Code Documentation

Maria Linea Horgen
(Dated: July 28, 2020)

I. GENERAL INFORMATION AND BASICS

The following document contains code documentation for the class `PlyModel`, which is developed for verifying a variety of features for a 3D triangle polygonial model. Here the 3D model is assumed to be constructed in a separate 3D modelling program, i. e. blender or maya. Some of the class methods are designed to make alterations to the data set, while others simply points out problems that needs to be handled in the modelling program. This document is also created in the attempt to give some insight in the thoughts and ideas that have gone into constructing this class.

The class has some dependency on other modules, especially `NumPy`. To create an instance of the class, simply give the path to the .ply file (including the filename) as argument. The instance variables are treated in a similar manner as in C++, so they are all declared initially with value `None`. The class methods are like void functions or so called nonvalue-returning functions, were they alter already existing instance variables.

A. Instance variables

Below follows a table of all the instance variables and what data it holds. As mentioned, they have a value `None` initially, and are declared in the `__init__` method.

Variable name	Type	Data
self.N	int	total number of vertices
self.M	int	total number of polygons/faces
self.vertices	NumPy ndarray [Nx3], float	spatial coordinates of all the vertices
self.faces	NumPy ndarray [Mx3], int	vertex indices of all faces
self.half_edges	NumPy ndarray [3Mx3], int	directed half edges and associated polygons
self.sorted_edges	NumPy ndarray [3Mx3], int	sorted half edges and associated polygons
self.adjacency_mat	NumPy ndarray [MxM], int	adjacency matrix for the polygons
self.group_num	int	number of continuous groups of polygons
self.groups_index	dict	stores the polygon indices for each group
self.groups_faces	dict	stores the polygon faces for each group
self.non_manifold	Numpy array	holds the polygons that are not connected to two edges
self.normal_vectors	Numpy ndarray [Mx4], float	normal vectors of all the polygons

Table I. The table lists the instance variables, their type and what data they are meant to store.

By calling the different class methods the instance variables are assigned their intended data.

B. Class methods

The class consists currently of 15 methods. They are all listed and summarized below.

1. `ReadPly`
   Reads a .ply (binary or ascii) file and stores the vertex- and face-data in numpy arrays.

2. `Rotate`
   Rotates the polygonial model along the x-, y- or z-axis with an angle theta (in radians).

3. `SingleVertex`
   Removes single/unconnected vertices, and corrects the number of vertices and their index.

4. `UniqueVertices`
   Ensures that each set of spatial coordinates are unique, and deletes any duplicates.
5. `UniquePolygons`
   Verifies that each polygon consists of three unique vertices, and deletes any excess polygons.
6. `HalfEdges`
   Calculates the directed and undirected half-edges between vertices.
7. `AdjacentPolygons`
   Identifies neighbouring polygons, and stores the information in an adjacency matrix.
8. `RemoveSingles`
   Identifies and removes single polygons from the adjacency matrix, vertice-, face- and half edge arrays.
9. `FindingGroups`
   Determines how many continuous groups of polygons exists within the data set.
10. `NormalVectors`
    Calculates the normal vectors of the faces.
11. `FlippPolygons`
    Checks if the polygon orientation is consistent among adjacent polygons. If it is not, the adjacent polygon is flipped.
12. `RayCasting`
    Verifying that all the polygons are oriented outwards through ray casting.
13. `Manifold`
    Runs a simple check whether the object represented in the data set is manifold.
14. `Visualize`
    Uses matplotlib to visualize the polygonial model.
15. `WriteStaticVertices`
    Writes the vertex number, followed by its x, y and z coordinates to a .csv file.
16. `WriteStaticPolygons`
    Writes the polygon number, followed by the vertex number of each of the three polygon vertices in right-hand oriented order to a .csv file.

The rest of the document is dedicated to give a walk-through of each method, except for `ReadPly`, which can be found here [3].

Disclaimer: A lot of the codes are not optimized with respect to run time and efficiency, and the author acknowledges that there is room for improvement.

II. CLASS METHOD DESCRIPTIONS

PlyModel.Rotate(self, theta, axis=0)

The method rotates the vertices along the x-, y- or z-axis with an angle theta. All sets of spatial coordinates are multiplied with a rotation matrix. The matrix will vary depending on the choice of rotation axis. To avoid rounding errors close to zero a lower limit is set, if broken the cosine or sine are set to zero.

This method is implemented solely for visualization proposes.

Parameters:

theta : *int or float*

Rotation angle in radians

axis : *int, optional*

Corresponds to the axis of rotation. Axis 0 is the x-axis, axis 1 is the y-axis and axis 2 is the z-axis. The default is 0.

Source code

```
def Rotate(self, theta, axis=0):
    epsilon = 1e-14

    cosine = np.cos(theta)
    sine = np.sin(theta)

    # To avoid rounding errors close to zero
    cosine = 0 if cosine < epsilon else cosine
    sine = 0 if sine < epsilon else sine

    # Constructing the rotation matrix
    if axis == 0:                                      # rotation around the x-axis
        rotation_mat = np.array([(1,0,0), (0, cosine, -sine), (0, sine, cosine)]).reshape((3,3))

    if axis == 1:                                      # rotation around the y-axis
        rotation_mat = np.array([(cosine, 0, sine), (0,1,0), (-sine, 0, cosine)]).reshape((3,3))

    if axis == 2:                                      # rotation around the z-axis
        rotation_mat = np.array([(cosine, -sine, 0), (sine, cosine, 0), (0,0,1)]).reshape((3,3))

    tmp = np.zeros(3)                                  # stores the vector matrix product for each row vector

    for i in range(self.N):
        x = self.vertices[i]
        for j in range(3):
            for k in range(3):
                tmp[j] +=  rotation_mat[j][k] * x[k]        # vector matrix multiplication
        self.vertices[i] = tmp
        tmp = np.zeros(3)

    return None
```

PlyModel.SingleVertex(self)

The method removes single/unconnected vertices by iterating over the vertices in an outer loop, and for each vertex an inner loop checks if the vertex appears in any of the faces. If the vertex is located, the inner loop breaks, if not, the vertex is appended to a list containing all the single vertices.

All vertices with an index higher than the single vertices then has to be decremented by one to update the face array. This is done by iterating over the single vertices in descending order. The single vertices are so deleted, and N is updated.

Parameters: None

Source code

```
def SingleVertex(self):
    print("Searching for single vertices")

    singles = []                                        # holds the indices of the single vertices

    for i in range(self.N):
        boolean_check = np.zeros(self.M)
    for j in range(self.M):
        if np.any(self.faces[j,:] == i):                # if True, element is set to 1, and the inner loop breaks
            boolean_check = 1
            break
    if not np.any(boolean_check):                       # if all elements are False, the condition is executed.
        singles.append(i)

    if not singles:
        print("No single vertices detected")
    else:

        # if it exists single vertices, have to change the indices in the face array

        singles.reverse()                                       # iterating over the single vertices from highest to lowest

        # decrementing the indices which are greater than the single vertex index with one
        for i in singles:
            indices = np.where(self.faces > i)
            self.faces[indices] -= 1

        self.N -= len(singles)
        self.vertices = np.delete(self.vertices, singles, axis=0)   # deleting the single vertices from the vertex array

        print("Deleted all single vertices")
    return None
```

Example

$$\text{vertices} = \begin{bmatrix} [0 & 0 & 0] \\ [0 & 1 & 0] \\ [2 & 2 & 0] \\ [1 & 0 & 0] \end{bmatrix}, \quad \text{faces} = \begin{bmatrix} [0 & 1 & 3] \end{bmatrix} \quad \xrightarrow{\texttt{SingleVertex}} \quad \text{vertices} = \begin{bmatrix} [0 & 0 & 0] \\ [0 & 1 & 0] \\ [1 & 0 & 0] \end{bmatrix}, \quad \text{faces} = \begin{bmatrix} [0 & 1 & 2] \end{bmatrix}$$

PlyModel.UniqueVertices(self)

The method ensures that each set of spatial coordinates are unique, i. e. vertex[i] $\neq$ vertex[j] $\forall\ i, j$, where $i \neq j$. This is done through a number of steps.

By defining a set of spatial coordinates as the position $r_i = (x_i, y_i, z_i)$, the duplicates are identified by comparing r_i and r_j for $i = 0, ...N-1$ and $j = i+1, i+2, ..., N-1$. The duplicates are stored in a hash table, were the keys corresponds with the vertex index of the duplicated vertex.

An important notice is that an empty entry in the hash table can represent both a duplicate and a unique vertex without any duplicates, so all empty entries are checked before they are deleted from the table.

Since a vertex can have a number of duplicates, the highest valued duplicate (in terms of index) are used to identify and delete the rest of the duplicates. Next the vertex- and face array has to be updated.

The unique vertices (the keys in the hash table) are stored in a separate list, and sorted in ascending order. The vertex array is updated by creating an array of integers from 0 to $N-1$, and if the integer is not in the unique vertex list the integer is set equal to a constant that is greater than N. After all the duplicated indices are identified, they are deleted from the vertex array and integer array.

The first step in updating the face array is trivial, the keys in the hash table are now all unique vertices, so all of the values belonging to a key (here the values are vertex indices) are set equal to the key. To correct the remaining indices, the integer array used to update the vertex array is compared to a new array of integers. This new array represents the indices of the vertices in correct order. The idea is to preform an element wise comparison between the two arrays, and if they have a discrepancy the face element is set equal to the new array element.

Finally N is updated.

Parameters: None

Source code

```python
def UniqueVertices(self):
    duplicates = {}

    for i in range(self.N):
        dup = []
        xyz1 = self.vertices[i]                                        # spatial coordinates first polygon
        for j in range(i+1, self.N):
            xyz2 = self.vertices[j]                                    # spatial coordinates second polygon
            if xyz1[0] == xyz2[0]:
                if xyz1[1] == xyz2[1]:
                    if xyz1[2] == xyz2[2]:
                        dup.append(j)
        duplicates["{}".format(i)] = dup                               # hash table of all the duplicated polygons

    if not duplicates:
        print("All sets of spatial coordiantes are unique")
    else:
        print("Removing duplicates")
        unique_vertices = []

        # verifying that the polygons with an empty list in the hash table are in fact unique
        for key in duplicates:
            if not duplicates[key]:
                bool_check = []
                for keyy in duplicates:
                    values = duplicates[keyy]
                    if int(key) not in values:
                        bool_check.append(1)
                    else:
                        bool_check.append(0)
                        break
                if np.all(np.array(bool_check) == 1):
                    unique_vertices.append(int(key))


    # removing excess polygons from the hash table, i. e. if polygon j and k are duplicates of i, j and k can be removed from the table.
```

```
        for i in range(self.N):
            try:
                if not duplicates["{}".format(i)]:                  # if the polygon has no duplicates, it is deleted from the table
                    del duplicates["{}".format(i)]
                index = duplicates["{}".format(i)][-1]

                for j in range(i+1,self.N):
                    try:
                        indices = duplicates["{}".format(j)]
                        if index in indices:
                            del duplicates["{}".format(j)]
                    except KeyError:
                        continue
            except KeyError:
                continue


        # correcting the face- and vertex arrays

        # changing the indices of the duplicated polygons to the unique polygon index
        for key, values in duplicates.items():
            unique_vertices.append(int(key))
            for i in range(len(values)):
                indices = np.where(self.faces == values[i])
                self.faces[indices] = key

        unique_vertices.sort()

        # each duplicated vertex is now equal a constant > N
        vertices = np.array([i for i in range(self.N)])
        const = np.max(self.N)*10
        for vertex in vertices:
            if vertex not in unique_vertices:
                vertices[vertex] = const

        # removing the duplicated vertices
        indices = np.where(vertices == const)
        vertices = np.delete(vertices, indices)
        self.vertices = self.vertices[vertices]

        # correcting the face indices
        indices = np.array([i for i in range(len(unique_vertices))])

        for i in range(len(vertices)):
            if not vertices[i] == indices[i]:
                value = vertices[i]
                where_value = np.where(self.faces == value)
                self.faces[where_value] = indices[i]

        deleted = self.N - len(unique_vertices)

        self.N = len(unique_vertices)

        print("Deleted {} set(s) of excess coordinates".format(deleted))

    return None
```

Example

$$\text{vertices} = \begin{bmatrix} [0\ \ 0\ \ 0] \\ [0\ \ 1\ \ 0] \\ [0\ \ 0\ \ 1] \\ [1\ \ 1\ \ 0] \\ [0\ \ 0\ \ 0] \\ [1\ \ 1\ \ 1] \\ [1\ \ 1\ \ 1] \\ [1\ \ 2\ \ 0] \\ [0\ \ 0\ \ 0] \end{bmatrix}, \quad \text{faces} = \begin{bmatrix} [0\ \ 1\ \ 2] \\ [3\ \ 4\ \ 5] \\ [6\ \ 7\ \ 8] \end{bmatrix} \xrightarrow{\texttt{UniqueVertices}} \text{vertices} = \begin{bmatrix} [0\ \ 0\ \ 0] \\ [0\ \ 1\ \ 0] \\ [0\ \ 0\ \ 1] \\ [1\ \ 1\ \ 0] \\ [1\ \ 1\ \ 1] \\ [1\ \ 2\ \ 0] \end{bmatrix}, \quad \text{faces} = \begin{bmatrix} [0\ \ 1\ \ 2] \\ [3\ \ 0\ \ 4] \\ [4\ \ 5\ \ 0] \end{bmatrix}$$

$$\text{duplicates} = \{"0" : [4, 8], "1" : [], "2" : [], "3" : [], "4" : [8], "5" : [6], "6" : [], "7" : [], "8" : []\}$$

PlyModel.UniquePolygons(self)

The method verifies that each polygon consists of three unique vertices, i. e. that none of the row elements in the face array are equal to each other. This corresponds with a polygon with no area.

To identify identical vertices in a polygon, the three vertices are compared against each other, if there is a match, the polygon is deleted.

Parameters: None

Source code

```python
def UniquePolygons(self):
    not_unique = []

    for i in range(self.M):
        if self.faces[i,0] == self.faces[i,1] or self.faces[i,1] == self.faces[i,2] or self.faces[i,0] == self.faces[i,2]:
            not_unique.append(i)

    if not not_unique:
        print("All polygons consists of three unique vertices")
    else:
        self.faces = np.delete(self.faces, not_unique, axis=0)
        self.M -= len(not_unique)

        print("All excess polygons deleted")
    return None
```

Example

$$\text{faces} = \begin{bmatrix} [0 & 1 & 2] \\ [3 & 4 & 5] \\ [6 & 8 & 8] \end{bmatrix} \xrightarrow{\texttt{UniquePolygons}} \text{faces} = \begin{bmatrix} [0 & 1 & 2] \\ [3 & 4 & 5] \end{bmatrix}$$

PlyModel.HalfEdges(self)

Calculates the half-edges between the vertices in each polygon. A half-edge is an edge of a polygon, with a pointer to its associated face. Since an edge is normally shared by two polygons, this data structure is called a half-edge. Each half-edge is created with its first vertex stored before the second vertex.

There are three half-edges for each polygon, making the array storing the half-edges $3M$ rows long. Each row contains the start- and end vertex for the edge, and their associated polygon. Hence the half-edge arrays are $3M \times 3$ matrices. The function calculates both the directed and undirected half-edges, and stores them in two different arrays.

The undirected half-edges are computed so that comparing edges, and identify adjacent polygons, is a matter of comparing first to first and second to second vertex indices.

The associated polygons are appended as a column vector, so that both of the half-edge arrays are $3M \times 3$ matrices.

Parameters: None

Source code

```python
def HalfEdges(self):
    edges = np.zeros([self.M*3,2])
    associated_polygon = np.zeros([self.M*3])

    numbers = [1,2,0]

    counter = 0
    for i in range(self.M):
        polygon = i
        for j in range(len(numbers)):
            edges[counter,0] = self.faces[i,j]                      # start vertex
            edges[counter,1] = self.faces[i,numbers[j]]             # end vertex
            associated_polygon[counter] = polygon
            counter += 1


    self.half_edges = edges.copy()
    self.half_edges = np.c_[self.half_edges, associated_polygon]    # directed half-edges and their associated polygons
    self.half_edges = self.half_edges.astype("int64")

    edges = np.sort(edges)                              # sorting each row
    edges = np.c_[edges, associated_polygon]
    sorted_index = np.lexsort(np.fliplr(edges).T)       # sorting the half-edges by its coordinates
    self.sorted_edges = edges[sorted_index]             # undirected half-edges and their associated polygons
    self.sorted_edges = self.sorted_edges.astype("int64")

    print("Computing half edges complete")
    return None
```

Example

$$\text{faces} = \begin{bmatrix} [0 & 1 & 2] \\ [3 & 4 & 5] \end{bmatrix} \xrightarrow{\texttt{HalfEdges}} \text{half edges} = \begin{bmatrix} [0 & 1 & 0] \\ [1 & 2 & 0] \\ [2 & 0 & 0] \\ [3 & 4 & 1] \\ [4 & 5 & 1] \\ [5 & 3 & 1] \end{bmatrix}, \quad \text{sorted edges} = \begin{bmatrix} [0 & 1 & 0] \\ [0 & 2 & 0] \\ [1 & 2 & 0] \\ [3 & 4 & 1] \\ [3 & 5 & 1] \\ [4 & 5 & 1] \end{bmatrix}$$

PlyModel.AdjacentPolygons(self)

By comparing the sorted half-edges this method identifies adjacent polygons, and stores the information in an adjacency matrix. The matrix is an undirected binary symmetric matrix, with zeros on its diagonal. If polygon i and polygon j are adjacent, the matrix element is $A_{ij} = 1$, if not, $A_{ij} = 0$. The polygon numbers are stored in the third column in the sorted half-edge array.

Since the matrix is symmetric, only the upper triangular values are calculated. Two polygons are adjacent if they share an edge, so the identification of neighbouring polygons are done through matching identical half-edges. If the half-edges were matched with the directed half-edges we would have $(3M - 1)^2$ iterations, comapred to $(3M - 1)$.

Note: The adjacency matrix is a sparse matrix, meaning most of the elements are zero, so an alternative storage format may be preferred if M is large, i. e compressed row storage [4] or a hash table.

Parameters: None

Source code

```
def AdjacentPolygons(self):
    self.adjacency_mat = np.zeros([self.M, self.M])

    # the matrix is symmetric, only calculating the upper triangular values
    for i in range(len(self.sorted_edges)-1):                       # the minus one is due to the i+1 in edge2
        edge1 = self.sorted_edges[i,0:2]
        edge2 = self.sorted_edges[i+1,0:2]
        if np.all(edge1==edge2):
            self.adjacency_mat[self.sorted_edges[i,-1], self.sorted_edges[i+1,-1]] = 1                        # A_ij = 1

    self.adjacency_mat = self.adjacency_mat + self.adjacency_mat.T - np.diag(np.diag(self.adjacency_mat))      # filling in the lower
     triangular values

    print("Computing adjacent polygons complete")
    return None
```

Example

$$\text{faces} = \begin{bmatrix} [0 & 1 & 3] \\ [1 & 2 & 3] \\ [4 & 5 & 6] \end{bmatrix}, \quad \text{sorted edges} = \begin{bmatrix} [0 & 1 & 0] \\ [0 & 3 & 0] \\ [1 & 2 & 1] \\ [1 & 3 & 0] \\ [1 & 3 & 1] \\ [2 & 3 & 1] \\ [4 & 5 & 2] \\ [4 & 6 & 2] \\ [5 & 6 & 2] \end{bmatrix}, \quad \xrightarrow{\texttt{AdjacentPolygons}} \quad \text{adjacency matrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

PlyModel.RemoveSingles(self)

Deletes single polygons from the data set. A polygon is single if its corresponding row- and column vectors are the null vectors in the adjacency matrix. So the singles are identified by iterating through the row vectors, and if the ith row only consists of zeros, polygon i is single and can be deleted.

The singles are stored in a list, so that they can be removed from the vertex-, faces-, directed half-edge arrays and the adjacency matrix. First the indices to the vertices belonging to the single polygon are identified and deleted.

By altering the vertex array, the indices in the faces array has to be updated, and the single polygon deleted. The first step is to identify where in the faces array the single polygon is located, so it can be deleted. Then each vertex index belonging to the single polygon are sorted in descending order and iterated over. In the for loop all indices that are greater than the current index are decremented by one.

To correct the directed half-edge array, the half-edges belonging to the single polygons are identified and deleted. Here the associated polygon numbers, as well as the half-edge indices has to be adjusted. This is done in the same manner as with the faces array.

Lastly the number of vertices and faces are corrected, and the adjancency matrix are updated.

Parameters: None

Source code

```
def RemoveSingles(self):
    single_poly = []

    for i in range(self.M):
        if not np.any(self.adjacency_mat[i,:] != 0):
            single_poly.append(i)


    if not single_poly:
        print("No single polygons detected")
    else:
        print("Detected {} single polygon(s)".format(len(single_poly)))

        for i in range(len(single_poly)):
            rm_vertices = self.faces[single_poly[i]]             # the indices to the vertices belonging to the single polygon

            # deleting the vertices in the vertex array
            self.vertices = np.delete(self.vertices, rm_vertices, axis=0)

            # deleting the faces in the faces array
            indices = np.where(self.faces == rm_vertices)
            decrement = np.sort(self.faces[indices])
            decrement = list(decrement[::-1])
            self.faces = np.delete(self.faces, indices[0][0], axis=0)

            for j in decrement:
                self.faces[np.where(self.faces > j)] -= 1

            # deleting the half edges in the directed half edge array
            polygon_num = self.half_edges[np.where(self.half_edges[:,:-1] == rm_vertices[0])[0][-1]][-1]   # associated polygon number of
      the single polygon
            indices = np.where(self.half_edges[:,-1] == polygon_num)
            decrement = np.sort(self.half_edges[indices,0])[0]
            decrement = list(decrement[::-1])
            self.half_edges = np.delete(self.half_edges, indices, axis=0)

            for j in decrement:
                self.half_edges[np.where(self.half_edges[:,:-1] > j)] -= 1

            decrement = np.where(self.half_edges[:,-1] > polygon_num)
            self.half_edges[decrement[0],-1] -= 1   # decrementing the associated polygon numbers which are larger than the deleted
     polygon

            self.N -= 3                             # correcting the number of vertices
            self.M -= 1                             # correcting the number of polygons
            self.adjacency_mat = np.delete(self.adjacency_mat, (single_poly[i]), axis=0)  # deleting the associated row in the adjacency
     matrix
            self.adjacency_mat = np.delete(self.adjacency_mat, (single_poly[i]), axis=1)  # deleting the associated column in the
     adjacency matrix
        print("Deleted all single polygons")
    return None
```

Example in 2D

$$\text{vertices} = \begin{bmatrix} [1 & 2] \\ [3 & 5] \\ [2 & 1] \\ [4 & 1] \\ [5 & 5] \\ [3 & 4] \\ [4 & 6] \end{bmatrix} \quad \text{faces} = \begin{bmatrix} [0 & 2 & 5] \\ [1 & 4 & 6] \\ [2 & 3 & 5] \end{bmatrix}, \quad \text{sorted edges} = \begin{bmatrix} [0 & 2 & 0] \\ [0 & 5 & 0] \\ [1 & 4 & 1] \\ [1 & 6 & 1] \\ [2 & 3 & 2] \\ [2 & 5 & 0] \\ [2 & 5 & 2] \\ [3 & 5 & 2] \\ [4 & 6 & 1] \end{bmatrix}, \quad \text{adjacency matrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$\xrightarrow{\texttt{RemoveSingles}}$$

$$\text{vertices} = \begin{bmatrix} [1 & 2] \\ [2 & 1] \\ [4 & 1] \\ [3 & 4] \end{bmatrix} \quad \text{faces} = \begin{bmatrix} [0 & 1 & 3] \\ [1 & 2 & 3] \end{bmatrix}, \quad \text{sorted edges} = \begin{bmatrix} [0 & 1 & 0] \\ [0 & 3 & 0] \\ [1 & 2 & 1] \\ [1 & 3 & 0] \\ [1 & 3 & 1] \\ [2 & 3 & 1] \end{bmatrix}, \quad \text{adjacency matrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

PlyModel.FindingGroups(self)

The method determines how many continuous groups of polygons exists within the data set, based on the adjacency matrix. All nonzero values in the upper- or lower triangular adjacency matrix represents a pair of neighbouring polygons. These face indices are zipped together into a list, and iterated over. The idea is that the highest index in the pair is given the value of the lowest index in an integer array from $0, ..., M-1$, and all values in the array which are greater than the highest index are decremented by one.

As you iterate and make changes to the integer array, an if-test is in place to make sure that you do not compare two polygons that belongs to same group.

At the end the number of unique elements represents the number of continuous groups in the data set. If it exists more than one group the method fills two dictionaries, one storing the polygon indices for each group and one storing the faces.

Parameters: None

Source code

```python
def FindingGroups(self):
    neighbours = np.where(np.tril(self.adjacency_mat) == 1)
    indices = list(zip(neighbours[0], neighbours[1]))
    group_vector = np.array([i for i in range(self.M)])

    for index in indices:
        index1 = index[0]
        index2 = index[1]
        high = max(group_vector[index1], group_vector[index2])
        low = min(group_vector[index1], group_vector[index2])
        if high != low:
            decrease = np.where(group_vector > high)
            same_value = np.where(group_vector == high)     # the indices in group_vector that has the same value as the high
            high = low                                      # sets the value correspoding to the highest value equal to the value of the
  lowest
            group_vector[same_value] = low                  # changing the values that are identical to the highest value
            group_vector[decrease] -= 1


    self.group_num = len(np.unique(group_vector))
    self.groups_index = {}
    self.groups_faces = {}


    # filling the groups dict with the corresponding polygons and their vertex indices
    for i in range(self.group_num):
        indices = np.where(group_vector == i)
        self.groups_index["group{}".format(i+1)] = indices
        self.groups_faces["group{}".format(i+1)] = self.faces[indices]

    print("All groups identified")
    print("Number of groups: {}".format(self.group_num))
    return None
```

Example

$$\text{adjacency matrix} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \xrightarrow{\texttt{FindingGroups}}$$

Matrix element	group_vector					
	0	1	2	3	4	5
A_{01}	0	0	1	2	3	4
A_{02}	0	0	0	1	2	3
A_{03}	0	0	0	0	1	2
A_{54}	0	0	0	0	1	1

$$\text{groups} = \{\text{"group1"} : [0, 1, 2, 3], \text{"group2"} : [4, 5]\}$$

PlyModel.NormalVectors(self)

The method calculates the surface normal vectors of the faces and stores them in an array. The first column in the array corresponds with the polygon index. For a triangle the surface normal can be calculated as the vector cross product of two (non-parallel) edges of the polygon.

The cross product is calculated in the following way,

$$(x_2 - x_1, y_2 - y_1, z_2 - z_1) \times (x_3 - x_1, y_3 - y_1, z_3 - z_1), \tag{1}$$

where x_i, y_i and z_i for $i \in \{1, 2, 3\}$ are the vertices of the polygon. The vectors being crossed in eq. (1) are $(v_2 - v_1)$ and $(v_3 - v_1)$.

Parameters: None

Source code

```
def NormalVectors(self):
    self.normal_vectors = np.zeros([self.M, 4])

    for index, polygon in enumerate(self.faces):
        vec1 = self.vertices[polygon[1]] - self.vertices[polygon[0]]
        vec2 = self.vertices[polygon[2]] - self.vertices[polygon[0]]
        cross_product = np.cross(vec1,vec2)
        self.normal_vectors[index] = [index, cross_product[0], cross_product[1], cross_product[2]]
    return None
```

PlyModel.FlippPolygons(self)

The method ensures that the different groups has a consistent mesh orientation; that they are all oriented the same way, either left-hand- or right-hand oriented.

The first step is to convert the adjacency matrix to an adjacency list. Then the first polygon to appear in the `groups_index` dictionary is chosen to be the starting polygon in each group. Each of its neighbours are checked, meaning that the directed half-edges are matched, and if they are identical the neighbouring polygon is flipped. This is recursively repeated for the neighbours of the neighbours, until all of the polygons in one group are checked once.

If some of the polygons are flipped all of the half-edges and normal vectors are recalculated.

An important notice is that if the starting polygon is left- hand oriented (right-hand oriented) all the polygons in that group will end up being left-hand oriented (right-hand oriented).

Parameters: None

Source code

```
def FlippPolygons(self):
    # changing the adjacency matrix to a adjaceny list
    adjacency_list = defaultdict(list)
    for i in range(self.M):
        for j in range(self.M):
            if self.adjacency_mat[i][j] == 1:
                adjacency_list[i].append(j)


    counter = 0
    for key, values in self.groups_index.items():
        polygons = list(values[0])
        while polygons:
            poly = polygons[0]                             # starting polygon
            HE1 = np.where(self.half_edges[:,-1] == poly)  # indices to the half-edges belonging to the start polygon
            neighbours = adjacency_list[poly]              # the neighbours to the start polygon
            for neighbour in neighbours:                   # iterating over the neighbours to the start polygon
                if neighbour in polygons:                  # verifying that the polygon are not previously checked
                    polygons.remove(neighbour)             # to make sure the polygon are not iterated over at a later time
                    polygons.insert(0, neighbour)          # by making the neighbour the first element, its neighbours are checked in the
      next iteration
                    HE2 = np.where(self.half_edges[:,-1] == neighbour)  # indices to the half-edges belonging to the neighbouring polygon

                    # matching half-edges
                    for i in range(3):
                        half_edge1 = self.half_edges[HE1[0][i],:-1]
                        for j in range(3):
                            half_edge2 = self.half_edges[HE2[0][j],:-1]
                            if half_edge1[0] == half_edge2[0] and half_edge1[1] == half_edge2[1]:          # if the polygons have a
     matching pair of half-edges they have opposite orientation
                                self.faces[neighbour][1], self.faces[neighbour][2] = self.faces[neighbour][2], self.faces[neighbour][1]
                                indices = np.where(self.half_edges[:,-1] == neighbour)
                                self.half_edges[indices[0],:-1] = np.flip(self.half_edges[indices[0],:-1], axis=1) # flips the directed
     half-edges
                                counter += 1
                                break
                        else:
                            continue
                        break
            polygons.remove(poly)                              # removing the start polygon when all the neighbours are checked

        print("Flipped {} polygons".format(counter))

        if counter != 0:
            print("Recalculating the half-edges and normal vectors")
            self.HalfEdges()
            self.NormalVectors()
        return None
```

Example in 2D

$$\text{vertices} = \begin{bmatrix} [1 & 1] \\ [2 & 1] \\ [3 & 1] \\ [2 & 3] \end{bmatrix}, \quad \text{faces} = \begin{bmatrix} [0 & 1 & 3] \\ [1 & 3 & 2] \end{bmatrix}$$

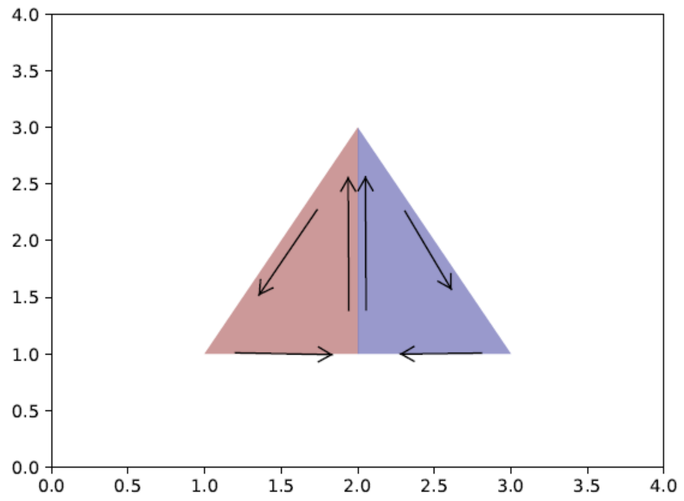

Figure 1. The figures display the orientation of two polygons, where the polygon to the right is left-hand oriented and the one to the left is right-hand oriented.

$$\xrightarrow{\texttt{FlippPolygons}}$$

$$\text{vertices} = \begin{bmatrix} [1 & 1] \\ [2 & 1] \\ [3 & 1] \\ [2 & 3] \end{bmatrix}, \quad \text{faces} = \begin{bmatrix} [0 & 1 & 3] \\ [1 & 2 & 3] \end{bmatrix}$$

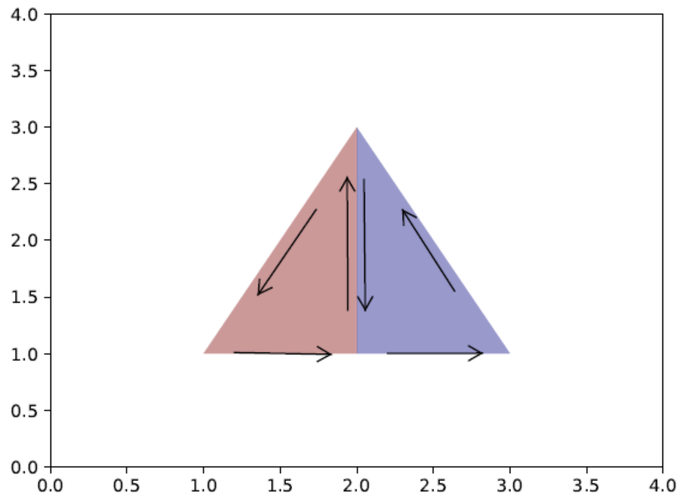

Figure 2. The figure displays two polygons that are right-hand oriented.

PlyModel.RayCasting(self)

The method determines if all the polygons are facing outwards or inwards. In the case where the polygonas are facing inwards, they are flipped. The desired result is that all polygons are facing outwards. In order for the method to work properly, the `FlippPlygons` method has to be applied in advance. The determination of orientation is done through the Möller–Trumbore ray-triangle intersection algorithm [2], which is abbreviated to the MT algorithm.

The idea is to cast a ray from the centroid (with a small displacement) of each polygon and count the number of times it intersect with a polygon face. If the number of intersections is odd the polygon is oriented inwards, and vice versa if number of intersections are an even number.

The MT algorithm is a fast method for calculating the intersection of a ray and a triangle in three dimensions without needing precomputation of the plane equation of the plane containing the triangle [2]. The algorithm is implemented in a static method and called upon fo

Parameters: None

Source code

Example

PlyModel.Manifold(self, x_lim=[], y_lim=[], z_lim=[], multiplier=1, visualize=False)

Runs a simple check whether the object represented in the data set is manifold, and in the case of non-manifold geometry, gives you the option to visualize the problematic parts of the object.

A manifold surface is one without topological inconsistencies, i. e. three or more polygons share an edge, or two or more corners touching each other [1]. If each edge is shared by exactly two faces, the model is said to be manifold. This can be verified by the simple equation,

$$V + F - E = 2, \tag{2}$$

where V is the number of vertices, F is the number of faces and E is the number of edges.

There are several different types of non-manifold errors:

– Disconnected vertices and edges
– More than two polygons share an edge
– Bowtie geometry
– Internal faces
– Areas with no thickness
– Holes and open objects
– Opposite normals

Some of these errors are already taken care of, disconnected vertices and edges and opposite normals, while other problems are easiest dealt with in the modelling program. This method identifies the polygons that are constructed by edges that are not shared by exactly two faces and if multiple faces share a common vertex but no edges (bowtie geometry). Internal faces and holes are easiest discovered in the GUI.

If a vertex is shared by two or more continuous groups, the model contains bowtie geometry. To identify vertices that are shared among the groups, each vertex is searched for in each group. If a vertex appears once in a particular group, the loop searching for it breaks and moves on to the next group. The shared vertices are stored in a hash table, were the index is the key.

To identify the edges that are not shared by two polygons the appearance of each undirected half-edge is counted. This is done through a static method which is parallelized to speed up the process. The index corresponding with the edge, in an array of zeros with the length of the number of edges (`HalfEdgeCounter`), is incremented by one each time the edge appears in the undirected half-edge array. Then the polygons that are constructed by these edges are identified through matching the half-edges with the edges that has a count $\neq 2$ in the `HalfEdgeCounter` array, and extracting the associated polygon number.

Parameters:

x_lim : *twoelement list, optional*

Set the x-axis view limit from [low, high]. Default is the maximum and minimum value of all the x coordinates.

y_lim : *twoelement list, optional*

Set the x-axis view limit from [low, high]. Default is the maximum and minimum value of all the y coordinates.

z_lim : *twoelement list, optional*

Set the x-axis view limit from [low, high]. Default is the maximum and minimum value of all the z coordinates.

multiplier : *int, optional*

Multiplies the vertex coordinates with the factor "multiplier". The default is 1.

visualize : *bool, optional*

If True the functions plots the polygons that contain edges that are not connected to exactly two faces on top of the other polygons. The unconnected polygons are red, while the others are green. The default is False

Source code

```
def Manifold(self, x_lim=[], y_lim=[], z_lim=[], multiplier=1, visualize=False):
    E = int(len(self.half_edges)*0.5)                   # division by two because each half edge should appear twice

    ManifoldCheck = self.N + self.M - E

    # check for multiple faces sharing a common vertex but no edge.

    if ManifoldCheck == 2:
        print("The model is manifold")
    else:
        print("The model is non-manifold")
        print("Identifying the problem...")

        # identifying shared vertices among the groups
        shared_vertices = {}

        for i in range(self.N):
            sharing_groups = []
            for key in self.groups:
                values = self.groups[key]
                for j in range(len(values)):
                    if i in values[j]:
                        sharing_groups.append(key)
                        break
            if len(sharing_groups)>1:
                shared_vertices["{}".format(i)] = sharing_groups

        # Identifying the polygons that are constructed by edges that are not shared by exactly two faces

        Edges = np.unique(self.sorted_edges[:,:-1], axis=0)         # the unique elements in sorted_edges equals all of the edges
        HalfEdgeCounter = np.zeros(len(Edges))                      # each half-edge should appear twice if the model is manifold

        self.ManifoldLoop(Edges, self.sorted_edges[:,:-1], HalfEdgeCounter)     # counting the appearances of each half-edge

        appear_once = len(np.where(HalfEdgeCounter == 1)[0])
        appear_twice = len(np.where(HalfEdgeCounter == 2)[0])
        appear_more = len(np.where(HalfEdgeCounter > 2)[0])

        NonManifold = []
        for i in range(len(Edges)):
            Edge = Edges[i]
            count = HalfEdgeCounter[i]
            if count != 2:
                for j in range(len(self.sorted_edges)):
                    HalfEdge = self.sorted_edges[j,:-1]
                    if Edge[0] == HalfEdge[0] and Edge[1] == HalfEdge[1]:
                        NonManifold.append(self.sorted_edges[j,-1])


        self.non_manifold = np.array(NonManifold)
        self.non_manifold = self.non_manifold.astype("int64")

        if not appear_once and not appear_more:
            print("Shared vertices in the data set:")
            for key, values in shared_vertices.items():
                print("Vertex {} is shared between {}".format(key, values))

        elif not shared_vertices:
            print("Number of half-edges appearing once: {}".format(appear_once))
            print("Number of half-edges appearing twice: {}".format(appear_twice))
            print("Number of half-edges appearing more than twice: {}".format(appear_more))
        else:
            print("Number of half-edges appearing once: {}".format(appear_once))
            print("Number of half-edges appearing twice: {}".format(appear_twice))
            print("Number of half-edges appearing more than twice: {}".format(appear_more))
            print("Shared vertices in the data set:")
            for key, values in shared_vertices.items():
                print("Vertex {} is shared between {}".format(key, values))

        if visualize:
            fig = plt.figure()
            ax = fig.add_subplot(projection="3d")
            v = self.vertices*multiplier
            faces = self.faces[self.non_manifold]

            pc = art3d.Poly3DCollection(v[self.faces], facecolors="lightgreen", edgecolor=(0,0,0,0.1))
            ax.add_collection(pc)

            pc = art3d.Poly3DCollection(v[faces], facecolors="darkred", edgecolor=(0,0,0,0.1))
            ax.add_collection(pc)

            for key in shared_vertices:
                x = self.vertices[int(key)][0]*multiplier
                y = self.vertices[int(key)][1]*multiplier
                z = self.vertices[int(key)][2]*multiplier

                ax.scatter(x,y,z, color="magenta", marker=".")

            if not x_lim and not y_lim and not z_lim:
                x_lim = [np.min(self.vertices[:,0]),np.max(self.vertices[:,0])]
                y_lim = [np.min(self.vertices[:,1]),np.max(self.vertices[:,1])]
                z_lim = [np.min(self.vertices[:,2]),np.max(self.vertices[:,2])]

                ax.set_xlim3d(x_lim[0], x_lim[1])
                ax.set_ylim3d(y_lim[0], y_lim[1])
                ax.set_zlim3d(z_lim[0], z_lim[1])

            else:
                ax.set_xlim3d(x_lim[0], x_lim[1])
                ax.set_ylim3d(y_lim[0], y_lim[1])
                ax.set_zlim3d(z_lim[0], z_lim[1])

            plt.show()
        return None

@staticmethod
@njit(parallel=True)
def ManifoldLoop(Edges, HalfEdges, HalfEdgeCounter):
    for i in prange(len(Edges)):
        HalfEdge1 = Edges[i]
        for j in range(len(HalfEdges)):
            HalfEdge2 = HalfEdges[j]
            if HalfEdge1[0] == HalfEdge2[0] and HalfEdge1[1] == HalfEdge2[1]:
                HalfEdgeCounter[i] += 1
        return None
```

Example

$$\text{vertices} = \begin{bmatrix} -1 & 1 & -3 \\ -1 & -1 & -3 \\ 0 & 0 & -1 \\ 1 & -1 & -3 \\ 1 & 1 & -3 \\ -1 & -1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ -1.8 & -0.5 & 1.9 \\ -1.8 & 1.5 & 1.9 \end{bmatrix} \quad \text{faces} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 2 & 1 \\ 3 & 4 & 2 \\ 0 & 3 & 1 \\ 5 & 6 & 2 \\ 6 & 7 & 8 \\ 0 & 2 & 4 \\ 7 & 6 & 5 \\ 8 & 7 & 2 \\ 2 & 7 & 5 \\ 2 & 6 & 8 \\ 4 & 3 & 0 \\ 5 & 6 & 9 \\ 10 & 9 & 6 \end{bmatrix} \quad \text{groups} = \left\{ \begin{array}{ll} \text{"group1"}: & [[0\ \ 1\ \ 2], \\ & [3\ \ 2\ \ 1] \\ & [3\ \ 4\ \ 2] \\ & [0\ \ 2\ \ 4] \\ & [4\ \ 3\ \ 0] \\ & [0\ \ 3\ \ 1]] \\ \text{"group2"}: & [[5\ \ 6\ \ 2] \\ & [6\ \ 7\ \ 8] \\ & [7\ \ 6\ \ 5] \\ & [8\ \ 7\ \ 2] \\ & [2\ \ 7\ \ 5] \\ & [2\ \ 6\ \ 8]] \\ & [5\ \ 6\ \ 9]] \\ & [10\ \ 9\ \ 6]] \end{array} \right\}$$

$$\xrightarrow{\texttt{Manifold([-3,3], [-3,3], [-3,3], multiplier = 1, visualize=True)}}$$

manifold_eksempel.pdf

Figure 3. The figure displays a non-manifold object with multiple faces are sharing a common vertex (in magenta) but no edge and the polygons that are connected to an edge that with more than two polygons (the red faces).

Output in terminal

```
The model is non-manifold
Identifying the problem...
Number of half-edges appearing once: 3
Number of half-edges appearing twice: 18
Number of half-edges appearing more than twice: 1
Shared vertices in the data set:
Vertex 2 is shared between ['group1', 'group2']
```

PlyModel.Visualize(self, x_lim=[], y_lim=[], z_lim=[], multiplier = 1, group=None)

The method plots the vertices and faces in a 3D-plot, using matplotlib. If the data set is one continuous group the plot will be unicolored, if not each group will have a different color (if choosing to plot all of the groups).

If the data set is one continuous group or if the groups is yet to be identified, and the `group` parameter has the default value, the method plots the object as one group.

In the case where `group` ≠ `None`, the specified groups are visualized.

Lastly, if the data set consists of several groups, they are plotted together, but in different colors.

Parameters:

x_lim : *twoelement list, optional*

Set the x-axis view limit from [low, high]. Default is the maximum and minimum value of all the x coordinates.

y_lim : *twoelement list, optional*

Set the x-axis view limit from [low, high]. Default is the maximum and minimum value of all the y coordinates.

z_lim : *twoelement list, optional*

Set the x-axis view limit from [low, high]. Default is the maximum and minimum value of all the z coordinates.

multiplier : *int, optional*

Multiplies the vertex coordinates with the factor ”multiplier”. The default is 1.

group : *int or list, optional*

Specify which group(s) to visualize. Default is `None`, which plots all of the groups.

Source code

```
def Visualize(self, x_lim=[], y_lim=[], z_lim=[], multiplier = 1, group=None):
        fig = plt.figure()
        ax = fig.add_subplot(projection="3d")

        v = self.vertices*multiplier

        if self.group_num == 1 or self.group_num == None and group == None:
            f = self.faces

            r = np.random.rand()
            b = np.random.rand()
            g = np.random.rand()
            color = (r, g, b)

            pc = art3d.Poly3DCollection(v[f], facecolors=color, edgecolor=(0,0,0,0.1))
            ax.add_collection(pc)

        elif group != None:

            if isinstance(group, int):
                f = self.groups_faces["group{}".format(group)]

                r = np.random.rand()
                b = np.random.rand()
                g = np.random.rand()
                color = (r, g, b)

                pc = art3d.Poly3DCollection(v[f], facecolors=color, edgecolor=(0,0,0,0.1))
                ax.add_collection(pc)

            elif isinstance(group, list):
               for i in range(len(group)):
                    f = self.groups_faces["group{}".format(group[i])]

                    r = np.random.rand()
                    b = np.random.rand()
                    g = np.random.rand()
                    color = (r, g, b)

                    pc = art3d.Poly3DCollection(v[f], facecolors=color, edgecolor=(0,0,0,0.1))
                    ax.add_collection(pc)
            else:
                raise TypeError("group parameter is neither list or int, but {}".format(type(group)))
        else:

            for i in range(self.group_num):
                f = self.groups_faces["group{}".format(i+1)]

                r = np.random.rand()
                b = np.random.rand()
                g = np.random.rand()
                color = (r, g, b)


                pc = art3d.Poly3DCollection(v[f], facecolors=color, edgecolor=(0,0,0,0.1))

                ax.add_collection(pc)

        if not x_lim and not y_lim and not z_lim:
            x_lim = [np.min(self.vertices[:,0]),np.max(self.vertices[:,0])]
            y_lim = [np.min(self.vertices[:,1]),np.max(self.vertices[:,1])]
            z_lim = [np.min(self.vertices[:,2]),np.max(self.vertices[:,2])]

            ax.set_xlim3d(x_lim[0], x_lim[1])
            ax.set_ylim3d(y_lim[0], y_lim[1])
            ax.set_zlim3d(z_lim[0], z_lim[1])

        else:
            ax.set_xlim3d(x_lim[0], x_lim[1])
            ax.set_ylim3d(y_lim[0], y_lim[1])
            ax.set_zlim3d(z_lim[0], z_lim[1])


        plt.show()

        return None
```

Example

$$\text{vertices} = \begin{bmatrix} -1 & 1 & -3 \\ -1 & -1 & -3 \\ 0 & 0 & -1 \\ 1 & -1 & -3 \\ 1 & 1 & -3 \\ -1 & -1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \text{faces} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 2 & 1 \\ 3 & 4 & 2 \\ 0 & 3 & 1 \\ 5 & 6 & 2 \\ 6 & 7 & 8 \\ 0 & 2 & 4 \\ 7 & 6 & 5 \\ 8 & 7 & 2 \\ 2 & 7 & 5 \\ 2 & 6 & 8 \\ 4 & 3 & 0 \end{bmatrix} \quad \text{groups} = \left\{ \begin{array}{ll} \text{"group1"}: & [[0\ \ 1\ \ 2], \\ & [3\ \ 2\ \ 1] \\ & [3\ \ 4\ \ 2] \\ & [0\ \ 2\ \ 4] \\ & [4\ \ 3\ \ 0] \\ & [0\ \ 3\ \ 1]] \\ \text{"group2"}: & [[5\ \ 6\ \ 2] \\ & [6\ \ 7\ \ 8] \\ & [7\ \ 6\ \ 5] \\ & [8\ \ 7\ \ 2] \\ & [2\ \ 7\ \ 5] \\ & [2\ \ 6\ \ 8]] \end{array} \right\}$$

$$\xrightarrow{\texttt{FindingGroups([-3,3], [-3,3], [-3,3], multiplier = 1, group=None)}}$$

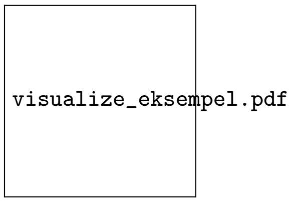

Figure 4. The figure displays how the `PlyModel.Visualize` method visualize an object made up of two continuous groups of triangular polygons. Notice: this is a non-manifold object.

PlyModel.WriteStaticVertices(self, path=None)

The function writes the vertex number, followed by its x, y and z coordinates to a .csv file. There is one line for each vertex. The file starts with a header (Static vertex), followed by number of elements in the file and then "end_header", before the vertex data is listed.

Parameters:

path : *str, optional*

Specifies the location to the written .csv file, excluding filename. Default is None, so the file will be saved to the current directory. The filename is "static_vertices".

Source code

```
def WriteStaticVertices(self, path=None):
    Header = "Static vertex \n{}\nend_header \n".format(self.N)

    if path == None:
        outfilename = "static_vertices.csv"
    else:
        outfilename = path + "/" + outfilename

    with open(outfilename, "w") as outfile:
        outfile.write(Header)
        for index, coordinate in enumerate(self.vertices):
            outfile.write("{},{},{},{} \n".format(index, coordinate[0], coordinate[1], coordinate[2]))
        return None
```

Example

$$\text{vertices} = \begin{bmatrix} -1 & 1 & -3 \\ -1 & -1 & -3 \\ 0 & 0 & -1 \\ 1 & -1 & -3 \end{bmatrix} \xrightarrow{\texttt{WriteStaticVertices(path=None)}}$$

Static vertex
4
end_header
$0, -1.0, 1.0, -3.0$
$1, -1.0, -1.0, -3.0$
$2, 0.0, 0.0, -1.0$
$3, 1.0, -1.0, -3.0$

PlyModel.WriteStaticPolygons(self, path=None)

The method writes the polygon number, followed by the vertex number of each of the three polygon vertices in right-hand oriented order to a .csv file. There is one line for each polygon. The file starts with a header (Static polygon), followed by number of elements in the file and then "end_header", before the polygon data is listed.

Parameters:

path : *str, optional*

Specifies the location to the written .csv file, excluding filename. Default is None, so the file will be saved to the current directory. The filename is "static_polygons".

Source code

```python
def WriteStaticPolygons(self, path=None):
    Header = "Static polygon \n{}\nend_header \n".format(self.M)

    if path == None:
        outfilename = "static_polygons.csv"
    else:
        outfilename = path + "/" + outfilename

       with open(outfilename, "w") as outfile:
        outfile.write(Header)
        for index, vertices in enumerate(self.faces):
            outfile.write("{},{},{},{} \n".format(index, vertices[0], vertices[1], vertices[2]))

        return None
```

Example

$$\text{faces} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 2 & 1 \\ 3 & 4 & 2 \\ 0 & 3 & 1 \end{bmatrix} \xrightarrow{\texttt{WriteStaticPolygons(path=None)}} \begin{array}{l} \text{Static polygon} \\ 4 \\ \text{end\_header} \\ 0,0,1,2 \\ 1,3,2,1 \\ 2,3,4,2 \\ 3,0,3,1 \end{array}$$

REFERENCES

[1] T. Akenine-Möller et al. *Real-Time Rendering.* 4th ed. 2018. Chap. 16.3, pp. 691–694. ISBN: 1138627003.

[2] Tomas Möller and Ben Trumbore. “Fast, Minimum Storage Ray-Triangle Intersection”. In: *Journal of Graphics Tools* 2.1 (1997), pp. 21–28. URL: https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf.

[3] Source code for `ReadPly` function. https://github.com/daavoo/pyntcloud/blob/master/pyntcloud/io/ply.py.

[4] Wikipedia contributors. *Sparse matrix — Wikipedia, The Free Encyclopedia.* 2020. URL: https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=968422230.