs d'arêtes et de rendre la décimation plus efficace.

Question 9

A l'aide des fonctions implémentées lors du TP3 (notamment les fonctions **`gmap_edge_flip_optimization`** et **`gmap_taubin_smoothing`** construire un processus itératif combinant les différentes opérations (dans l'ordre et avec l'intensité souhaités) de façon à réduire le nombre d'éléments tout en préservant voire améliorant la qualité du maillage. Pour cela vous pourrez utiliser la fonction suivante **`gmap_triangular_quality`** (dans **`gmap_tools.py`**) mesurant l'eccentricité moyenne des triangles et la valence moyenne des sommets. Présenter le résultat de vos essais.

6 Finalisation : du L-System au maillage optimisé

L'heure est à présent venue de tester l'ensemble de la chaîne de génération de maillage en utilisant comme entrée le fichier produit à l'aide de L-Py. On donne la fonction suivante qui permettra de charger un fichier tel que celui donné en exemple comme une liste de points et diamètre.

Question 10

En utilisant l'ensemble des fonctions développées jusqu'à présent écrire un script qui lit un fichier d'arbre généré à l'aide de L-Py, crée le squelette associé en utilisant les coordonnées de points et les diamètres, et utilise ce squelette pour générer une surface implicite. Appliquer ensuite les étapes d'amélioration de maillage que vous jugerez nécessaires. Tester d'abord sur des arborescences peu complexes avant d'augmenter le niveau de ramification (en pensant à ajuster la taille et la résolution de la grille d'évaluation) et présenter succintement les résultats obtenus (screenshots et commentaires bienvenus).

Question 11

Concours de la plus jolie structure générée. Essayez de produire des forme de coraux comme sous la figure ci dessous.