Robust Point Membership For Polygonal Meshes

Iona Vincent

Master of Science

Computer Animation and Visual Effects

Bournemouth University

August 2014

Contents

List of Figures

List of Tables

List of Acronyms

AWPN	Angle Weighted Pseudo Normal
PIP	Point in Polyhedron
GWN	Generalised Winding Number
AFM	Advancing Front Mesh

Abstract

Boundary descriptions of solid shapes are used throughout computer graphics and are often represented using polygonal meshes. For such meshes, the ability to determine if a point is inside or outside the mesh is of great importance in both computer graphics and the fabrication industry. For instance it is necessary for 3D printing, collision detection, rendering and voxelisation. Existing techniques either compromise on efficiency or robustness, often resulting in the misclassification of points with the most time efficient methods. This thesis presents a new algorithm to classify a point as inside or outside a mesh. This is done through the careful combination of existing techniques and the introduction of an original algorithm. In this way we are able to produce a classification method robust to the usual mesh defects of self intersections and holes without the usual large time implications.

Acknowledgements

I would like to thank Mathieu Sanchez for his continued support and advice both on this project, and throughout the year. I would also like to thank Oleg Fryazinov and Prof. Alexander Pasko for their theoretical input and advice.

Thank you also goes to Mathieu Sanchez for granting permission to use his SDF library.

Chapter 1

Introduction

The Point in Polyhedron (PIP) test is one of the fundamental problems in both computer graphics and fabrication. Necessary for procedures as elementary as boolean operations to more complex collision detection, rendering and voxelization, it is a test that is frequently applied. For instance, in an animated sequence it may be necessary to determine if the tip of an object has penetrated the surface of another in each frame. This means an efficient and robust algorithm to classify points is often required.

The three most commonly used methods for the PIP test are ray casting (Requicha and Voelcker 1985), using the Angle Weighted Pseudo Normal (AWPN) (Baerentzen and Aanaes 2005) and a method utilising the GWN (Jacobson *et al.* 2013). Whilst it is clearly important that the techniques work for ideal cases, namely clean watertight meshes, consideration also needs to be taken for meshes that do not conform to such strict requirements. Holes, duplicate faces, non-manifold attachments and self-intersections often appear in meshes, and ideally classification methods should be able to cope with such situations.

This thesis presents an algorithm for the PIP test which combines the AWPN method with the GWN method through the introduction of a new unique algorithm. The aim is to produce a classification algorithm which is robust to the usual mesh defects, whilst being more time efficient than the GWN method. To achieve this the GWN technique is only utilised in ambiguous areas of the mesh and the AWPN elsewhere. In this way we aim to combine the time efficiencies of the AWPN with the robustness of the GWN.

In Chapter 2, we present an overview of existing methods to conduct the PIP test and the benefits and drawbacks of each. Particular attention is paid to their ability to deal with mesh defects.

In Chapter 3, the theory behind the GWN and AWPN methods is explained before the different types of mesh defects are outlined. The theory behind the key steps in the proposed algorithm is then explained.

In Chapter 4, the implementation details of the techniques used are presented. Pseudo code for the main algorithms is included, as well as explanations of the difficulties which arose during their implementation and the solutions used to overcome them.

In Chapter 5, we present the results of the algorithm. These are compared to the other main PIP algorithms.

In Chapter 6, we summarise the method presented and discuss its advantages and disadvantages. Potential future work is also outlined.

Chapter 2

Background

There are three key methods used to carry out the PIP test: ray casting, pseudo normals and the generalised winding number. In this section an overview of each method will be given, along with the situations in which each can be used, before outlining another alternative method of classification.

Ray Casting is the oldest and most widely used method to determine a point's location as inside or outside a polyhedra in computer graphics. At the most basic level, a ray is fired in a random direction from the point of interest to infinity and the number of intersections with the mesh are counted. An odd number of intersections denotes that the point is outside the mesh and an even number inside (Requicha and Voelcker 1985), as shown in Figure 2.1.

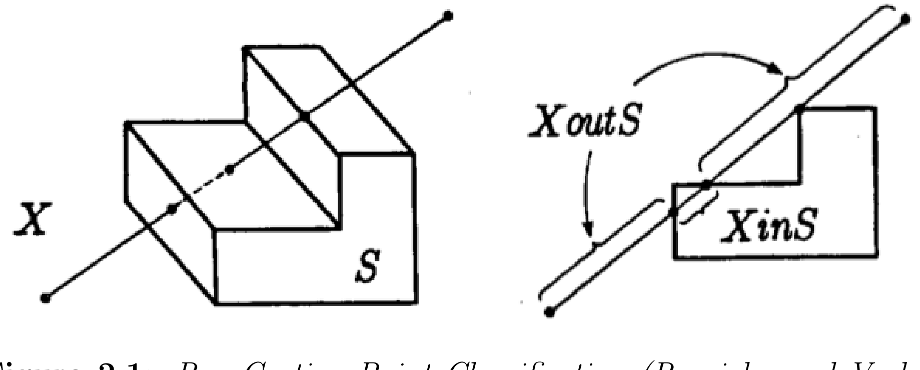


Figure 2.1: *Ray Casting Point Classification (Requicha and Voelcker 1985)*

Unfortunately if the cast ray hits the mesh at a vertex, edge or is collinear to an edge this method comes across difficulties (Requicha and Voelcker 1985). Similarly, if the ray passes through a hole in the mesh, or is itself within numerical precision of the boundary of the mesh, it may result in the incorrect classification of the point. However as the vast majority of rays will not hit any ambiguities, a simple solution to overcome this is to simply discard any rays that hit degeneracies (J. 1998). Alternatively multiple rays can be fired in different directions for each point, and the value that occurs most often used.

Another solution is to only take note of intersections that change the inside or outside parity. To establish the type of edge intersections, neighbourhoods can be introduced (Requicha and Voelcker 1985). Such neighbourhoods essentially highlight the areas containing material and can be represented by the signs of the normals and tangents at the edge and the edges that bound it, as shown in figure 2.2. For edges which are not 2-manifold, lists of neighbourhoods can be stored as opposed to just the one.

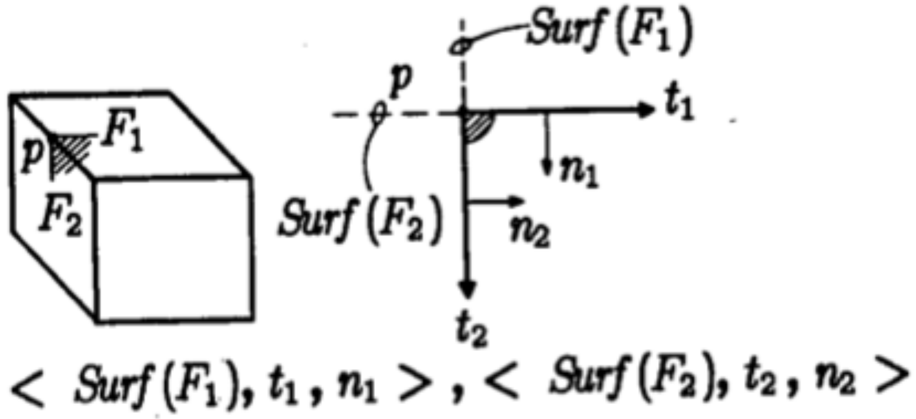


Figure 2.2: *3D Edge neigbourhoods denoting on which side of each edge the geometry lies (Requicha and Voelcker 1985)*

With these neighbourhoods established, the direction of the ray can be compared with the 3D edge neighbourhood, as shown in figure 2.3 to establish whether or not the ray enters, leaves or simply touches the mesh.

To resolve ambiguous intersections with vertices Requicha and Voel-

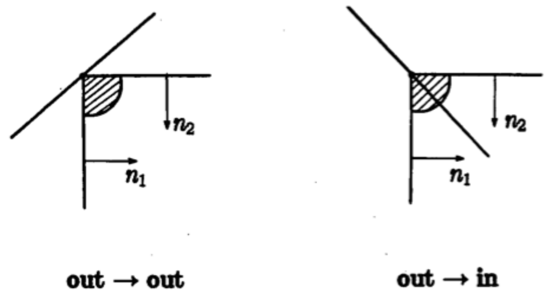


Figure 2.3: *Classification with edge neighbourhoods to a ray has entered the geometry or not (Requicha and Voelcker 1985)*

cker (1985) highlight a method in which they need not be directly taken into consideration. Instead, at any vertex singularity, the midpoint of the line segment either side of the point is found, and classified. These two classifications can then be used to determine the nature of the intersection, as shown in figure 2.4.

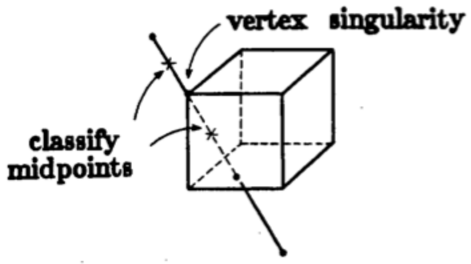


Figure 2.4: *Classifying vertices through the classification of the mid point of the ray either side of the vertex (Requicha and Voelcker 1985)*

If an acceleration structure is used, the time implications of this method are $\mathcal{O}(\log N)$, where N denotes the number of faces and otherwise it is linear. Whilst there are methods to reduce the classification ambiguity if the cast ray hits the mesh at a vertex, edge or is collinear to an edge, as highlighted above, the classification from such rays may still be incorrect. Similarly if a ray passes through a hole in the mesh, or is itself within numerical precision of the boundary of the mesh, it may result in a missclassification of the point.

Pseudo Normals on the other hand are a natural extension to face normals which have long been used to determine the inside and outside of closed, orientable and smooth surfaces. Unlike face normals though, pseudo normals are defined at edges and vertices where there is a lack of C^1 continuity (Baerentzen and Aanaes 2005).

Many pseudo normals have been proposed, however Andreas Bærentzen and Henrik Aanæs suggest that the AWPN presented independently by both Thürmer and Wüthrich [1998] and Sequin [1986] is best suited for the inside or outside classification of points. As the name suggests, the AWPN for a vertex is defined as the weighted average of the face normals of the surrounding triangles, where the weighting comes from the size of the incident angle for that triangle, as shown in figure 2.5.

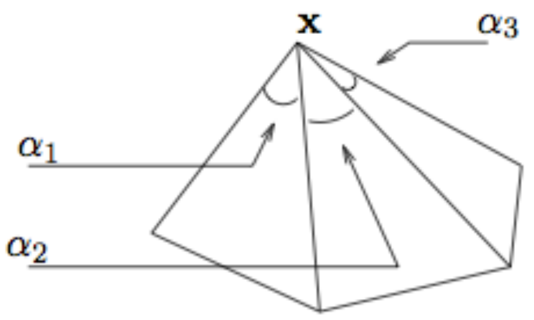


Figure 2.5: *Incident Angles α_1, α_2 and α_3 for vertex $\boldsymbol{x}$ (Baerentzen and Aanaes 2005)*

With the AWPN defined, the classification principle is then simple. For an arbitrary point **p**, and closest point **c** on the mesh, the inner product is taken between the the vector (**p** - **c**) and the pseudo normal at **c**. The sign of this inner product is then used to determine if **p** is inside or outside the mesh, with a negative sign denoting the latter and a positive the former. As the closest point on the mesh is required for this method, it has a complexity of $\mathcal{O}$(N) without an acceleration structure, where N is the number of triangles in the mesh. However if the closest point is already known this reduces the complexity to $\mathcal{O}(1)$. As with ray casting, the AWPN classification can only be used with closed, 2-manifold meshes. As a mesh comes within numerical precision of being non-manifold, or the distance between the point of interest and the mesh is with the numerical precision, numerical instabilities occur.

This method is also not able to close all meshes. Take for example a mesh shaped like a trumpet. Pseudo normals will create a discontinuity across the plane created by the end of the cone.

The Generalised Winding Number is a more recent approach to the PIP test. First proposed in two dimensions by Meister (1769), the winding number is a signed integer representing the total number of times a given curve travels anti clockwise around a point, as shown in Figure 2.6. A value of 1 signifies that the point is inside and 0 outside whilst 2 denotes a point which is doubly inside and -1 a point on a flipped face.

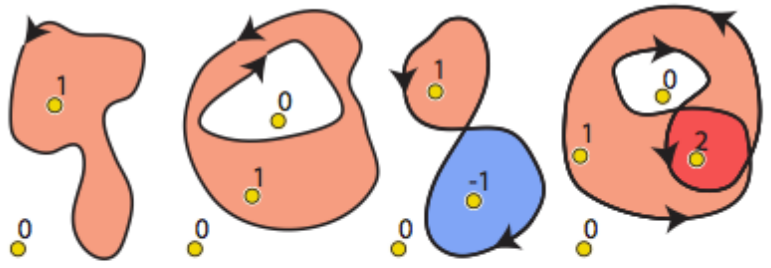


Figure 2.6: *Winding Number Segmentations (Jacobson* et al. *2013)*

By generalising this idea to three dimensions, so that the GWN analogously represents the signed number of times a surface wraps around a point, Jacobson *et al.* (2013) propose a method to segment meshes based on this value in a similar manner to the two dimensional case. This method allows for the classification of points within meshes containing non-manifold geometry or holes. However it is considerably slower than the previous two methods, with a time complexity of $\mathcal{O}(\mathrm{N})$ which cannot be improved with an acceleration structure. It also struggles to classify areas which contain thin, almost two dimensional features, such as clothing on figures, or the leaves on trees. Similarly, if a duplicate face encloses a region, it may cause a misclassification of the interior points as outside, as shown in figure 2.7. Whist Jacobson et al. propose tagging such sections prior to the winding number calculation so that they are not included, there is currently no algorithm to automate this process (Jacobson *et al.* 2013).

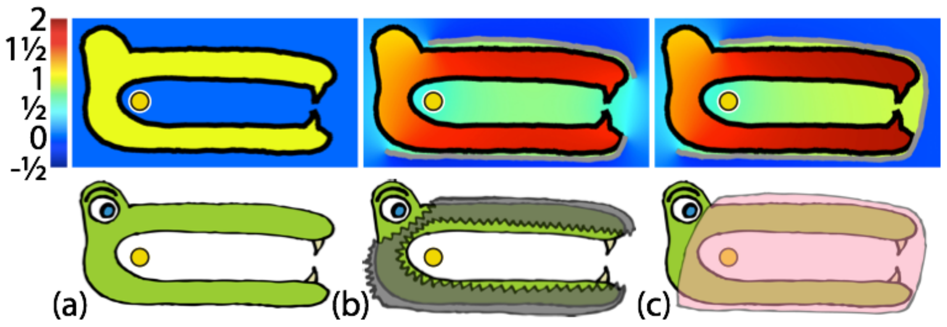


Figure 2.7: *Thin sheets cause (b) and (c) to have different inside/outside classifications (Jacobson* et al. *2013)*

Liu et al. have also proposed a method somewhat similar in nature to pseudo normals, however theirs contains an interesting preprocessing step (J. *et al.* 2010). First an octree is constructed using the bounding box of the polyhedra as the root before each vertex is inserted into the tree. If the pre-decided maximum number of vertices in a leaf node is reached, the tree is split. After this each cell is labelled as either black, if it is entirely outside the polyhedron, white if it is entirely inside and grey otherwise. In this way those points in unambiguous areas can be quickly classified without the need for further tests. For those in grey cells further tests are carried out similar to pseudo normals.

Chapter 3

Theory

When considering techniques for the PIP test it is important to take into account the robustness, complexity and behaviour for different types of input meshes, be it perfectly closed and watertight meshes, or non-manifold meshs with many holes and self intersections. This is particularly crucial due to the high prevalence of meshes containing defects such as holes, duplicate faces, flipped normals and self intersections which often go unnoticed in the design process. Similarly, computer aided design models often comprise multiple connected components which regularly lead to multiple holes and self intersections.

As outlined above for closed manifold meshes, the AWPN, GWN and ray casting methods are all able to correctly classify points, however as artefacts begin to appear in a mesh the reliability of each method varies considerably. It is clear from the previous chapter that the GWN is by far the most robust of the three algorithms, however the time implications of this method render it far from ideal on input geometry with a large number of faces. Instead a method combining the AWPN and GWN is proposed as follows. By identifying holes and self intersections within the input mesh, areas of potential classification ambiguity can be found. The GWN can then be utilised in these areas whilst the AWPN algorithm can be used on the rest of the mesh. In this way the robust nature of the generalised winding number algorithm can be utilised in areas of potential classification instability without a large time overhead when using it for the entire mesh.

In order to explain this new proposed method in more detail, the theory behind the AWPN and GWN PIP methods will now be presented. After this the usual defects in meshes will be covered as well as ways to overcome them before a more concrete explanation of the proposed algorithm is given.

3.1 Angle Weighted Pseudo Normals

As explained previously, the AWPN method classifies points by taking the inner product between the AWPN at the closest point on the mesh to the point **p** being classified, and the vector from this closest point to **p**. Specifically, for a point **x** on a closed, orientable 2-manifold mesh $\mathcal{M}$ in R^3 Euclidean space the AWPN $\mathbf{n}_\alpha$ is defied as follows:

$$\mathbf{n}_\alpha = \frac{\sum \alpha_i \mathbf{n}_\alpha}{||\sum \alpha_i \mathbf{n}_\alpha||} \tag{3.1}$$

where i runs over all the incident faces to **x** and α_i are the incident angles, as shown in figure 2.5.

With this definition of the AWPN Andreas Bærentzen and Henrik Aanæs propose it is then possible to classify points as inside or outside a mesh using the following equations:

$$n_\alpha \cdot (\mathbf{p} - \mathbf{c}) > 0 \quad \text{if } \mathbf{p} \text{ is outside the surface} \tag{3.2}$$
$$n_\alpha \cdot (\mathbf{p} - \mathbf{c}) < 0 \quad \text{if } \mathbf{p} \text{ is inside the surface} \tag{3.3}$$
$$n_\alpha \cdot (\mathbf{p} - \mathbf{c}) = 0 \quad \text{if } \mathbf{p} \text{ is on the surface} \tag{3.4}$$

where **p** is the point of interest and **c** the closest point on a closed, orientable 2-manifold mesh $\mathcal{M}$ in R^3 Euclidean space (Baerentzen and Aanaes 2005).

To avoid a linear time complexity when implementing this algorithm a spacial partitioning is required. Andreas Bærentzen and Henrik Aanæs suggest using a hierarchy of oriented bounding boxes (OBB) as an adequate compromise between the time and accuracy implications. To access the hierarchy a priority queue is stored, with the lower bound of the shortest distance to the mesh as the key.

This method however requires the meshes to be both 2-manifold and closed and thus cannot be used with non watertight or self intersecting meshes. Its robustness heavily relies on the manifold assumption too. If the mesh comes within numerical precision of becoming non-manifold, then it is possible for all the normals to be perpendicular to r and all the α_i to be small, both of which would cause the method to become numerically unstable. We can see this by considering the equation used to categorise points:

$$\sum \mathbf{r} \cdot \mathbf{n_i} \alpha_\mathbf{i} + \epsilon > 0 \tag{3.5}$$

For any values of epsilon which are of similar magnitude to $\sum \mathbf{r} \cdot \mathbf{n_i} \alpha_\mathbf{i}$, such as in the previous two cases, numerical instabilities will occur. Similarly, if the length of $\mathbf{r}$ is similar to that of the numerical precision, then $\mathbf{p}$ is essentially on the surface of the mesh, and thus the sign becomes irrelevant.

3.2 Generalised Winding Number

As outlined in the previous section, an alternative more recent approach is to generalise the winding number to three dimensions to create inside-outside mesh segmentations. As explained above, for a closed, self-crossing Lipschitz curve C in R^2, around a point $\mathbf{p}$, the winding number, $\Omega(\mathbf{p})$, is a signed integer representing the total number of times the curve travels anti clockwise around $\mathbf{p}$ (Rossignac *et al.* 2013). Without loss of generality if $\mathbf{p} = \mathbf{0}$, and C is parameterised using polar coordinates then:

$$w(\mathbf{p}) = \frac{1}{2\pi} \oint_c d\theta \tag{3.6}$$

where the result of 0 denotes that the point $\mathbf{p}$ is outside C and 1 inside.

In order to generalise this to R^3 we require the notion of a solid angle. For a point $\mathbf{p}$ in R^3 and a Lipschitz surface S is defined in spherical coordinates the solid angle Ω is defined as:

$$\Omega(\mathbf{p}) = \int \int_S sin(\phi) d\theta d\phi \tag{3.7}$$

where without loss of generality we let $\mathbf{p} = 0$ (Jacobson *et al.* 2013). Thus it measures the signed surface area of S, when projected onto a unit sphere centred at $\mathbf{p}$, as shown in figure 3.1. With this definition the winding number $\omega(\mathbf{p})$ in R^3 of a point $\mathbf{p}$ and a closed surface S is defined to be:

$$w(\mathbf{p}) := \frac{\Omega(\mathbf{p})}{4\pi} \tag{3.8}$$

Hence the winding number measures the signed number of times a surface wraps around a point [Baerentzen 2005].

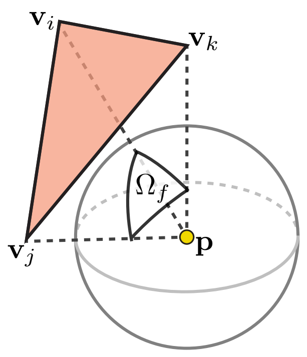


Figure 3.1: *Winding Number Generalisation to Three Dimensions(Jacobson* et al. *2013)*

If the original mesh is entirely free of ambiguities, then the generalised winding number produces an exact segmentation. That is, it will evaluate to integers which unambiguously classify points. However for meshes with duplicate faces, the winding number is locally shifted. Similarly, as ambiguities begin to appear, the generalised winding function smoothly shifts to a confidence measure, as shown in figure 3.2. In this way Jacobson et al. claim it correctly handles duplicate, or close to being duplicate, faces, holes and non-manifold attachments. For this reason however, if a simple threshold is used to classify points as inside or outside based on the integral average of the winding number for each element, incorrect classifications may be seen.

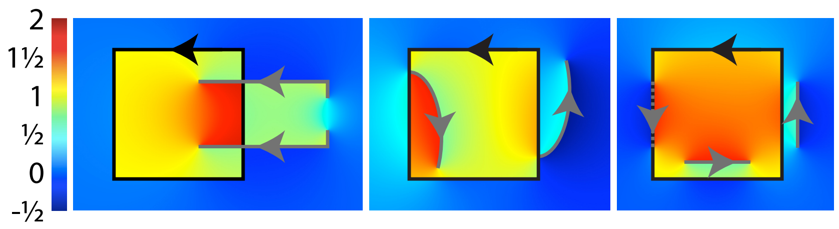


Figure 3.2: *The GWN evaluated over meshes with self intersections, non-manifold attachments and duplicate faces (Jacobson* et al. *2013)*

Instead Jacobson, A. et al. present an energy functional with enforced smoothness, and hence better behaviour, with a minimum respecting the winding number. This is defined as follows:

$$E = \sum_{i=1}^{m} [u(x_i) + \gamma \frac{1}{2} \sum_{j \in N(i)} v(x_i, x_j)] \tag{3.9}$$

where *N(i)* is the set of all elements who share a facet with the element e_i, x_i is the unknown binary segmentation function at e_i and γ is a parameter to control the balance between the data and smoothness terms.

The data term is defined as:

$$u(x_i) = \begin{cases} max(w(e_i) - 0, 0) & \text{if } x_i = \text{outside} \\ max(1 - w(e_i), 0) & \text{otherwise} \end{cases}$$

Whilst the smoothness term is defined as:

$$v(x_i, x_j) = \begin{cases} 0 & \text{if} x_i = x_j \\ \frac{a_{ij} exp(-|w(e_i) - w(e_j)|^2}{2\sigma^2} & \text{otherwise} \end{cases}$$

This method does however assume that a mesh "intuitively represents or loosely approximates the surface of some solid" (Jacobson *et al.* 2013), an assumption Jacobson et al. justify by proposing that most meshes are designed to represent the surface of a solid when lit from one side.

3.3 Mesh Defects

Mesh defects are a very common problem in the computer graphics and fabrication industry. The 3D printing company Shapeways estimates that 90% of meshes they received for printing contain defects. The result of most defects is the lack of a well defined normal which causes many further algorithms to fail, or unexpected behaviour, notably with the AWPN. Such defects can be split into the following categories: flipped faces, degenerate triangles, duplicate faces, self intersections and holes, all of which will now be explained along with methods to potentially overcome them.

3.3.1 Flipped Faces

The most obvious cause of an incorrect normal is when a face on the mesh is flipped, as shown in figure 3.3. That is, the direction of its normal is opposite to the majority of the surrounding faces. More precisely if two triangles have the same start and end vertex then they must have normals which point in opposite directions. In order to resolve this, a correct normal direction must be chosen, after which this direction is essentially propagated through the mesh. To do so every triangle excluding the seed one is marked as having an undefined orientation. The orientation of the faces sharing a vertex with the seed vertex are then checked, and if they don't match that of the seed they are flipped, otherwise their orientation is stored.The same process is then carried out on the newly flipped faces, continuing until all faces have been assigned a normal direction. If the mesh contains disjoint geometry then it is necessary to define multiple seed faces for each disjoint section of geometry. This process is known as unifying normals (Sanchez 2011).

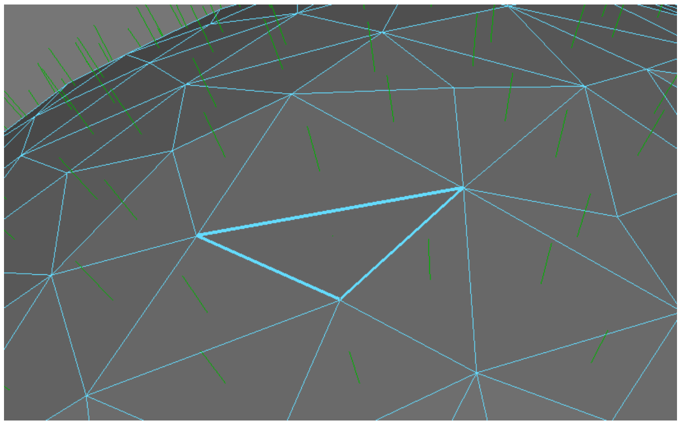


Figure 3.3: *A flipped normal on the triangle with the thick edges*

There is however one type of geometry this method will not work for. These are known as non-orientable surfaces. This is because all faces are simultaneously correctly orientated and in need of flipping. The Klein bottle is possibly the best known of these, as shown in figure 3.4. As the name suggests however such surfaces have no correct orientation and thus are rejected by the algorithm.

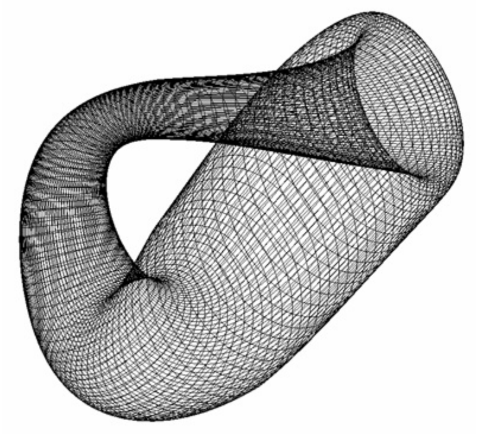


Figure 3.4: *Klein Bottle: A Non-Orientable Surface (Bourke 1996)*

3.3.2 Degenerate Triangles

Similarly, degenerate triangles in a mesh cause anomalies in many algorithms carried out on it due to their lack of well defined normals.

Degenerate triangles are those with a very close to zero area. These tend to fall into two categories, those with an angle close to 180° and those whose longest edge is significantly longer than its shortest edge. Botsch and Kobbelt (2001) name the former type caps and the latter needles. They suggest that for needles we need only collapse the shortest edge to resolve the degenerate triangle. Caps, on the other hand, must be treated with greater care as simply collapsing one of their relatively long edges may cause the neighbouring triangles to become degenerate. Instead they suggest splitting them into further faces with smaller angles. To maintain continuity the neighbouring triangles must also be split after this operation.

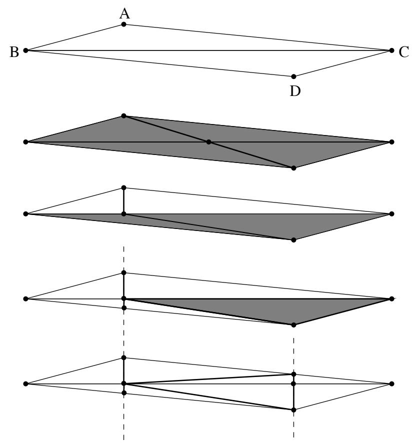


Figure 3.5: *Resolving caps by cutting the mesh at vertices with a large angle, such as A and D. Alternative methods may results in further caps, as highlighted in grey (Botsch and Kobbelt 2001)*

3.3.3 Duplicate Vertices

Duplicate vertices occur when there are multiple vertices at close to, if not identical, positions. There are common on meshes made up from multiple components. For instance there Utah Teapot contains many duplicate vertices. These cause problems with detecting holes and self intersecting triangles, often resulting in false classifications of both, as shown in figure 3.6. To resolve such issues vertex welding can be applied. This process combines all vertices within a set threshold of each other, thereby eliminating the problem.

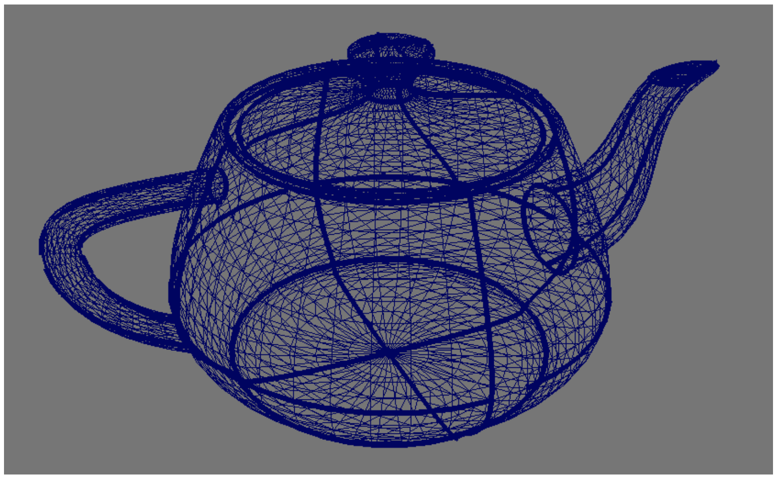


Figure 3.6: *Duplicate vertices cause incorrect classification of boundary edges, marked with thicker lines*

3.3.4 Self Intersections

Self intersections also occur frequently in meshes, often simply as a result of an artist's creativity. For instance, when modelling animals it is common to simply intersect each whisker with the face geometry. Unfortunately however, self intersecting geometry causes anomalies with many algorithms and some to fail completely and thus detecting such triangles is incredibly useful.

There are many algorithms in existence to isolate self intersecting triangles. The most intuitive of these relies on the fact that when two triangles intersect, either one or two edges of the first triangle normally protrude the interior of the other, as shown in figure 3.7. Then by checking each triangle edge in turn for intersection with the other triangle, triangle triangle intersections can be found. On the other hand if all six such tests for each triangle pair are false, there is no intersection between them.The algorithm unfortunately fails if the two triangles being tested are coplanar and due to the large number of checks is also slow to implement.

An alternative method is known as the separating axis test. As the

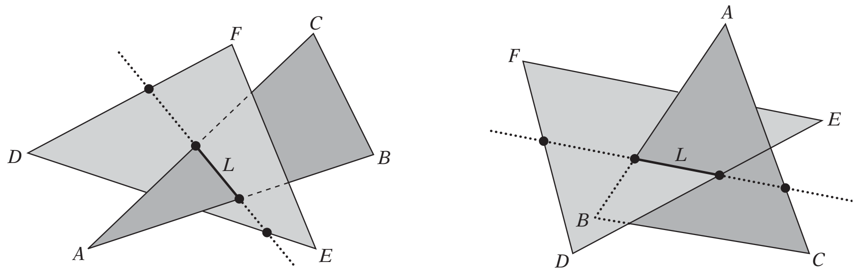


Figure 3.7: *Triangle Triangle Intersection Possibilities (Ericson 2004)*

name suggests, a series of tests are carried out to determine whether one of eleven axes separate the two triangles being tested. These are the nine edge combinations taking one from each triangle, and the two axis parallel to the normal of each triangle. For each axis in turn the vertices are projected onto it and the interval of each triangle's projection is tested against the interval of the other to see if there is an overlap. Two triangles only intersect if they have overlapping intervals on all eleven intervals. This allows the test to end early if any axis has two disjoint projection intervals, thereby cutting down the computation time. Nonetheless this method still requires a large number of checks and is unable to correctly classify coplanar triangles.

An alternative algorithm similar in vein to the separating axis test but less computationally expensive is presented by Möller (1997). The algorithm essentially splits into two key steps. First a check is carried out to see whether either triangle intersects the plane the other lies on. If no intersection occurs the triangles are marked as not intersecting. In this way more complicated checks can be eliminated for many triangles early on. This is done as follows. For triangle (v_0^1, v_1^1, v_2^1) the distances between each vertex and the plane π_2 which the second triangle lies in are found. By checking whether all the distances have the same sign or not, we are able to determine whether or not the two overlap. The same calculations are then done for the second triangle with vertices (v_0^2, v_1^2, v_2^2) and the plane π_1 which the first triangle lies on.

If any two distances for each triangle have a different sign, the two planes containing each triangle must intersect along a line with direction

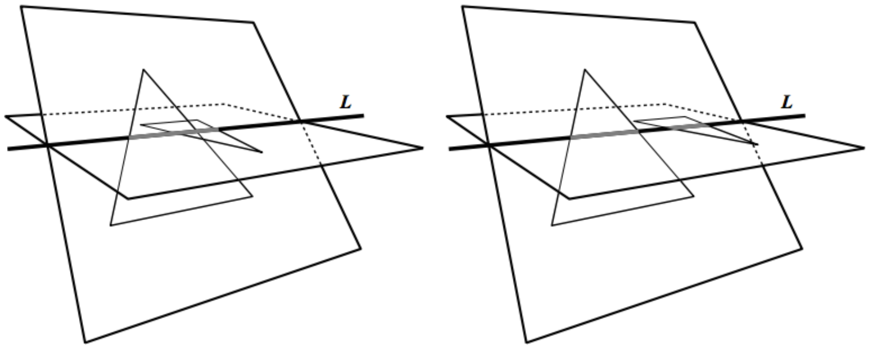


Figure 3.8: *Triangle Line Intersection Possibilities (Möller 1997)*

$N_1 \times N_2$ where N_1 is the normal of the first triangle, and N_2 is the normal of the second. This is because at least one vertex of the first triangle must lie on the opposite side of the plane in which the other triangle lies to the other two triangle vertices. If this overlap is found there are then two possible cases of intersection along this line, as shown in figure 3.8. Namely, either the two planes intersect but the triangles do not, or both the planes and the triangles intersect.

In a similar manner to the separating axis test, the projection interval of each triangle vertex onto this line of intersection is then found to establish which of these two cases we are in. For each vertex its projection onto the line of intersection is given by:

$$P_{v_i^1} = (N_1 \times N_2) \cdot (V_i^1 - O) \tag{3.10}$$

as shown in figure 3.9 where O is a point on the line of intersection. However as translating the interval does not affect the classification result, this can be simplified to:

$$P_{v_i^1} = (N_1 \times N_2) \cdot (V_i^1) \tag{3.11}$$

As with all the methods outlined above though, if the distance to the plane for any one of the six vertices is 0 then the two triangles are coplanar the test will fail. For this reason a second check must be carried out to ensure intersecting coplanar triangles are also marked

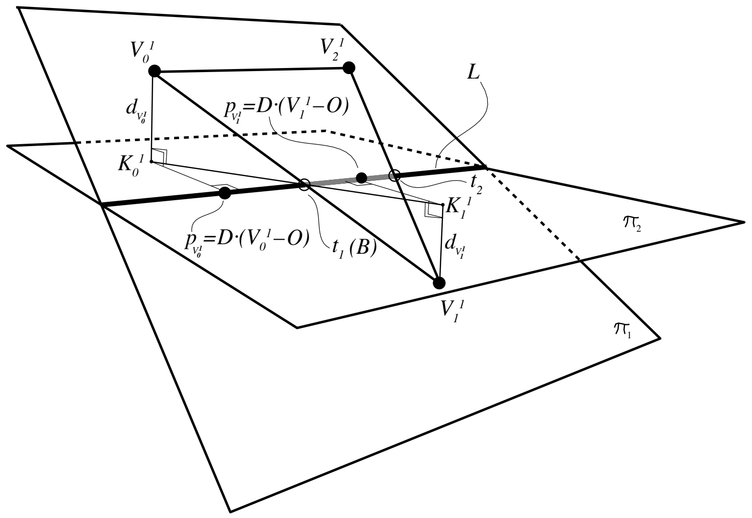


Figure 3.9: *Projecting the vectors of triangle 1 onto the plane of triangle 2 (Möller 1997)*

as self intersecting. This can be done through the use of Barycentric coordinates. For a triangle Barycentric coordinates allow us to express any point in the same plane as the triangle as a linear combination of its vertices. So for a triangle with vertices A, B and C we can express a point p as $p = uA + vB + wC$ where (u, v, w) are the Barycentric coordinates and $u + v + w = 1$. With such a representation we can then very easily test whether a point lies within the triangle by checking whether $0 < u, v, w < 1$ (Ericson 2004).

3.3.5 Holes

Holes in meshes cause significant problems with many algorithms, particularly ray casting and classifying points using the AWPN. There are many hole patching algorithms in existence for triangular meshes. For instance Carr *et al.* (2001) use polyharmonic Radial Basis Functions to build an implicit surface to fill a given hole. This results in smooth hole filling and extrapolation of the surface around the hole. Whilst this method works well for complex holes and convex geometry, for more complex surfaces it becomes difficult to describe them using a single-value function and thus problems arise. Another approach which is able to deal with more complex holes is proposed by Jun (2005). It works by splitting complex holes into more simple sub holes before individually filling each. To do so the sub holes are projected onto a projection plane, before being patched using two dimensional Delaunay triangulation. Unfortunately though, if a hole contains a large number of overlapping sections or twists the process becomes incredibly slow. Perhaps a more intuitive approach is known as ear cutting. This makes use of Meister's Two Ears Theorem which states that for any simple closed polygonal plane curve other than a triangle with a finite number of sides there exists at least two non overlapping ears (Meisters 1975). This allows us to take the two consecutive hole edges with the smallest angle between them and form a triangle between them, known as an ear. By applying this step repeatedly we are able to close a hole through the repeated creation of ears. Unfortunately however this method often leads to multiple self intersections and counterintuitive geometry.

An alternative similar but more robust method is known as the advancing front mesh technique. This works by creating an initial front consisting of all the edges surrounding the hole, known as boundary edges. Starting with the two consecutive edges with the smallest angle between them, either one, two or three triangles are inserted in the plane created by the two edges. The number of triangles inserted is dependant on the size of the angle between the edges. The front is then updated with the new front created by the addition of these triangles before the process starts again. This allows for a robust method to fill holes, whilst remaining fairly intuitive and efficient.

3.4 Polygon Soups

A polygon soup is the name given a series of polygons with no particular relationships, as seen in figure 3.11. In this particular case the polygon soup is created by rotating each triangle on the original cat mesh (figure 3.10) by a random amount in an arbitrary direction (Jacobson *et al.* 2013). Whilst polygon soups do not intuitively represent a solid, they could be meshed using the marching cubes algorithm (Lorensen and Cline

1987) based on the value of the GWN. Such a process would produce a watertight mesh which could then be used with aPIP algorithm.

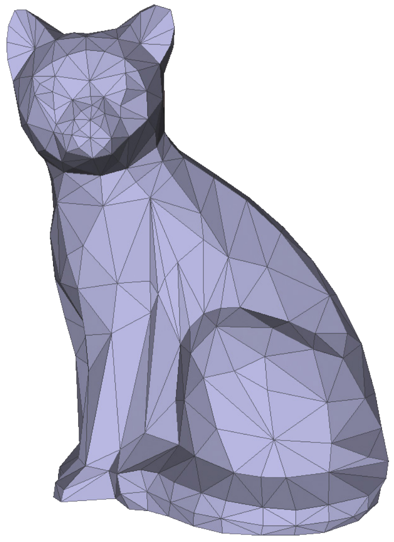


Figure 3.10: *The original cat model (Jacobson* et al. *2013)*

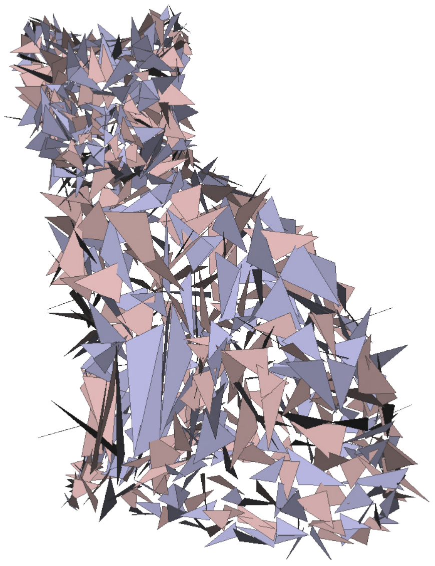


Figure 3.11: *A polygon soup based on the cat model (Jacobson* et al. *2013)*

3.5 Proposed Algorithm Overview

As explained at the start of the chapter, the proposed algorithm comprises a combination of the GWN and the AWPN PIP methods. For this reason it is important to consider the limitations of each method prior to combining them. As previously noted the AWPN algorithm is not able to correctly classify points if the input mesh is not a closed 2-manifold. This could be the case for instance if the input mesh contains holes or self intersections. Thus in both scenarios further attention is needed.

Considering first the case where the input mesh contains self intersecting geometry we note the following. As the AWPN algorithm uses the closest triangle on the mesh to classify each point, only the points for which this triangle is either inside the geometry or intersected by another triangle will be incorrectly classified. For this reason we propose excluding the triangles inside the mesh from the AWPN algorithm. To do so a new algorithm utilising the GWN is presented below to establish whether a triangle lies fully inside the geometry. In this way we are able to prevent such triangles from incorrect classifying points using the AWPN algorithm.

For the second case in which the triangles themselves are intersected, it is clear these cannot be excluded from the algorithm as they form the edge of the mesh. Instead once such triangles have been identified using the triangle triangle intersection algorithm proposed by Möller (1997), we use the GWN algorithm to classify all points for which their closest triangle is one of these self intersected ones. This prevents the AWPN algorithm being used for such points, as it may incorrectly classify them.

On the other hand, areas which contain holes must be treated with a different approach. Whilst the GWN method can be used in such areas, the affect of such a hole on the GWN field can be far reaching, as shown in Figure 3.12. This proposes the question of how to identify the areas affected by any given hole. One approach considered was to construct an octree for the input mesh, using the bounding box of the mesh as the root. The GWN was then evaluated using interval arithmetic for each cell, and cells containing an interval crossing +0.5 were split. In this way the areas affected by the holes could be identified and evaluated using the GWN and the rest classified using the AWPN method. Unfortunately though, the large overestimations often seen with interval arithmetic resulted in this being unusable. To see such overerstimations consider the equation

$$x^2 + \frac{x}{4} \tag{3.12}$$

for x in the range $[-1, 1]\backslash\{0\}$. Using interval arithmetics the output range would be [0,1] for the x^2 term and $[\frac{-1}{4}, \frac{1}{4}]$ for the $\frac{x}{4}$ term which sums to $[\frac{-1}{4}, \frac{5}{4}]$. However we need only consider the equation briefly to see that as the first term must always be positive and as zero is excluded from our range it is impossible to ever achieve a value of $\frac{-1}{4}$ and thus interval arithmetics has overestimated the possible output values.

Instead the approach used is to calculate a patch for any holes on the

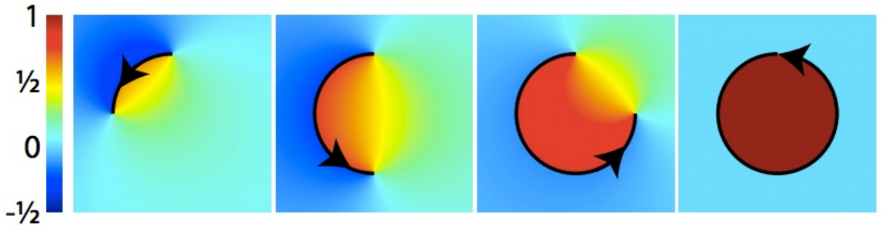


Figure 3.12: *The GWN Field for Open Curves (Jacobson* et al. *2013)*

mesh. This is done as a two step process. First the edges surrounding each hole are identified, before a patch is calculated for each using the Advancing Front Mesh (AFM) algorithm, as presented by Zhao *et al.* (2007). Provided all self intersecting geometry has already been considered in the manner outlined above, the AWPN algorithm can then be utilised with these additional triangles which form a valid closing of the mesh. This allows the AWPN to correctly classify points closest to a hole.

In this way we can use a combination of both algorithms at different parts of the mesh to classify points without the time implications of using the GWN everywhere but still gaining robustness through its targeted use. As the algorithm uses either the AWPN or GWN for the classification of each point, it has a best case complexity of $\mathcal{O}(\log(\mathrm{N}))$ where N is the number of triangles and an acceleration structure is used to calculate the AWPN and a worst case complexity of $\mathcal{O}(\mathrm{N})$.

Chapter 4

Engineering & Implementation

We will now give implementation details for each step of our proposed PIP algorithm which has been implemented in C++ as a Maya plugin. The plugin inherits from the MPxNode which is Maya's base node for custom dependency nodes. It allows the user to input the mesh they wish to classify points on and easily create and orientate a plane for which they would like to visualise the results. The plugin then creates a new mesh to show the patches used to fill holes, and which triangles have been removed. In order to determine the world coordinates for each point a SamplerInfo node is used, and the classification value of each is then passed to a blender node to visualise the result. For ease of comparison the plugin also has the option to use just the AWPN or GWN methods.

4.1 Hole Patching

The first step in the algorithm is to patch any holes on the input mesh. This is done as a two part process. First the edges surrounding each hole are found before a patch is calculated to close the hole using the advancing front method outlined by Zhao *et al.* (2007). It is worth noting however that this patch is simply for classification purposes, the input mesh is not altered in any way. Instead a copy is made which is patched and used for calculations.

4.1.1 Hole Identification

In order to calculate patches to close holes in the mesh it is necessary to identify all the edges which enclose each hole, known as border edges. Furthermore for use in later algorithms these need to be stored in order. To do so all the edges in the mesh are iterated through until the first border edge is found, or until all edges have been considered and found not to be border edges, in which case there are no holes within the mesh. Assuming a border edge is found, all edges connected to this border edge are tested until another border edge is found. This process is repeated until the next border edge found is the same as the first border edge established. In this way all the edges around the hole are found in order. The iteration through all the edges then continues until either another border edge not already accounted for is reached and the process starts again, or until all edges have been checked as shown in algorithm 1.

When implementing this in Maya it is necessary to get the edge information from Maya's own iterator using the MitMeshEdge class. This returns both vertices of each edge, however the orientations of these vertices are inconsistent. This necessitates an extra step whereby the vertex positions of the ends of two adjoining edges are compared to establish which vertex is shared between them. In this way we are able to store all the hole vertices in the correct order for later use as we traverse around the hole.

4.1.2 Advancing Front Mesh Patch Creation

The advancing front mesh (AFM) technique explained in 3.3.5 was chosen to patch holes in the input mesh. To recap, the idea is as follows. Given the boundary edges of a hole, the angle between every two adjacent edges is calculated. For the two edges with the smallest angle either

```
Input:  $Edges$ : the edges of the input mesh
Output:  $holeEdges$ : the edges around each hole in order
$holeInfo$: the number of edge in each hole, and the index into
$holeEdges$ of the first edge for each
foreach Edge i do
    if i is a boundary edge then
        $boundaryEdges \leftarrow i$
    end
end
while $holeEdges.size() \neq boundaryEdges.size()$ do
    foreach Boundary Edge j do
        $currentEdge \leftarrow j$ ;
        $holeEdges[0] \leftarrow j$ ;
        while $currentEdge \neq holeEdges[0]$ do
            get edges connected to $currentEdge$ ;
            foreach connectedEdge k do
                if $k \in boundaryEdge$ and $k \notin holeEdges$ then
                    $holeEdges \leftarrow k$ ;
                    $currentEdge = k$ ;
                    $count+ = 1$ ; break ;
                end
            end
        end
        $holeEdges \leftarrow count$ ;
        $holeEdges \leftarrow$ holeEdges.size();
    end
end
return $holeEdges, holeInfo$ ;
```

Algorithm 1: Finding all holes in the mesh

one, two or three triangles are then created in the plane formed by the two edges to fill the gap. The decision as to the number of triangles used to close the hole is dependant on the angle between the two edges, theta. For $\theta \leq 75°$ one additional triangle is created, if $75 < \theta \leq 135°$ the two are added and for $\theta > 135°$ three are. This can be seen in Figure 4.1 where v_{new} denotes the new vertices created to form each additional triangle. To prevent the additional triangles becoming significantly smaller than the original triangles on the mesh, a check is made as to whether any new vertex is within a set threshold of an existing one. If so, the original vertex is used rather than creating a new one. In essence this means that ear clipping is carried out in such scenarios, as shown in algorithm 2. Such a threshold is calculated as a fraction of the average length of the edges in the original advancing front.

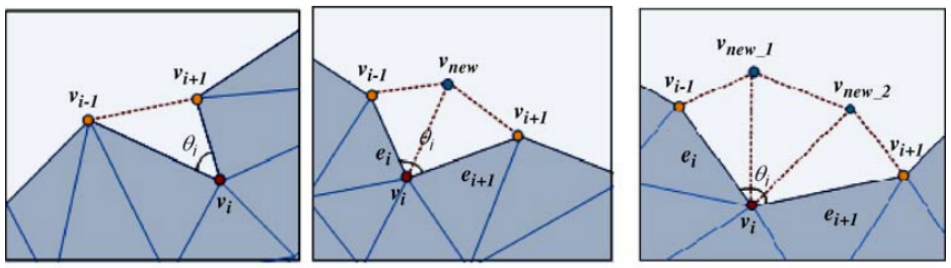


Figure 4.1: *Triangle Creation Rules*
left to right: $\theta \leq 75°$, $75 < \theta \leq 135°$, $\theta > 135°$

When implementing this algorithm there were a few extra considerations which will be explained now. Firstly the paper doesn't explicitly say how to determine the location of the additional vertices beyond that the new triangle must be on the same plane as the two adjacent boundary edges. For this reason a decision was made to calculate the location of each new vertex by ensuring that the length of the new edge was a weighted average of either boundary edge, and the direction a weighted average of each edge vector. For the case where $75 < \theta \leq 135°$ the weighting of the additional edge is simply an average of the two existing boundary edges. However when $\theta > 135°$ and thus two new edges must be added each is weighted $\frac{1}{3}$ and $\frac{2}{3}$ with the greater fraction corresponding to the closest border edge.

Secondly extra consideration has to be taken into the order in which the indices of each new triangle are stored. This is because this order determines the direction of the triangle's normal. Thus special care must be given to ensure the normal direction of any additional triangles matches those already on the mesh. This order can be seen in figure 4.2. This is done by establishing the face id of the triangle connected to the first boundary edge found. The MitMeshPolygon class is then used which provides an iterator to each face on the mesh, with which we can establish the faces vertex indices. As these must be orientated correctly for the mesh, we can use this order to store the direction of the boundary edge and thus the new triangle indices.

```
Input: $holeBoundaryVerts$ : the vertices around the hole in order
Output: $triangleIndices$ : the indices of the patch triangles
$triangleVertices$ : the vertex positions of the patch triangles
while $holeBoundaryVerts.size() > 2$ do
    $\theta \leftarrow$ the smallest angle between two consecutive hole edges;
    $i \leftarrow$ the index of the first edge corresponding to $\theta$;
    if $\theta \leq 75°$ then
        add a single new triangle with vertices $(V_{i-1}, V_{i+1}, V_i)$
    end
    if $75 < \theta \leq 135°$ then
        add two new triangles with vertices $(V_{i-1}, V_{new}, V_i)$ and
        $(V_{new}, V_{i+1}$ and $V_i)$
    end
    if $\theta > 135°$ then
        add three new triangles with vertices $(V_{i-1}, V_{new_1}, V_i)$ and
        $(V_{new_1}, V_{new_2}$ and $V_i)$ and $(V_{new_2}, V_{i+1}$ and $V_i)$
    end
    remove vertex $V_i$ from $holeBoundaryVerts$ ;
    foreach new vertex do
        foreach related vertex do
            $d \leftarrow$ the distance to each related vertex ;
            if $d < threshold$ then
                merge the new vertex with the existing one ;
            end
        end
    end
    $triangleIndices \leftarrow$ the new triangle indices ;
    $triangleVertices$ : the new triangle vertices ;
end
return $triangleIndices, triangleVerticds$ ;
```

Algorithm 2: AFM patch calculation to close a given hole

Finally, difficulties arose in calculating the smallest internal angle between two consecutive border edges at the start of the algorithm. This is because formulas such as the well known $\theta = acos(\mathbf{a}, \mathbf{b})$, where θ is the angle between vectors **a** and **b** return the smallest angle between the two. However there may well be cases where the angle we wish to calculate is the larger of the two. Unfortunately the same problem occurred using the more robust formula

$$\theta = atan\left(\frac{||\mathbf{a} \times \mathbf{b}||}{\mathbf{a} \cdot \mathbf{b}}\right) \tag{4.1}$$

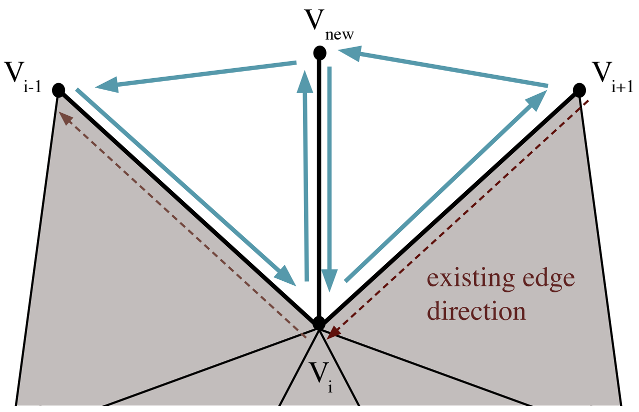


Figure 4.2: *Correct Edge Directions for additional triangles*

and hence a further check was implemented. After finding the angle between two adjacent edges using equation 4.1 this is checked by taking the sum of the angles in all the existing triangles with a vertex at that position. In two dimensions this can be seen in figure 4.3. By taking the sum $\phi_1 + \phi_2 + \phi_3$ we can compare this angle with ϕ. If the two are equal, then we know the angle required is the greater of the two so we set θ to equal $2\pi - \phi$.

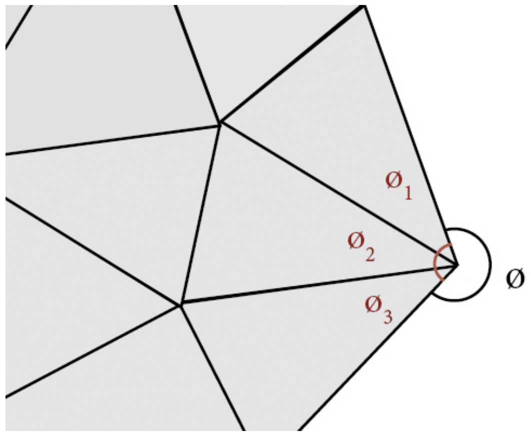


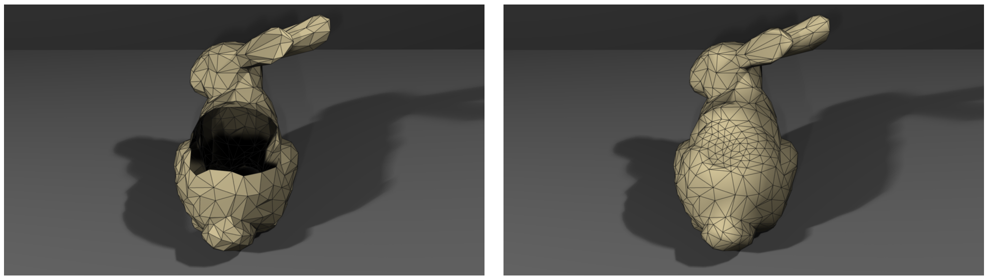
Figure 4.3: *Checking where the angle required is the reflex one by taking the sum of $\phi_1, \phi_2$and ϕ_3 and comparing with the angle already calculated*

There is one slight complication however when translating this to three dimensions. As it is not necessary for the triangles on the mesh to lie on the plane created by the two border edges, and indeed they normally do not, we cannot simply measure the angle between each edge. Instead each edge must be projected into the plane created by the border edges, before the angles between each can be measured. This is done using the following equation:

$$\mathbf{e_{proj}} = \mathbf{n} \times (\mathbf{e} \times \mathbf{n}) \tag{4.2}$$

where $\mathbf{e}$ is the edge to be projected, $\mathbf{n}$ is the normal to the plane created by the two border edges either side of it and $\mathbf{e_{proj}}$ is the projection of $\mathbf{e}$ into this plane.

Once projected however we can measure and compare the angles as previously described and take a comparison between that and the previously found angle to check if we have measured the correct side. The result of the AFM hole patching with these extra steps can be seen in figure 4.4.



Stanford bunny containing a hole *Patched using the AFM technique*

Figure 4.4: *Patching a Stanford bunny using the AFM technique*

It is worth noting there are few requirements for this algorithm which have implications for the applicability of our method. Firstly, the input geometry must be manifold, as well as connected and orientated and secondly it must not contain any islands. These are areas of geometry which are part of the mesh, but entirely separated by a surrounding hole, thus forming an island.

4.2 Internal Triangle Exclusion

As noted in 3.1 the AWPN classifies points using the closest triangle on the mesh. However, it not able to correctly classify points close to self intersecting triangles. Thus, by removing any triangles fully inside a mesh we are able to increase the number of points for which the AWPN can be used, thereby reducing the computation time for our algorithm.

To do so an original algorithm is presented which utilises the GWN. As explained previously the GWN represents the signed number of times a surface wraps around a point (Jacobson *et al.* 2013). To evaluate the winding number for a point on the mesh, we take the sum of the signed projections of each triangle onto the unit sphere centred at that point, before dividing by 4π, as explained further in section 4.4. In this way each triangle on the mesh affects the overall winding number. It is this fact we use to establish whether triangles lie fully inside the mesh as follows.

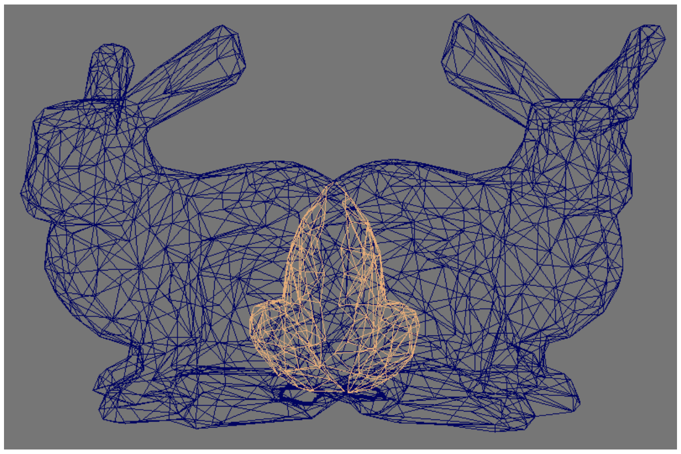


Figure 4.5: *Selecting the triangles fully inside the geometry*

Each triangle in turn is removed from the mesh, and the GWN evaluated at the centroid of the removed triangle. The value of the GWN evaluation at that point is then used to establish whether the triangle being tested lies on the edge of the mesh or not. This is because the projection of the triangle onto the unit sphere will contribute either +0.5 or -0.5 to the overall GWN. As we know for watertight meshes the GWN evaluates to exactly 1 inside the mesh and 0 outside, any point evaluated in the way described above that results in a GWN of less than or equal to 0.75 must lie on the exterior of the mesh as it's inclusion would either take the GWN below or above to 0 or 1, thereby being outside or inside. This can be seen in Figure 4.6 and is summarised in Algorithm 3. As the GWN only evaluates exactly for watertight meshes this check must be carried out after any holes have had patches calculated for them. It is also necessary to leave those triangles which themselves are self intersected as these form the boundary of the shape and are therefore required for the AWPN algorithm to be able to evaluate points correctly. For this reason such triangles are dealt with later. The result of this algorithm can be seen in figure 4.5.

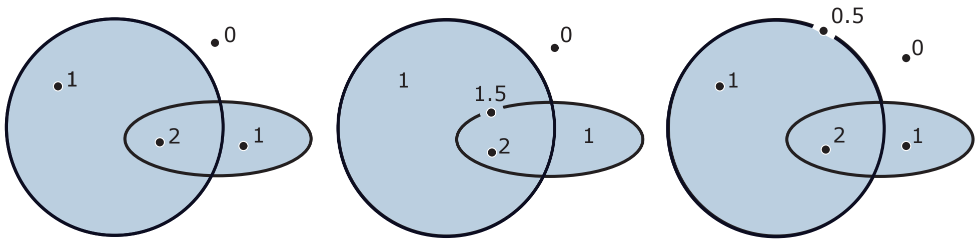


Figure 4.6: *Evaluating the GWN at the Centre of Removed Triangles*

```
Input: indices : the vertex indices of the input mesh
vertices : the vertices of the input mesh
Output: externalTriangles : the triangle indies of all triangles not
        fully inside the mesh
foreach Triangle i on the patched mesh with vertices $(x_1, y_1, z_1)$,
$(x_2, y_2, z_2)$, $(z_1, z_2, z_3)$ do
    $Centroid \leftarrow (\frac{x_1+x_2+x_3}{3}, \frac{y_1+y_2+y_3}{3}, \frac{z_1+z_2+z_3}{3})$ ;
    $GWN \leftarrow$ GWN evaluated at the centroid of triangle $i$ over all
    triangles excluding $i$ itself ;
    if $GWN < 0.75$ then
        $externalTriangles \leftarrow i$
    end
end
return externalTriangles ;
```

Algorithm 3: Removing Fully Inside Self Intersecting Triangles

4.3 Self Intersecting Triangle Identification

The final problematic triangles to identify are those which are themselves intersected by another triangle. As previously noted these cannot be removed from the mesh as they form the boundary of the object. Instead they are marked as self intersecting so that the GWN technique can be used to classify all points closest to a self intersecting triangle. To do so we have implemented the triangle triangle intersection algorithm presented by Möller (1997).

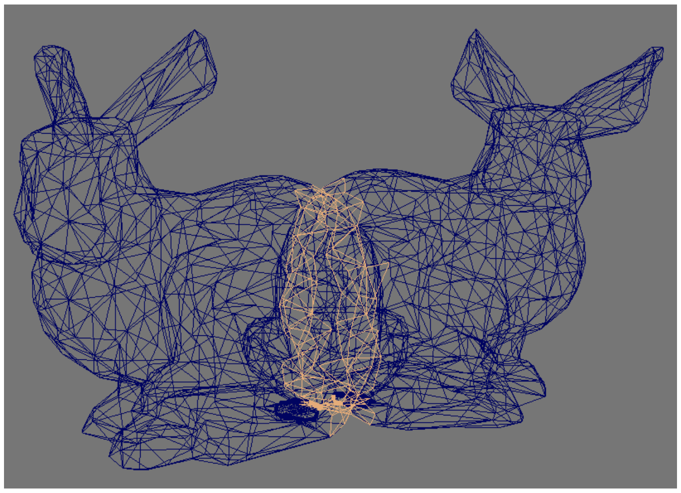


Figure 4.7: *Selecting the self intersecting triangles on a mesh*

As explained in 3.3.4 this algorithm works by first checking if the planes containing any two triangles intersect anywhere. If so, then eleven axis checks are carried out by projecting the vertices of each triangle onto the axis and comparing the intervals for each. This is summarised in algorithm 4 and can be seen applied to two intersecting Stanford bunnies in figure 4.7.

4.4 Generalised Winding Number

In order to evaluate the GWN for a given point we use the following two observations. If C is piecewise linear, then we immediately have an analogous discrete equation for the two dimensional winding number given in equation 3.6 as follows:

$$w(\mathbf{p}) = \frac{1}{2\pi} \sum_{i=1}^{n} \theta_i \tag{4.3}$$

where θ_i is the angle between the vectors from $\mathbf{p}$ to two consecutive

```
Input: indices : the indices of the input mesh
vertices : the vertices of the input mesh
Output: intersectingTri: the indices of all intersected triangles
foreach triangle i with vertices ($v_1, v_2, v_3$) do
    foreach other triangle j on the mesh with vertices ($u_1, u_2, u_3$) do
        distToJ ← calculate the distances from $v_1, v_2$ and $v_3$ to the
        plane triangle j lies in ;
        distToI ← calculate the distances fromfrom $u_1, u_2$ and$u_3$ to the
        plane triangle i lies in ;
        if all distToJ and distToI have the same sign, and none are
        equal to 0 then
            break;
        end
        else
            L ← the line of intersection between the planes each triangle
            lies in ;
            intV ← the interval of the projection of $v_1, v_2$ and $v_3$ onto $L$ ;
            intU ← the interval of the projection of $u_1, u_2$ and $u_3$ onto $L$
            ;
            if intV and intU overlap then
                intersectingTri ← i
            end
            else
                break;
            end
        end
    end
end
return intersectingTri ;
```

Algorithm 4: Finding triangle triangle intersections

vertices c_i and c_{i+1} on C, see figure 4.8.

Similarly the three diminutional generalised winding number given in equation 3.8 can be directly discretised for piecewise-linear triangulated surfaces as follows:

$$\omega(\mathbf{p}) = \sum_{f=1}^{m} \frac{1}{4\pi} \Omega_f(\mathbf{p}) \tag{4.4}$$

where Ω_f is the solid angle of the oriented triangle $(\boldsymbol{v_i}, \boldsymbol{v_j}, \boldsymbol{v_k})$ with respect to $\mathbf{p}$ as previously explained and shown in figure 3.1. Thus,

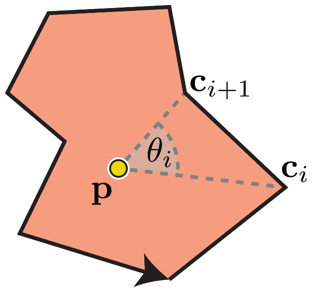


Figure 4.8: *Winding Number Discretization in Two Dimensions (Jacobson* et al. *2013)*

utilising the work of Van Oosterom and Strackee (1983), we have the following discrete formula for the solid angle, Ω_f

$$\Omega(\mathbf{p}) = 2arctan(\frac{det([\mathbf{abc}])}{abc + (\mathbf{a}\cdot\mathbf{b})c + (\mathbf{b}\cdot\mathbf{c})a + (\mathbf{c}\cdot\mathbf{a})b}) \tag{4.5}$$

where $\mathbf{a} = \boldsymbol{v_i}$ - $\mathbf{p}$, $\mathbf{b} = \boldsymbol{v_j}$ - $\mathbf{p}$, $\mathbf{c} = \boldsymbol{v_k}$ - $\mathbf{p}$ and $a = ||\ \mathbf{a}||$, $b = ||\ \mathbf{b}||$ and $c = ||\ \mathbf{c}||$.

With equations 4.5 and 4.4 we are able to calculate the winding number for any given point as highlighted in algorithm 5.

```
Input: $\mathbf{p}$: the point to evaluate the GWN for
$indices$ : the indices of the input mesh
$vertices$ : the veracities of the input mesh
Output: $wn$: the winding number for point $\mathbf{p}$
$wn \leftarrow 0$
foreach Triangle in triangleVerts with corners $\boldsymbol{v_i}, \boldsymbol{v_j}, \boldsymbol{v_k}$ do
    $A \leftarrow \boldsymbol{v_i}$ - $\mathbf{p}$ ;
    $B \leftarrow \boldsymbol{v_j}$ - $\mathbf{p}$ ;
    $C \leftarrow \boldsymbol{v_k}$ - $\mathbf{p}$ ;
    $a \leftarrow$ length of $A$ ;
    $b \leftarrow$ length of $B$ ;
    $c \leftarrow$ length of $C$ ;
    $det \leftarrow$ determinant of matrix $[ABC\ ]$ ;
    $val \leftarrow 2 * arctan(\frac{det}{abc+(A\cdot B)c+(B\cdot C)a+(C\cdot A)b})$
    $wn+ = val$;
end
return $wn$ ;
```
Algorithm 5: GWN Evaluation in Three Dimensions

4.5 Point Classification

With the preprocessing in place, the actual PIP test is fairly simple. As the copy of the input mesh now contains patches to close all holes, and has all the fully internal triangles removed we are able to use the AWPN to evaluate most points. However self intersecting triangles will still cause misclassifications with this method, and thus any point closest to a triangle marked as self intersecting is evaluated using the GWN technique. In this way we are able to evaluate unambiguous cases using the AWPN and the time efficiencies that come with the method, whilst reserving the GWN for areas of potential ambiguity. This is shown in Algorithm 6.

```
Input: p: the coordinates of the point being classified
i : the index of the closest triangle
Output: result : The inside (1) or outside (0) classification of point p
if i is self intersected then
    WN ← the GWN evaluated at p ;
    if $WN > 0.5$ then
        result = 1 ;
    end
    else
        result = 0 ;
    end
end
else
    PN ← AWPN evaluated at p on a mesh (excluding all self
    intersecting triangles entirely inside the input mesh and include all
    hole patches previously calculated)
    if $PN > 0$ then
        result = 1 ;
    end
    else
        result = 0 ;
    end
end
return result ;
```

Algorithm 6: Point Classification Algorithm

Chapter 5

Results

The algorithm has been tested on a series of different meshes as shown in figures 5.1 to 5.5. To visualise the classifications a plane has been placed through each mesh and red assigned to those points classified as inside and blue to those outside. Additionally a breakdown of our method is shown detailing where each technique is used. For such images red denotes the GWN and blue the AWPN. As the techniques for resolving flipped faces, degenerate triangles, degenerate vertices and triangulating meshes are well know and outside the scope of this project all meshes have had such operations applied to them prior to being classified. Results for classifications using the AWPN and GWN methods are also shown to help evaluate the performance of our algorithm.

The computation times for each mesh and technique can be seen in table 5. For each mesh the computation time for our method is at least 2.5 times faster than that of the GWN. As expected this computation time increases relative to the AWPN computation time as the number of self intersecting triangles on the mesh increase, due to the increased use of the GWN. The precomputation times for our method were measured on a Linux Workstation with twelve Intel Xeon E5-1650 3.20GHz CPU with 32GB of memory by calculating the difference in seconds between the time at the start and end of the function call. Whilst not negligible, these are at most 25 percent of the computation time required for the GWN. Including the pre calculation time our algorithm it is still at least

2.4 times faster than the GWN and up to 31 times faster, as in the case of the Dragon.

					Computation Times			
Mesh Name	NF	NV	NH	NSIT	PN	GWN	OM	POM
Telephone	83998	42003	0	1780	0:0:10	3:09:14	0:1:07	0:46:16
Dragon	12500	6250	0	8	0:0:11	0:34:54	0:0:20	0:1:1
Apollo Soyuz	4642	4829	977	2585	0:0:07	0:22:14	0:8:48	0:0:27
Double Bunny	2130	1088	2	110	0:0:05	0:5:39	0:0:12	0:0:2
Utah Teapot	6376	3231	4	309	0:0:09	0:16:37	0:2:35	0:0:19

Table 5.1: *Computation time comparisons*

NF is the number of faces, NV the number of vertices, NH the number of holes and NSIT the number of self intersecting triangles found by the triangle triangle intersection algorithm presented by Möller (1997). OM denotes our method and POM the pre calculation times for our method in hours, minutes and seconds

As seen in figures 5.1 to 5.5, unlike the AWPN method, our technique is able to correctly classify points on meshes containing holes and self intersections. This is done in at most 40 percent of the computation time required for the GWN algorithm, but has a considerably greater accuracy than the AWPN technique. Additionally, as manifold meshes without any holes or self intersections are always classifiably by both the AWPN and GWN methods, it is evident that our method is able to unambiguously classify such points too. Whilst there is a small increase in calculation time compared with simply using the AWPN method in such cases, it is still considerably quicker than the GWN. This means in scenarios where it is not known at the outset whether the mesh for which the classifications are desired contains defects, our method will obtain the correct results within a time scale similar to that of the AWPN method without the user having to specify whether there are self intersections or holes within the mesh. This is particularly useful as such defects are both difficult to detect and often go unnoticed during the modelling process. The Apollo Soyuz model provides a good example of the type of meshes often found in industry containing many holes and self intersections, as well as large variations in triangle sizes.

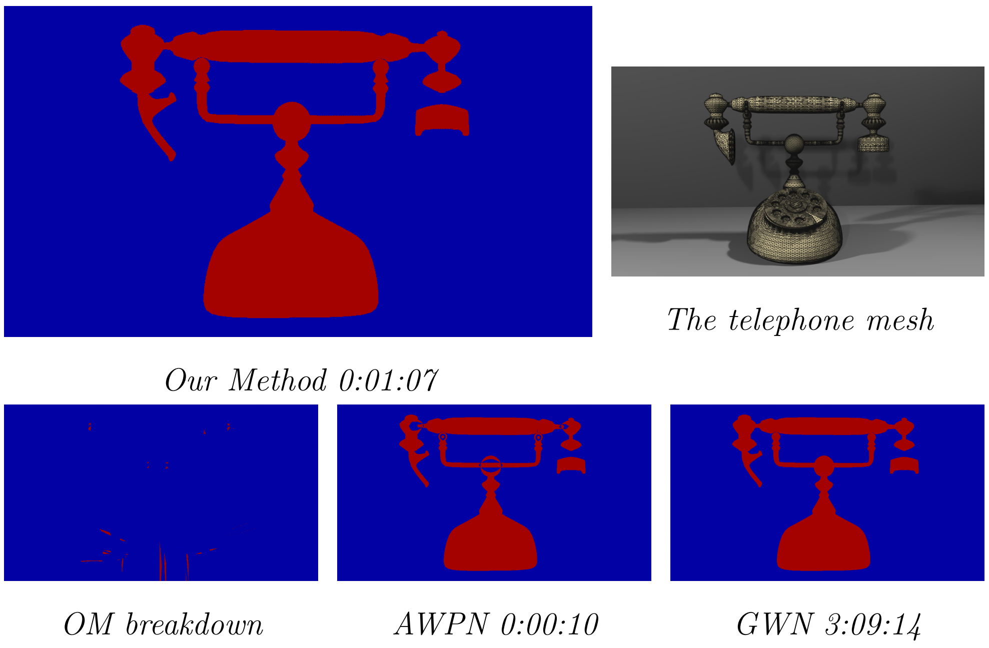
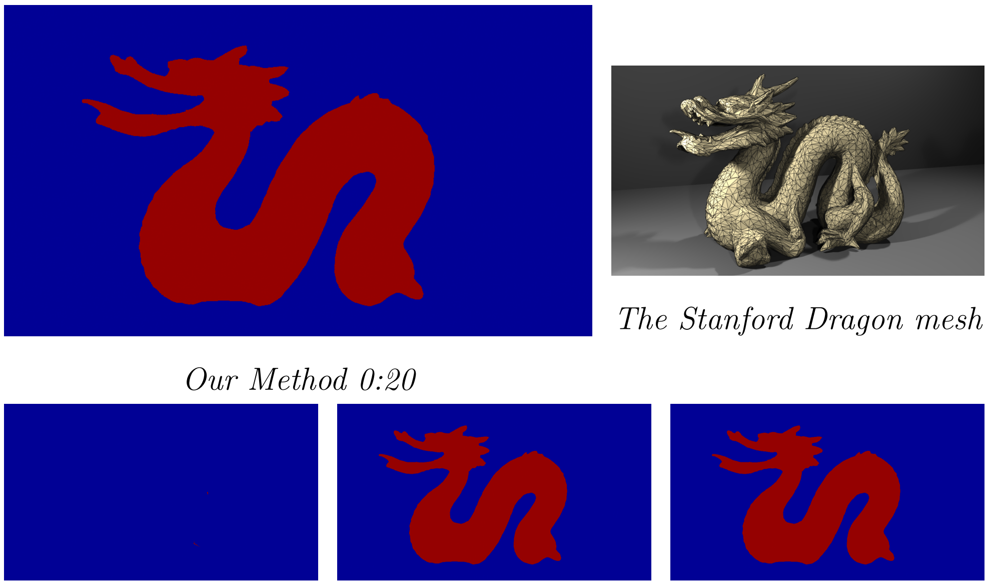


Figure 5.1: *Classification of a plane through a telephone model containing 83998 faces and 1780 self intersecting triangles. Model available from Jacobson* et al. *(2013) supplementary material*

Careful observation of the classifications of points on the this mesh and the Utah Teapot in figures 5.3 and 5.5 show small areas where our algorithm incorrectly classifies points, for instance around the handle on the teapot. This is due to the triangle triangle intersection algorithm used not marking all intersecting triangles as such. In the case of the Utah Teapot, this can be seen in figure 5.6 where none of the triangles on the upper side of the handle are marked as intersecting, despite being partially inside the teapot. Similarly those triangles on the surface of the pot itself which are interested by the handle are not highlighted. As a consequence our algorithm will classify all points for which one of the unmarked self intersecting triangles is the closest to the point being classified using the AWPN technique. As this method is not able to deal with self intersecting geometry this causes the incorrect classification of points in this area. This means our method is as reliable as the triangle triangle intersection algorithm.

Robust triangle triangle intersection detection however is still a current open area of research (Sabharwal and Leopold (2013), Elsheikh



The Stanford Dragon mesh

Our Method 0:20

OM breakdown *AWPN 0:11* *GWN 34:54*

Figure 5.2: *Classification of a plane through the Stanford Dragon containing 12500 faces and 8 self intersecting triangles. Model available from the Stanford University Computer Graphics Laboratory (2013)*

and Elsheikh (2014) and Wei (2013)) and thus outside the scope of this project. However as work in this area improves, our technique will see direct improvements too. This is highlighted in figure 5.7 where all triangle triangle intersections have been found by placing oriented bounding boxes around each triangle and checking for intersections between each using the separating axis theorem. However this method is still not perfect as any two self intersecting triangles which share a vertex will still not be marked as intersecting, hence the thin line of misclassified points by the handle on the left hand side of the teapot. Nonetheless we can already see improvements to the method and this confirms that once the robust triangle triangle intersection problem is solved, our method will be robust.

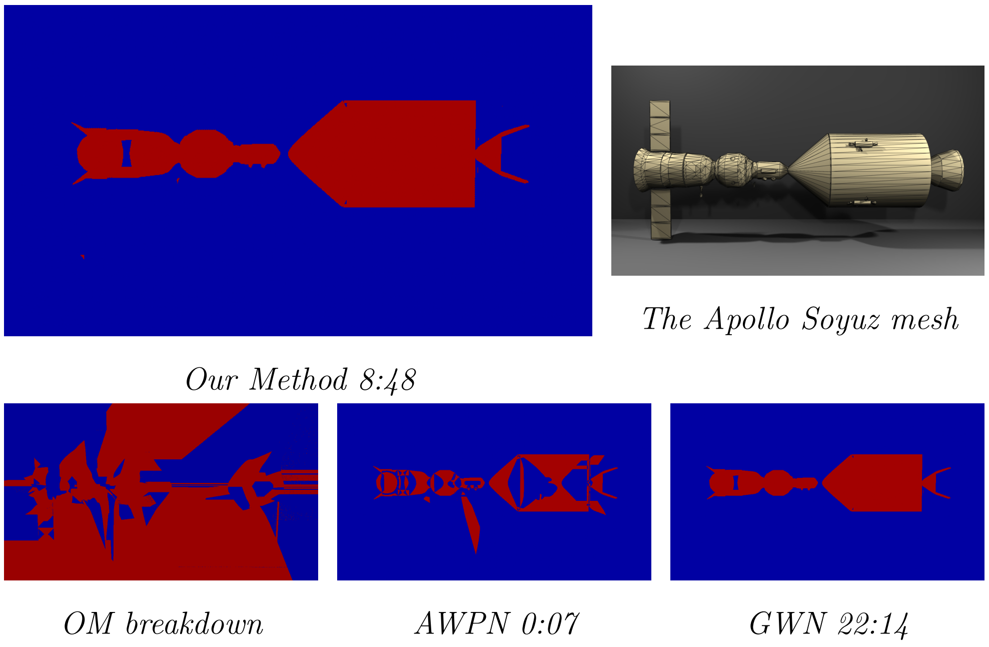


Figure 5.3: *Classification of a plane through an Apollo Soyuz Model containing 4642 faces, 2585 self intersecting triangles and 977 holes. Model available from NASA*

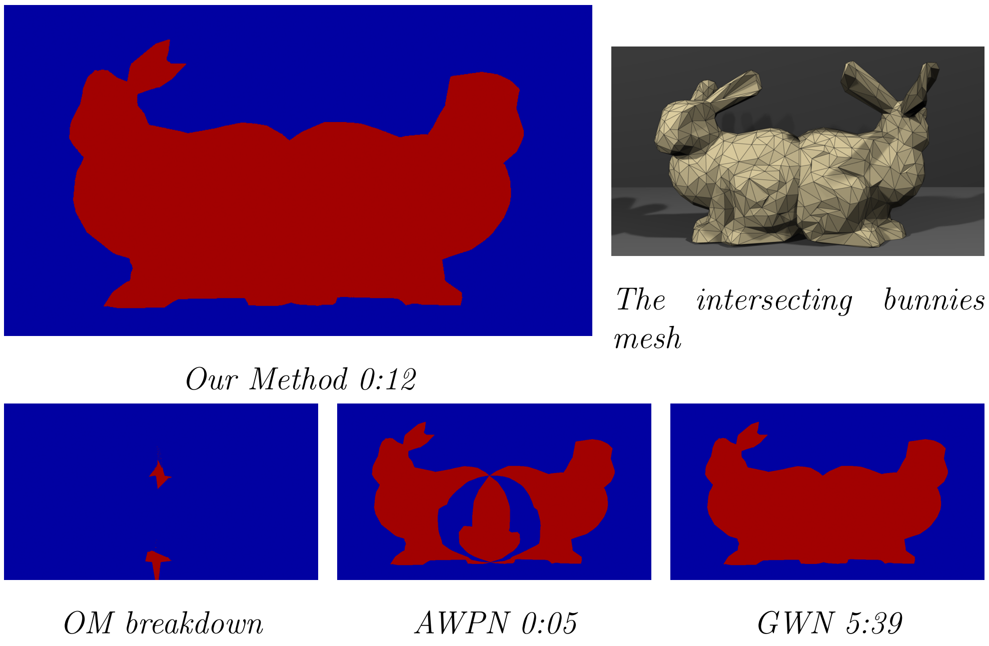
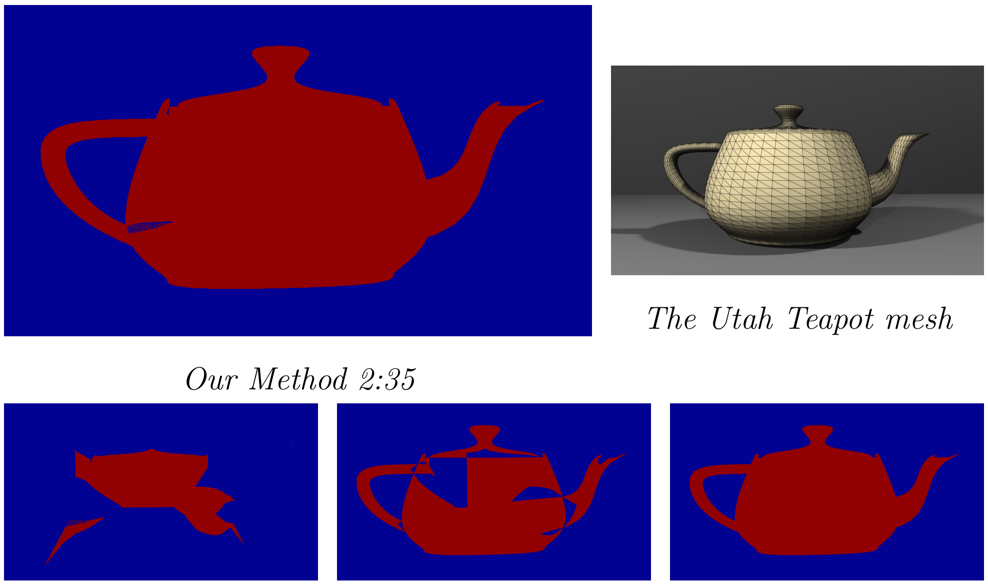


Figure 5.4: *Intersection of two Stanford bunnies containing 2130 faces, 2 holes in the base and 110 self intersecting triangles. Original model available from the Stanford University Computer Graphics Laboratory (2013)*



The Utah Teapot mesh

Our Method 2:35

OM breakdown *AWPN 0:09* *GWN 16:37*

Figure 5.5: *Classification of a plane through the Utah teapot containing 6376 faces, 4 holes and 309 self intersecting triangles. Model available from Martin Newell*

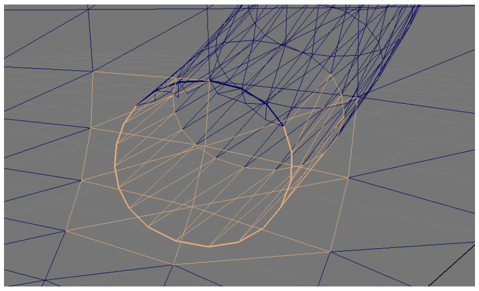


Figure 5.6: *Triangle Triangle Intersection Detection Error using the algorithm by Möller (1997)*

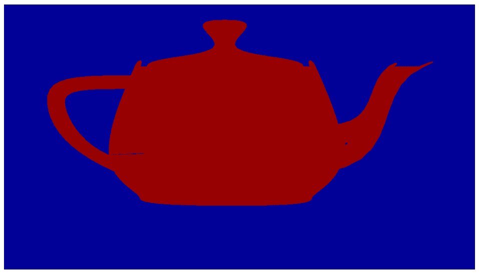


Figure 5.7: *Our method of classification using an alternative triangle triangle algorithm*

Chapter 6

Conclusion

6.1 Summary

In this thesis we have presented a new point membership algorithm for triangulated meshes which combines two existing techniques, the AWPN and the GWN. It has been highlighted that the computation time for classifications with this method is faster than the GWN method and certainly more robust than the AWPN technique. This is because unlike the AWPN technique it is able to deal with meshes containing both holes and self intersections. This is of particular significance as the vast majority of meshes used in industry contain such defects. For instance CAD models are often made up of multiple connected components which makes them particularly susceptible to holes and self intersections.

The implementation of this technique as a Maya plugin allows for both an intuitive visualisation of classification results, and the ability for users to classify different models with ease.

6.2 Future work

There are two key areas for improvement with the proposed method. Firstly, as previously mentioned there are cases in which the Möller

(1997) triangle triangle intersection algorithm fails to mark all intersecting triangles, and thus misclassifications occur at any point for which a missed self intersecting triangle is the closest on the mesh. As this is an open area of research, when a robust triangle triangle intersection algorithm is found this will result in our method being able to correctly classify all points on meshes with self intersecting geometry.

Our method could also benefit from an improved hole filling algorithm. Not only does the current algorithm restrict the method both to manifold geometry and geometry which does not contain islands, but as is highlighted in fig 4.4, the advancing front mesh method implemented often flattens concave shapes. Whilst this still forms a valid closing of the mesh, it is often somewhat unintuitive. An possible solution would be to snap each new vertex to a radial basis function, thereby maintaining a more intuitive shape (Carr *et al.* 2001). Another possibility presented by Zhao *et al.* (2007) is to compute the desired normals for each new triangle added using the AFM technique. By solving the Poisson equation based on the hole vertices and desired normals the new vertices can then be repositioned so that they more appropriately fill the hole. The calculation of the desired normals however is dependant on whether the hole is closer to being planar or curved and thus the user must specify this for each hole.

The project could also benefit from further optimisations, particularly to reduce the precomputation time as at present no acceleration structures are used and many of the algorithms employ a brute force method.

Bibliography

Baerentzen J. and Aanaes H., 2005. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, **11**(3), 243–253.

Botsch M. and Kobbelt L. P., 2001. A robust procedure to eliminate degenerate faces from triangle meshes. In *Proceedings of the Vision Modeling and Visualization Conference 2001*, 283–290.

Bourke P., 1996. Klein bottle image. http://paulbourke.net/geometry/klein/.

C S., 1986. Procedural spline interpolation in unicubix. *Proceedings of the 3rd USENIX Computer Graphics Workshop.*

Carr J. C., Beatson R. K., Cherrie J. B., Mitchell T. J., Fright W. R., McCallum B. C. and Evans T. R., 2001. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, 67–76.

Elsheikh A. H. and Elsheikh M., 2014. A reliable triangular mesh intersection algorithm and its application in geological modelling. *Engineering with Computers*, (30), 143–157.

Ericson C., 2004. *Real Time Collision Detection.* CRC Press, San Francisco.

G. T. and C. W., 1998. Computing vertex normals from polygonal facets. *Journal of Graphics Tools*, **3**(1), 43–46.

J. L., Q. C. Y., Maisog J. M. and G. L., 2010. A new point containment test algorithm based on preprocessing and determining triangles. *Computer-Aided Design*, **42**, 1143–1150.

J. O., 1998. *Computational geometry in C.* Cambridge University Press, England.

Jacobson A., Kavan L. and Sorkine-Hornung O., 2013. Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph.*, **32**(4), 33:1–33:12.

Jun Y., 2005. A piecewise hole filling algorithm in reverse engineering. *Computer-Aided Design*, **37**(2), 263 – 270.

Laboratory S. U. C. G., 2013. The stanford 3d scanning repository. Available from: https://graphics.stanford.edu/data/3Dscanrep/.

Lorensen W. E. and Cline H. E., 1987. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 163–169.

Meister A., 1769. Generalia de genesi gurarum planarum et inde pendentibus earum ajfectionibus. *Novi. Comm. Soc. Reg. Scient.*, 180–189.

Meisters G. H., 1975. Polygons have ears. *The American Mathematical Monthly*, **82**(6), 648–651.

Möller T., 1997. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, **2**, 25–30.

NASA . Apollo soyuz.

Requicha A. G. and Voelcker H. B., 1985. Boolean operations in solid modeling: Boundary evaluation and merging algorithms. *Proceedings of IEEE*, **73**(1), 30–34.

Rossignac J., Fudos I. and Vasilakis A., 2013. Direct rendering of boolean combinations of self-trimmed surfaces. *Computer Aided Design*, **45**(2), 288–300.

Sabharwal C. L. and Leopold J. L., 2013. A fast intersection detection algorithm for qualitative spatial reasoning. *Research Journal on*

Computer Science and Computer Engineering with Applications, **48**, 13–22.

Sanchez M. Continuous signed distance field representation of polygonal meshes. Master's thesis, Bournemouth University, 2011.

Van Oosterom A. and Strackee J., 1983. The solid angle of a plane triangle. *IEEE Transactions on Biomedical Engineering*, **30**(2), 125–126.

Wei L.-y., 2013. A faster triangle-to-triangle intersection test algorithm. *Computer Animation and Virtual Worlds.*

Zhao W., Gao S. and Lin H., 2007. A robust hole-filling algorithm for triangular mesh. *Vis. Comput.*, **23**(12), 987–997.