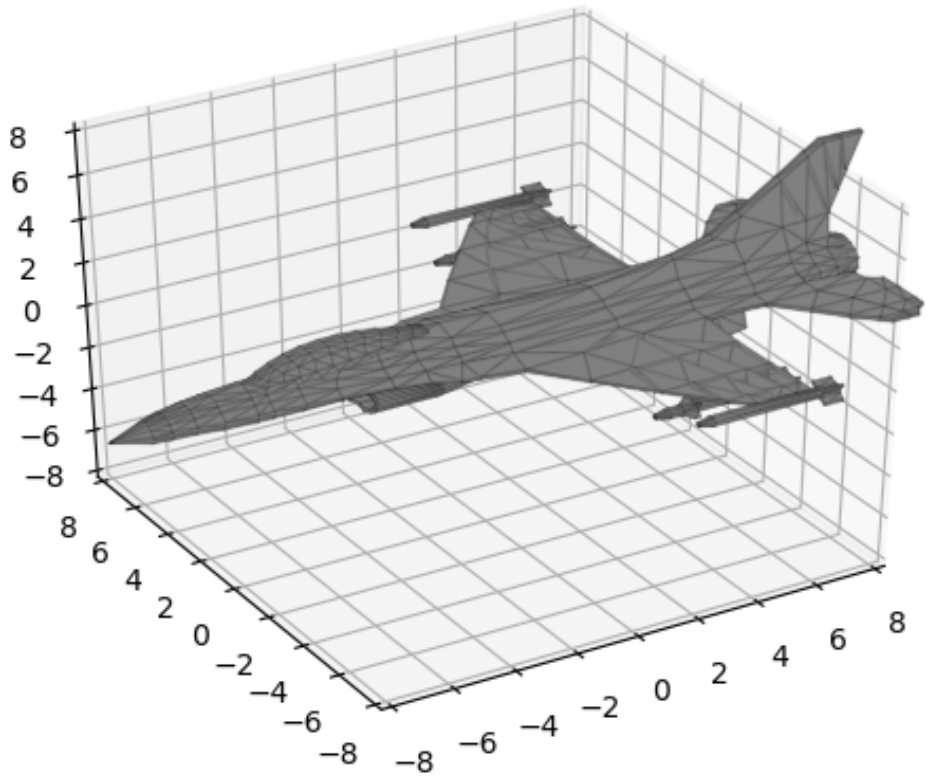
Code Documentation for PlyModel

A class developed in python for verifying a variety of geometric features in a 3D triangular polygon model



Summer project 2020

Maria Linea Horgen

I. GENERAL INFORMATION AND BASICS

The following document contains code documentation for the class `PlyModel`, which is developed for verifying a variety of geometric properties in a 3D triangular polygon model in the polygon file format (.ply). The 3D model is assumed to be constructed in a separate 3D modelling program, e. g. blender or maya. After applying the class methods to the model the desired outcome would be one of two: the model is checked and ready for rendering, or that enough feedback regarding the model has been provided to the user, so that the model can be corrected. Some of the class methods are designed to make alterations to the data set, while others simply uncover problems that needs to be handled externally. This document is also created in the attempt to give some insight in the thoughts and ideas that have gone into constructing this class. A brief section is included at the end regarding future work and some final comments to my own work.

The class has some dependency on other modules, especially `NumPy`, see [2] for the complete source code. The instance variables are all initially declared with value `None`. The class methods are constructed as a void function, or so called nonvalue-returning function, were they alter already existing instance variables, i. e they do not create new variables.

A. Instance variables

Below follows a table of all the instance variables and the data they hold. As mentioned, they have a `None`-value 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	undirected/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.manifold	bool	if True, the mesh is manifold
self.non_manifold	Numpy array	holds the polygons that are not connected to two edges
self.more_neighbours	Numpy array	holds the polygons that have more than three neighbours
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 16 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. `Manifold`
    Runs a simple check whether the object represented in the data set is manifold.

12. `FlippPolygons`
    Checks if the polygon orientation is consistent among adjacent polygons. If not, the adjacent polygon is flipped.

13. `RayCasting`
    Verifying that all the polygons are oriented outwards through ray casting.

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]. The instance variables `self.N` and `self.M` are assigned their data in `ReadPly`. The class further contains three methods which calls a selection of the methods above that are meant to inspect specific aspects of the model; loose geometry, continuous groups within the data set and polygon orientation. They are listed at the end of section II.

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 and if its broken the cosine or sine are hard coded to zero.

This method is implemented solely for visualization purposes.

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 has to be decremented by one to ensure that the indices in the faces array are correct. 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

The row highlighted in red is a single vertex, and do not appear in the faces array.

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

PlyModel.UniqueVertices(self)

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

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 in form of a dictionary. If vertex j is a duplicate of vertex i, the entry in the hash table would be $= \{"i" : [j]\}$.

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 multiple 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 with a value greater than N) After all the duplicated indices are identified, them being all the integers that are equal to the constant, 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

```
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

The colored entries represent duplicates.

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

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

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 to 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

```
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

The row highlighted in red is a polygon with no area.

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

PlyModel.HalfEdges(self)

Calculates the half-edges between the vertices in each polygon, and fills the instance variables `self.half_edges` and `sorted_edges` with data. 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

```
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 undirected/sorted half-edges this method identifies adjacent polygons, and stores the information in an adjacency matrix, `self.adjacency_mat`. 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, $A_{ij} = 0$. Here i and j are the polygon numbers extracted from the third column in the sorted half-edge array when the two first column elements match between two polygons.

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 to compute $(3M-1)^2$ iterations, compared to $(3M-1)$ iterations with the sorted edges.

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, e. g compressed row storage [4] or a hash table. If this is to be done, some of the remaining methods that depend on the adjacency matrix has to be rewritten with respect to the new storage format.

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

The two half-edges colored red are a matching set of half-edges, indicating two neighbouring polygons.

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

PlyModel.RemoveSingles(self)

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

The singles are stored in a list, so that they can be removed from the vertex-, faces-, 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 half-edge arrays, 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 adjacency matrix is updated.

Parameters: None

Source code

```
def RemoveSingles(self):

    single_poly = []

    for i in range(self.M):
        if not np.any(self.adjacency_mat[i,:] == 1):
            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


            # deleting the half edges in the undirected half edge array
            polygon_num = self.sorted_edges[np.where(self.sorted_edges[:,:-1] == rm_vertices[0])[0][-1]][-1]   # associated polygon
     number of the single polygon
```

```
            indices = np.where(self.sorted_edges[:,-1] == polygon_num)
            decrement = np.sort(self.sorted_edges[indices,0])[0]
            decrement = list(decrement[::-1])
            self.sorted_edges = np.delete(self.sorted_edges, indices, axis=0)

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

            decrement = np.where(self.sorted_edges[:,-1] > polygon_num)
            self.sorted_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
        return None
```

Example in 2D

The entries colored red belong to the single polygon.

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

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

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

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 polygon numbers are zipped together into a list, and iterated over. The idea is that the polygon with the highest index in the pair is given the index belonging to the remaining polygon, and all other higher indices are decremented by one. An integer array from $0, ..., M-1$ keep tracks over the index values. If polygon pair (i, j) is being evaluated, and $i > j$, i is set equal to j, and all indices which are greater than i is 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 belong to the same group.

After all neighbouring pairs are iterated through, the number of unique elements in the integer array 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 (`self.groups_index`) and one storing the faces (`self.groups_faces`).

Parameters: None

Source code

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

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. An exception occurs if bowtie geometry is involved, so to be on the safe side the relation

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

where V is the number of vertices, F is the number of faces and E is number of edges, should be true for a manifold mesh.

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. Others are easiest dealt with in the modelling program, such as internal faces, lack of thickness and holes. This method identifies the polygons that are constructed by edges that are not shared by exactly two faces, polygons with more than three neighbours and if multiple faces share a common vertex but no edges (bowtie geometry).
* If the method finds that the model is manifold, the problem with opposite normals can be resolved with the methods `PlyModel.FlippPolygons` and `PlyModel.RayCasting`.

To identify the edges that are not shared by two polygons the appearance of each half-edge is counted in the `sorted_edges` array. To keep track over the number of appearances for each half-edge, an array of zeros with the length of the number of edges (i. e. unique undirected half-edges), is incremented by one each time the edge appears in the undirected half-edge array. The array is named `HalfEdgeCounter`, and from its data it is possible to extract the polygon faces that are constructed from half-edges with a count $\neq 2$. The number of half-edges shared between one, two or more than two polygons are printed to the terminal window, and the polygons are stored in `self.non_manifold`.

The adjacency matrix can be converted into an adjacency list, and from this it is trivial to identify polygons with more than three neighbours. Since all adjacent polygons are listed, one can simply check if any of the polygons has a list entry that exceeds three elements. The polygons with more than three neighbours are stored in `self.more_neighbours`.

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 local hash table, were the index is the key, and printed to the terminal window.

All the problematic elements regarding the geometry are printed out in the terminal window. It is also possible to visualize these elements by setting `visualize=True`. The polygons that contain edges that are not connected to exactly two faces are colored red, while the non-problematic polygons are green. Polygons with more than three neighbours has a ray pointing out/in of the polygon (depending on orientation). Lastly, vertices shared among groups are colored magenta.

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 plot and highlight problematic geometry. Polygons that contain edges that are not connected to exactly two polygons are colored red, while the non-problematic polygons are green. Polygons with more than three neighbours has a ray pointing out/in of the polygon (depending on orientation). Vertices shared among groups are colored magenta. The default is False.

Source code

```
def Manifold(self, x_lim=[], y_lim=[], z_lim=[], multiplier=1, visualize=False):
    constant = self.N + self.M - len(self.half_edges)*0.5

    # checking if each half-edges is connected to 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
    i = 0

    while i < len(self.sorted_edges):
        half_edge1 = self.sorted_edges[i,:-1]
        index = int(np.where((Edges == half_edge1).all(axis=1))[0])
        counter = 1                                                  # keeps track over number of times the half-edge has appeared
        if i == len(self.sorted_edges)-1:
            HalfEdgeCounter[index] = counter
            i += 1
        for j in range(i+1, len(self.sorted_edges)):                 # iterating over the sorted half-edges until a non-matching edge is
     reached
            half_edge2 = self.sorted_edges[j,:-1]
            if np.all(half_edge1 == half_edge2):
                counter += 1
                if j == len(self.sorted_edges)-1:
                    HalfEdgeCounter[index] = counter
                        i += counter
            else:
                HalfEdgeCounter[index] = counter
                i += counter
                break

    self.manifold =  True                       # initial assumption that the mesh is manifold

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

    if appear_once > 0 or appear_more > 0:
        print("Some edges are not shared by exactly two faces")
        self.manifold = False

    if self.manifold and constant == 2:
        print("The model is manifold")
    else:
        print("The model is non-manifold")

        # identifying shared vertices among the groups
        shared_vertices = {}
```

```
        for i in range(self.N):
            sharing_groups = []
            for key in self.groups_faces:
                values = self.groups_faces[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 with more than three neighbours

        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)

        more_neighbours = []
        for key, values in adjacency_list.items():
            if len(values) > 3:
                more_neighbours.append(int(key))

        self.more_neighbours = np.array(more_neighbours)

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

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

        if 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))
        elif self.manifold == True and constant != 2:
            for key, values in shared_vertices.items():
                print("Vertex {} is shared between {}".format(key, values))
        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")

            for i in range(self.M):
                vtx = self.faces[i]
                tri = art3d.Poly3DCollection([self.vertices[vtx]*multiplier])
                if i in self.non_manifold:
                    tri.set_color((0.811, 0.149, 0.196,0.8))
                else:
                    tri.set_color((0.188, 0.909, 0.329,0.8))
                if i in self.more_neighbours:
                    x1, y1, z1 = self.vertices[self.faces[i][0]]
                    x2, y2, z2 = self.vertices[self.faces[i][1]]
                    x3, y3, z3 = self.vertices[self.faces[i][2]]
                    centroid = (np.array([((x1 + x2 + x3) / 3), ((y1 + y2 + y3) / 3), ((z1 + z2 + z3) / 3)]))*multiplier
                    X, Y, Z = centroid
                    U, V, W = (self.normal_vectors[i,1:]/np.linalg.norm(self.normal_vectors[i,1:]))*multiplier

                    ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.1)

                tri.set_edgecolor("black")
                ax.add_collection3d(tri)

            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, s=10**2, color="m", marker="o")

        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

The highlighted face indices represents a vertex shared among two polygons that do not share an edge.

$$\text{vertices} = \begin{bmatrix} -1 & 1 & -3 \\ -1 & -1 & -3 \\ 0 & 0 & -1 \\ 1 & -1 & -3 \\ 1 & 1 & -3 \\ -1 & -1 & 1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 1 & 1 \\ -1.8 & -0.5 & 1.9 \\ -1.8 & 1.5 & 1.9 \end{bmatrix} \quad \text{faces} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 2 & 1 \\ 3 & 4 & 2 \\ 0 & 3 & 1 \\ 5 & 6 & 2 \\ 6 & 7 & 8 \\ 0 & 2 & 4 \\ 7 & 6 & 5 \\ 8 & 7 & 2 \\ 2 & 7 & 5 \\ 2 & 6 & 8 \\ 4 & 3 & 0 \\ 5 & 6 & 9 \\ 10 & 9 & 6 \end{bmatrix}$$

$$\text{groups} = \begin{Bmatrix} \text{"group1"}: & [[0 & 1 & 2], \\ & [3 & 2 & 1] \\ & [3 & 4 & 2] \\ & [0 & 2 & 4] \\ & [4 & 3 & 0] \\ & [0 & 3 & 1]] \\ \text{"group2"}: & [[5 & 6 & 2] \\ & [6 & 7 & 8] \\ & [7 & 6 & 5] \\ & [8 & 7 & 2] \\ & [2 & 7 & 5] \\ & [2 & 6 & 8]] \\ & [5 & 6 & 9]] \\ & [10 & 9 & 6]] \end{Bmatrix}$$

```
Manifold([-3,3], [-3,3], [-3,1], multiplier = 1, visualize=True)
```

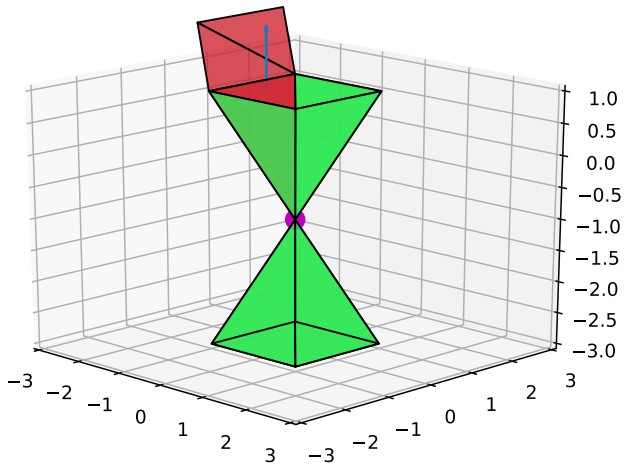


Figure 1. The figure displays a non-manifold object with multiple faces are sharing a common vertex (in magenta) but no edge and the polygons that are connected to an edge that with more than two polygons (the red faces). The face connected to the ray has more than three neighbouring polygons.

Output in terminal

```
The model is non-manifold
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.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.

It follows from the algorithm 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).

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

An important notice is that this method expects a manifold mesh. In the case of an non-manifold (i. e. not watertight) mesh, the method returns an erroneous result.

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

The highlighted row in the faces array is left-hand oriented.

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

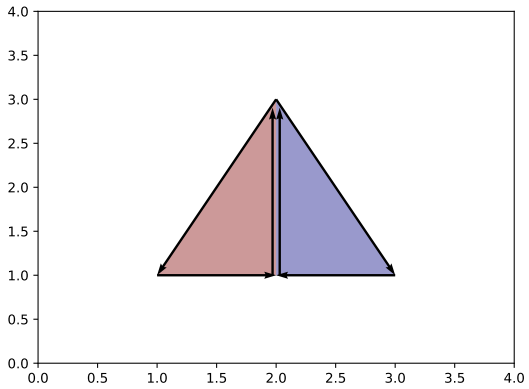


Figure 2. The figures display the orientation of two polygons, where the polygon to the right is left-hand oriented and the one to the left is right-hand oriented. They share a half-edge were the direction of traversal is the same for both polygons.

$\xrightarrow{\texttt{FlippPolygons}}$

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

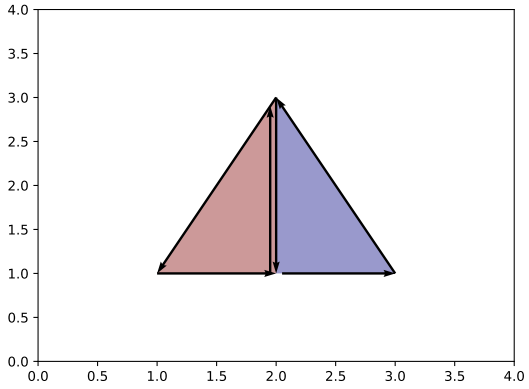


Figure 3. The figure displays two polygons that are right-hand oriented.

PlyModel.RayCasting(self)

The method determines if all the polygons are facing outwards or inwards. In the case where the polygons 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 `FlippPolygons` method has to be applied in advance. The algorithm is based on casting a ray from the centroid of each polygon and count the number of times it intersects with a polygon face. The algorithm is implemented in a static method called `RayTriangleIntersect`. If the number of intersections is odd the polygon is oriented inwards, and facing out when the number is even.

The rays are casted from the centroids of all polygons, so these are all computed and stored in a local array. The data set is of a type which makes it a troublesome task to determine which of the polygon faces are opposite of one another, so the ray casting is performed in a double for-loop inside an outer for-loop which iterates over all the continuous groups. Polygon i is casting a ray and the algorithm checks whether polygon j is intersected for $i, j = 0, ..., N-1$ and $i \neq j$. In the outer for-loop the centroid and direction of the ray are loaded into variables that are sent to the ray casting algorithm. In the inner for-loop the coordinates to all the vertices belonging to plygon j are loaded and sent into `RayTriangleIntersect`.

Number of intersections are stored in an array, were the index corresponds with the polygon number. At this point the polygon mesh is topological consistent, so it is sufficient to check if one of the polygons have zero intersections. In this case the mesh is faced outwards. If an inward orientation is detected, all the polygons are flipped and the half-edges and normal vectors are recalculated.

Since the method relies on `FlippPolygons`, the method expects a manifold mesh. In the case of an non-manifold (i. e. not watertight) mesh, the method returns an erroneous result.

Parameters: None

PlyModel.RayTriangleIntersect(self, origin, direction, v0, v1, v2)

The ray parametric equation is

$$P = O + tR, \tag{3}$$

where P is the intersection point between the ray and polygon plane, O is the ray's origin, R its direction (which is the same as the direction of the normal vector belonging to the polygon from where the ray was cast from) and t is the distance from O to P. Remembering that the equation of a plane can be expressed as

$$Ax + By + Cz + D = 0, \tag{4}$$

where A, B and C are the components of the normal vector $\vec{n}$ to the plane, and D is the distance from the origin (0,0,0) to the plane, and x, y and z are coordinates to any arbitrary point P_0 that lies in the plane. Combining eq. (3) and eq. (4) with $P_0 = v_0$ yields the following expression for t,

$$t = \frac{\vec{n} \cdot (v_0 - O)}{\vec{n} \cdot R}, \tag{5}$$

where v_0 is one of the vertices belonging to the polygon that span the plane. With an expression for t, we can now compute the intersection point P.

Before determining whether P lies inside the plane, two special cases has to be checked. If the ray and polygon plane are parallel, the ray will not intersect and the plane's normal vector will be perpendicular to the ray. Meaning that the dot product in the denominator in eq. (5) is equal to zero. Division by zero is undefined and will cause a computational error. By first computing $\vec{n} \cdot R$ we can circumvent this potential error, and if its close to zero, i. e. below an lower limit $\varepsilon = 10^{-7}$, the algorithm returns `False`.

If the plane and ray are not parallel, a value for t is obtained. In the case were $t < 0$, the intersection point P will be in the opposite direction from the ray's direction. So if t is negative the algorithm returns `False`. The dot product between R and the vector $(P - O)$ are also calculated to verify the direction of the camera. If its negative the results is `False`.

Lastly the algorithm determines if the intersection point P lies inside or outside the triangle. If it lies inside, an intersection occurs and the method returns `True`. The first step is to calculate the cross product between the edge $(v_n - v_m)$ and the vector $(P - v_m)$ for all the three edges belonging to the polygon that span the plane. Each of the resulting vectors are then dotted with $\vec{n}$. If the result is negative the point lies on the outside and the algorithm returns `False`.

Parameters:

origin : *float*
3D cartesian coordinates to the ray's origin.

direction : *float*
Normalized direction of the ray.

v0 : *float*
3D cartesian coordinates to the first vertex belonging to the polygon that spans the plane.

v1 : *float*
3D cartesian coordinates to the second vertex belonging to the polygon that spans the plane.

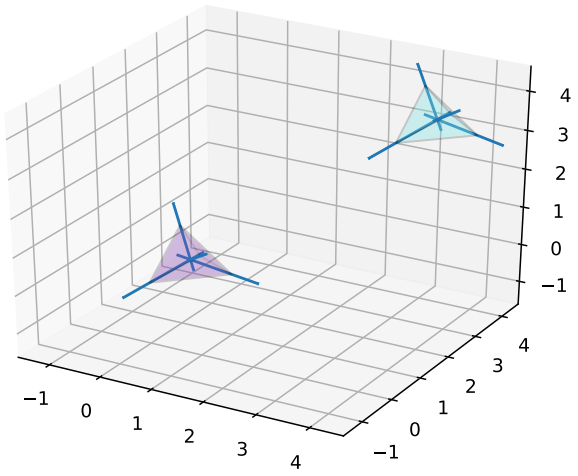
v2 : *float*
3D cartesian coordinates to the third vertex belonging to the polygon that spans the plane.

Returns: *bool*
If True an intersection occurs, if False, no intersection.

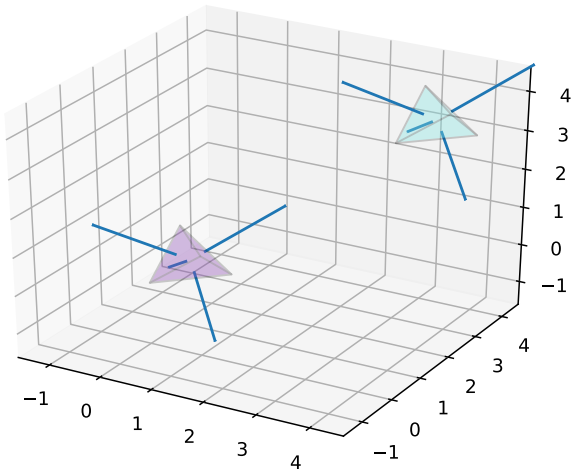
Source code

```
def RayCasting(self):
    # calculating the centroids for each polygon
    centroids = np.zeros([self.M,3])

    for i in range(self.M):
        x1, y1, z1 = self.vertices[self.faces[i][0]]
        x2, y2, z2 = self.vertices[self.faces[i][1]]
        x3, y3, z3 = self.vertices[self.faces[i][2]]

        centroids[i] = np.array([((x1 + x2 + x3) / 3), ((y1 + y2 + y3) / 3), ((z1 + z2 + z3) / 3)]) + (0 * self.normal_vectors[i,1:]/np.
     linalg.norm(self.normal_vectors[i,1:]))     # adding a small displacement

    Recalculate = False                                     # if True after the iterations, the half-edges and normal vectors are
      recalculated
    intersections = {}

    for key, values in self.groups_index.items():
        intersect = np.zeros(len(values[0]))
        for i in values[0]:
            origin = centroids[i]                           # origin of the ray
            direction = self.normal_vectors[i,1:]/np.linalg.norm(self.normal_vectors[i,1:])
            for j in range(len(values[0])):
                if i != j:
                    v0 = np.array([self.vertices[self.faces[j][0]][0], self.vertices[self.faces[j][0]][1], self.vertices[self.faces[j
     ][0]][2]])
                    v1 = np.array([self.vertices[self.faces[j][1]][0], self.vertices[self.faces[j][1]][1], self.vertices[self.faces[j
     ][1]][2]])
                    v2 = np.array([self.vertices[self.faces[j][2]][0], self.vertices[self.faces[j][2]][1], self.vertices[self.faces[j
     ][2]][2]])

                    result = self.RayTriangleIntersect(origin, direction, v0, v1, v2)          # if an intersection occurs, the function
      return True

                    if result:
                        intersect[i] += 1


        intersections[key] = intersect


    for key, values in intersections.items():
```

```
        if not np.any(values == 0):                                    # since the model is topological consistent, if facing outwards
     at least one polygon should have zero intersections
            for j in self.groups_index[key][0]:
                self.faces[j][1], self.faces[j][2] = self.faces[j][2], self.faces[j][1]       # flipping polygons
                Recalculate = True
            print("Detected inward orientation for {}".format(key))


    if Recalculate:
        print("Recalculating the half-edges and normal vectors")
        self.HalfEdges()
        self.NormalVectors()

        print("All polygons are flipped and has an outwards orientation")
    else:
        print("All polygons face outwards")
    return None

@staticmethod
def RayTriangleIntersect(origin, direction, v0, v1, v2):

    # calculates normal vector to the plane
    v0v1 = v1-v0
    v0v2 = v2-v0
    N = np.cross(v0v1,v0v2)

    eps = 1e-7
    NdotRayDirection = np.dot(N, direction)

    if NdotRayDirection > -eps and NdotRayDirection < eps:            # the ray and plane are parallel
        return False


    t = (np.dot(N,v0-origin))/NdotRayDirection
    if t < 0:                                                         # the polygon is behind the ray
        return False

    P = origin + t*direction

    DotProduct = np.dot(direction, P-origin)
    if DotProduct < eps:                                             # if negative, the camera direction is backward
        return False

    edge0 = v1-v0
    vp0 = P-v0
    C = np.cross(edge0,vp0)
    if np.dot(N,C) < 0:
        return False

    edge1 = v2-v1
    vp1 = P-v1
    C = np.cross(edge1,vp1)
    if np.dot(N,C) < 0:
        return False

    edge2 = v0-v2
    vp2 = P-v2
    C = np.cross(edge2,vp2)
    if np.dot(N,C) < 0:
        return False

    return True
```



(a)



(b)

Figure 4. **a)**: The figure displays two polygons with inward orientation. The rays all intersect with the polygon face opposite of their origin. Another way to determine the inward orientation is to notice that all of the rays intersect each other in the tetrahedrons center of mass. **b)**: After calling **RayCasting** the rays no longer intersect the polygon faces, and are all facing outward.

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

The method plots the vertices and faces in a 3D-plot, using the matplotlib library. Two highlightning options are available: groups or polygons, it is not possible to choose both in one calling. The method can also plot the surface normal vectors for each polygon, independently of highlightning choice.

If the parameter `group` has the default value `None`, and the data set is one continuous group, the plot will be unicolored, if not each group will have a different color. If the groups are not yet identified, they object will be plotted as one continuous group.

In the case where `group` $\neq$ `None`, the specified groups are visualized, this can be a list of groups or just one specific group.

When the `polygons` parameter has a non-`None`, the polygons specified (here corresponding with their index in the face array) are plotted in black. The polygons specified can either be a list of polygons or a single polygon.

In the case were the `normals` parameter is `True` and the groupwise visualization is chosen, each surface normal vector is plotted for the polygons in the specified groups. In the case where polygons are highlighted, only normal vectors for the chosen polygons are displayed.

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.

polygons : *int or list, optional*

Specify which polygon(s) to visualize in a different color (black). If this is specified the group wise visualization is not possible. Default is None.

normals : *bool, optional*

Plots normal vectors if True. Default is False.

Source code

```
def Visualize(self, x_lim=[], y_lim=[], z_lim=[], multiplier = 1, group=None, polygons=None, normals=False):
    if group != None and polygons != None:
        raise ValueError("Both parameters 'group' and 'polygons' are specified, one has to be None.")

    fig = plt.figure()
    ax = fig.add_subplot(projection="3d")

    v = self.vertices*multiplier

    centroids = np.zeros([self.M,3])

    for i in range(self.M):
        x1, y1, z1 = self.vertices[self.faces[i][0]]
        x2, y2, z2 = self.vertices[self.faces[i][1]]
```

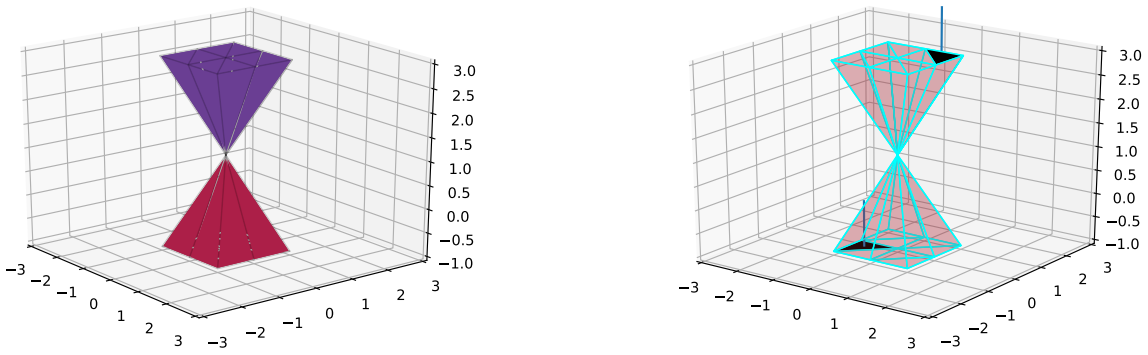
```
        x3, y3, z3 = self.vertices[self.faces[i][2]]

        centroids[i] = (np.array([((x1 + x2 + x3) / 3), ((y1 + y2 + y3) / 3), ((z1 + z2 + z3) / 3)]) + (0.01 * self.normal_vectors[i,1:]/
     np.linalg.norm(self.normal_vectors[i,1:]))*multiplier


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

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


        if normals:
            for i in range(self.M):
                X, Y, Z = centroids[i]
                U, V, W = (self.normal_vectors[i,1:]/np.linalg.norm(self.normal_vectors[i,1:]))*multiplier

                ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.01)

            color = (r, g, b,0.2)
        pc = art3d.Poly3DCollection(v[f], facecolors=color, edgecolor=(0,0,0,0.1))
        ax.add_collection(pc)
    elif group != None and polygons == 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)

            if normals:
                for i in self.groups_index["group{}".format(group)]:
                    for j in i:
                        X, Y, Z = centroids[j]
                        U, V, W = (self.normal_vectors[j,1:]/np.linalg.norm(self.normal_vectors[j,1:]))*multiplier
                        ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.01)

                color = (r, g, b, 0.2)

            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)

                if normals:
                    for k in self.groups_index["group{}".format(group[i])]:
                        for j in k:
                            X, Y, Z = centroids[j]
                            U, V, W = (self.normal_vectors[j,1:]/np.linalg.norm(self.normal_vectors[j,1:]))*multiplier

                            ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.01)
                    color = (r, g, b,0.2)

                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)))
    elif polygons == None:
        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 normals:
            for i in range(self.M):
```

```
            X, Y, Z = centroids[i]
            U, V, W = (self.normal_vectors[i,1:]/np.linalg.norm(self.normal_vectors[i,1:]))*multiplier

            ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.01)
        color = (r,g,b,0.2)



elif polygons != None and group == None:
    r = np.random.rand()
    b = np.random.rand()
    g = np.random.rand()
    color = (r, g, b, 0.2)
    if isinstance(polygons, int):

        for i in range(self.M):
            vtx = self.faces[i]
            tri = art3d.Poly3DCollection([self.vertices[vtx]*multiplier])
            if i == polygons and normals:
                tri.set_color("black")
                X, Y, Z = centroids[i]
                U, V, W = (self.normal_vectors[i,1:]/np.linalg.norm(self.normal_vectors[i,1:]))*multiplier

                ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.01)
            elif i == polygons and not normals:
                tri.set_color("black")
            else:
                tri.set_color(color)
            tri.set_edgecolor("cyan")
            ax.add_collection3d(tri)

    elif isinstance(polygons, list):

        for i in range(self.M):
            vtx = self.faces[i]
            tri = art3d.Poly3DCollection([self.vertices[vtx]*multiplier])
            if i in polygons and normals:
                tri.set_color("black")
                X, Y, Z = centroids[i]
                U, V, W = (self.normal_vectors[i,1:]/np.linalg.norm(self.normal_vectors[i,1:]))*multiplier*2

                ax.quiver(X,Y,Z,U,V,W,arrow_length_ratio=0.01)
            elif i in polygons and not normals:
                tri.set_color("black")
            else:
                tri.set_color(color)
            tri.set_edgecolor("cyan")
            ax.add_collection3d(tri)

    else:
        raise TypeError("polygons parameter is neither list or int, but {}".format(type(polygons)))



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

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

$\xrightarrow{\texttt{Visualize}}$



(a) (b)

Figure 5. **(a):** The figure displays how the `PlyModel.Visualize` method visualize an object made up of two continuous groups of triangular polygons. Here the method call reads: `Visualize([-3,3], [-3,3], [-1,3], multiplier = 1, group=None, polygons=None, normals=False)`. **(b)**: Here the same object is displayed, but now polygons 3 and 7 are highlighted with normals. The method call is `Visualize([-3,3], [-3,3], [-1,3], multiplier = 1, group=None, polygons=[3,7], normals=True)`. 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

$$\text{vertices} = \begin{bmatrix} -1 & 1 & -3 \\ -1 & -1 & -3 \\ 0 & 0 & -1 \\ 1 & -1 & -3 \end{bmatrix} \xrightarrow{\texttt{WriteStaticVertices(path=None)}} \begin{array}{l} \text{Static vertex} \\ 4 \\ \text{end\_header} \\ 0, -1.0, 1.0, -3.0 \\ 1, -1.0, -1.0, -3.0 \\ 2, 0.0, 0.0, -1.0 \\ 3, 1.0, -1.0, -3.0 \end{array}$$

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

```
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

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

PlyModel.LooseGeometry(self)

A collection of class methods that ensures that the model contains no loose geometry such as single vertices, duplicated polygons or vertices.

Parameters: None

Source code

```
def LooseGeometry(self):
    self.SingleVertex()
    self.UniqueVertices()
    self.UniquePolygons()

    return None
```

PlyModel.GroupIdentification(self)

Calls methods that contribute to identify the continuous groups within the data set.

Parameters: None

Source code

```
def GroupIdentification(self):
    self.HalfEdges()
    self.AdjacentPolygons()
    self.RemoveSingles()
    self.FindingGroups()

    return None
```

PlyModel.PolygonOrientation(self)

Calls the assortment of methods necessary to determine whether the geometry is manifold or not, and in the case of manifold geometry ensures that the polygons are all faced outward.

Parameters:

vis : *bool, optional*

Corresponds with the `visualize` parameter belonging to the method `PlyModel.Manifold`. If not specified, the default value is `False`.

Source code

```
def PolygonOrientation(self, vis=False):
    self.NormalVectors()
    self.Manifold(visualize=vis)

    if self.manifold:
        self.FlippPolygons()
        self.RayCasting()
    else:
        print("The mesh needs to be manifold before polygon orientation can be made consistent.")
    return None
```

III. FUTURE WORK

The aim of this project was to write a class that can verify that a polygonial model has the qualifications and geometrical features required for rendering. This result can be said to be partially obtained, but there are still room for improvement in several areas. One being the run time and optimization of the code. There is also some geometrical properties regarding the model that still needs to be checked, e. g. that all the polygons are convex and existence of self-intersecting polygons.

The ideal result would be to not only to have a class that verifies the model, but in the case of a flawed model, it would also be able to perform the necessary repair actions automatically. As of today, problems that arises during the verification process has to be manually fixed and corrected in an external 3D modelling program.

REFERENCES

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

[2] Link to Github repository with source code. https://github.com/marialinea/PlyModel.

[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.