THE SEQUENTIAL EDGE DETECTION ALGORITHM - A METHOD FOR DETERMINING THE PROFILE OF A VOLUME MESH

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Mechanical Engineering

in the

College of Graduate Studies

University of Idaho

by

Alexander P. Hanson

Major Professor: Jay McCormack, Ph.D.

Committee Members: Steven Beyerlein, Ph.D.; John Crepeau, Ph.D.

Department Administrator: John Crepeau, Ph.D.

May 2015

AUTHORIZATION TO SUBMIT THESIS

This thesis of Alexander P. Hanson, submitted for the degree of Master of Science with a Major in Mechanical Engineering and titled “The Sequential Edge Detection Algorithm – a Method for Determining the Profile of a Volume Mesh” has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor:	_____ Jay McCormack, Ph.D.	Date: _____
Committee Members:	_____ Steven Beyerlein, Ph.D.	Date: _____
	_____ John Crepeau, Ph.D.	Date: _____
Department Administrator:	_____ John Crepeau, Ph.D.	Date: _____

ABSTRACT

The algorithm presented in this work, known as the sequential edge detection algorithm (SEDA), finds exact perimeter for a given projection of a volume mesh through the use of vertex and edge data from the original mesh. The state of the art for determining this type of boundary is the alpha shape technique. Alpha shapes however are only capable of approximating the intended shape and in many cases do not represent the projection with sufficient accuracy. The Python implementation of SEDA presented in this work introduces localized vertex and edge searching methods to assist with mesh filtering and edge intersection detection. The implementation demonstrates SEDA's superior quality to the alpha-shape alternative and is able to determine the exact profile of very large and complex meshes and is ready for general use.

ACKNOWLEDGMENTS

Many thanks go to the faculty and staff of the Mechanical Engineering Department at the University of Idaho for their guidance, support and education.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE EXCERPTS

GLOSSARY

additive manufacturing

Method of creating parts through adding material layer by layer. Includes 3D printing and 3D prototyping.

Additive Manufacturing File Format

A XML document class written to extend the functionality of the aging STL file format.

alpha

A parameter that controls the allowable concavity of an alpha shape.

alpha shape

A linear piecewise representation of a set of point either 2D or 3D which allows for concave hulls.

AM

See additive manufacturing.

AMF

See Additive Manufacturing File Format

base (MBA)

The resulting body that will house both the embedded component and the cap.

cap

A preprinted part which provides a rigid flat surface for the printer to continue printing the remainder of the base. Conforms the embedded component to a valid printing surface.

CNC

See Computer Numerically Controlled.

embedded component

A part, such as a circuit board, which is completely encapsulated in a non removable manner into the base. See base.

exact perimeter

The set of modified edges and vertexes that define the perimeter of the precise profile.

Extensible Markup Language

A language that defines a common set of rules for encoding data which focus on being intuitive and readable. Originally defined for storing documents but is commonly used for more abstract data classes.

facet

See face.

intersecting edges

Edges that are not collinear but occupy some of the same space.

MBA

See Multi Body Assimilation.

mesh

A method to represent 3D objects through the use of points, lines, and faces.

MeshData

PyQtGraph's index array implementation for storing mesh data.

module

Python's method of grouping relevant code together into a single package.

Multi Body Assimilation

A proposed method to embed components with additive manufacturing.

OpenGL

An open source graphics API that focuses on speed and reliability.

PCB

See Printed Circuit Board.

preprocessing of mesh

Processes which eliminates duplicate or invalid vertexes / faces / and edges.

profile

The shape enclosed by perimeter edges of the simple projection of a volume mesh.

PyQtGraph

Python module responsible for providing Qt classes and applications capable of visualizing 2D and 3D data.

simple projection

The result of projecting all entities of a volume mesh onto a flat plane.

SLA

See stereolithography apparatus.

starting edge

A vertex found on the perimeter, which must contain the maximum x component.

An edge which contains the starting vertex and has the minimum angle between the y unit vector.

stereolithography apparatus

A type of 3D printer which uses a photo curable resin as the working material

stereolithography interface specification

File format created by 3D Systems to store mesh data. Has been universally adopted in

See stereolithography interface specification.

STL

See stereolithography interface specification.

unit normal

A vector with length (magnitude) of 1 and is normal to a face.

vertex

A point in 3D space represented with a x, y, and z location.

XML

See Extensible Markup Language.

CHAPTER 1: INTRODUCTION

While additive manufacturing (AM) has been around since the 1980s [7], the last decade has seen nearly 600% growth of system sales [1]; Most of this growth can be attributed to new processes, increased functionality, greater range of materials, and reduction in cost. The rise of AM has added multiple manufacturing capabilities ranging from complicated geometry never before conceivable on traditional CNC machines as well as the ability to vary material properties such as composition and density [8]. AM is no longer just for prototyping but is becoming more commonplace in many industries such as biomedical, molding and tooling, apparel, aviation, and dental. Table 1 shows a summary of current AM technologies and common applications. The medical field in particular has used AM to make life saving

Table 1: Review of AM material capabilities

Refs	Technology	Materials
[5]	Fused Deposition Modeling (FDM)	Thermoplastics (ABS, PLA, PC, PPS, PEI), Nylon
[5, 4]	Selective Laser Sintering (SLS)	Powdered thermoplastics, Polyaryletherketone (PEEK), Nylons (flame-retardant, impact- resistant, glass-filled, aluminum-filled, and carbon-fiber grades), Ti, Fe, Al
[5]	Stereolthography (SLA)	Photopolymers
[4, 6]	Selective Laser Melting (SLM)	Stainless Steel, Ti, Cu, Au, Ni, Fe, Al, Co
[4]	Laser Metal Deposition (LMD)	Ti, Ta, Ni, Fe, Co
[3]	Electron Beam Melting (EBM)	Ti, CoCr, TiAl, Inconel, Tool steel, Al,
[2]	Laminated Object Manufacturing (LOM)	Thermoplastics, Paper, Composites, Ferrous metals, Ceramics
[1]	Ink Jet Printing (IJP)	Photopolymers
[1]	Laser Engineered Net Shaping	Stainless Steel, Ni, Ti, Al, Cu, Tooling Steel, NiWC, Beryllium, Amorphous metals, Niobium, Invar

breakthroughs by producing everything from airway splints for newborns and prosthetics for humans and animals [9].

The AM industry has seen double digit growth for fifteen of the last twenty four years and the low-cost personal 3D printer market has grown by 289% [10]. The application of AM to direct manufacturing - the use of AM to make finished products, has grown from virtually 0% of total revenue in 2003 to 24% in 2011 [10]. Further development of direct manufacturing is dependent upon future innovations that will lead to printers that are cheaper, faster, more accurate, and with increased functionality. One such innovation is Multi Body Assimilation (MBA), which embeds pre-manufactured components during the printing process to produce one continuous part. Embedding components during the print allows for cleaner, lighter, stronger parts that do not require post assembly. This leads to

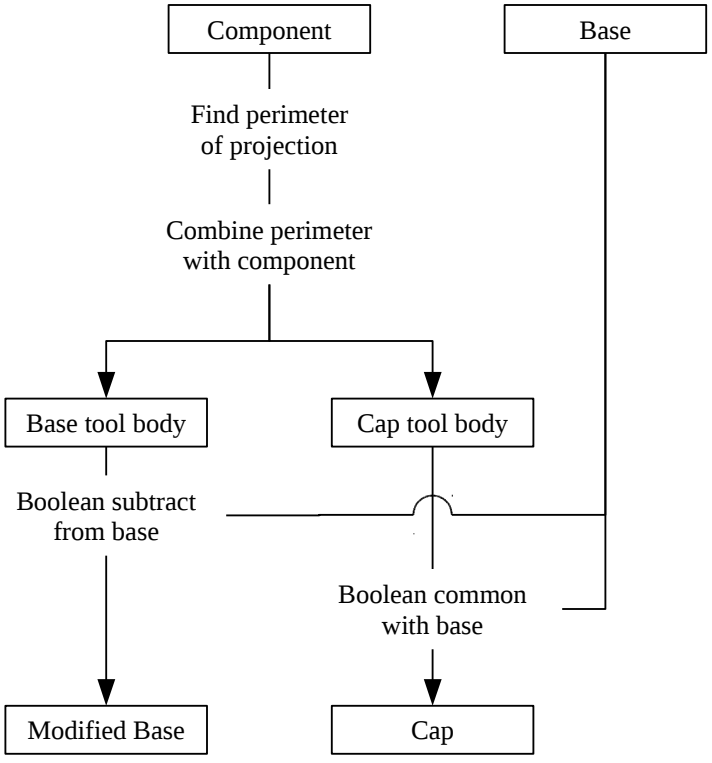


Figure 1: MBA Methedology

reduced part count, minimized labor, and allows for complex shapes not feasible with traditional manufacturing and assembly methods. Sensors, PCBs, screen and button interfaces, bearings, couplings, wearables, and printable circuits are just a few candidates for embedded components.

MBA is made up of three bodies, the base, the embedded component, and the cap. Fig. 3 shows the relative positioning of the bodies, note that the top and base comprise only one body but are shown as two for clarification. The base houses the embedded component with the help of the cap. The cap is responsible for providing a flat surface on which to continue

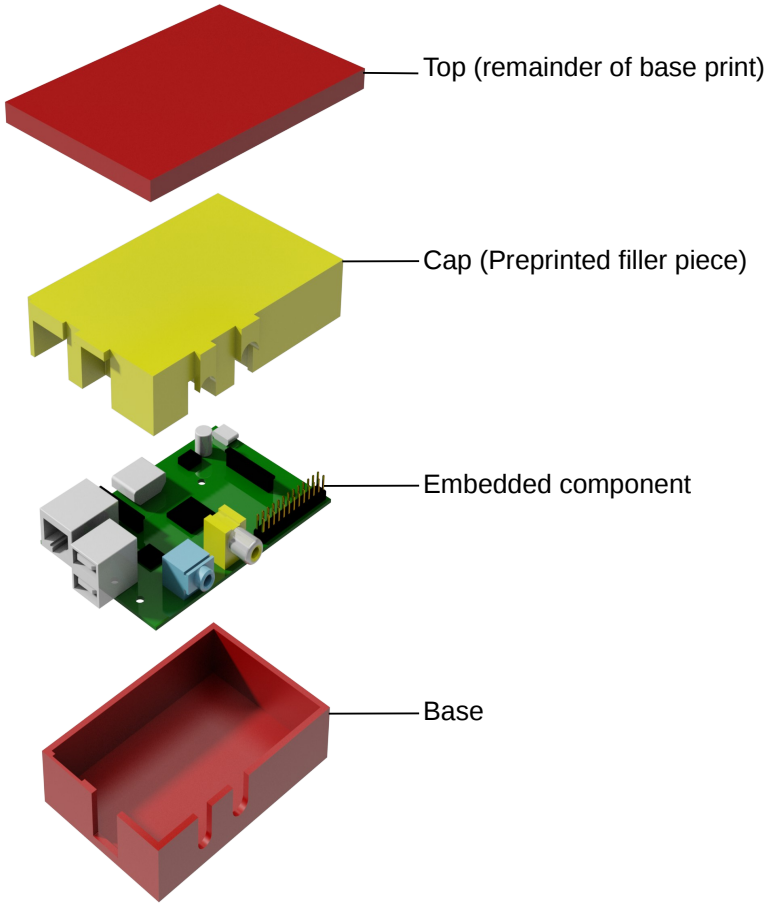


Figure 2: MBA Exploded Overview

the print after the embedded component has been inserted. The end work-flow is as follows: 1) the cap is printed and set aside for later use, 2) the base print is started, 3) the printer pauses and waits for component and the cap to be inserted, 4) the printer finishes printing the base, called the top in Fig 3.

Initial work on the MBA method focused on defining the bodies that comprise the cap and modified base. This initial work indicates that finding the exact perimeter of the simple projection, or the profile, is a critical step to ensure success. Fig. 3 shows a sample object and the profile. This thesis presents the algorithm developed to profile of the embedded component from a tessellated representation of the geometry.

CHAPTER 2: LITERATURE REVIEW

2.1 The STL Format

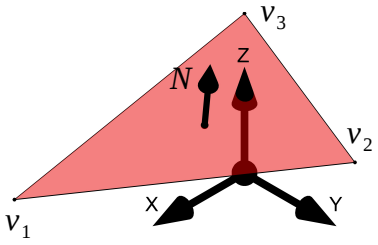
In 1988 3D systems published the stereolithography interface specification (also called standard tessellation language), or STL for short. to use with their SLA (stereolithography apparatus) printers [11]. Because of its relative simplicity and ease of use, STL quickly became the industry standard and is supported by almost every printer and CAD software [12], [13].

The STL format defines a 3D triangular mesh surface as an unordered list of triangular facets. Each facet consists of three vertexes (referred to as v1, v2, v3) and the unit normal (referred to as n) of the facet that indicates the exterior direction. Each vertex and unit normal are comprised of three floating point values: the x, y, and z components of a Cartesian coordinate systems. The specification also requires that the three vertexes are presented in the order in which when positioned at v2, the cross product of v1 and v3 (using the right-hand rule) produces a normal vector in the direction of the unit vector. Because of the requirement, many software packages chooses to ignore the provided unit normal and calculate new normals on import. Some software packages take advantage of this fact and use the normal to store other information such as color [14].

STL files are stored in two formats, binary and ASCII. The ASCII format is structured with keywords to help with formating. The ASCII file starts with the keyword *solid* followed by a required single space and optional solid name. Each facet is defined with the keywords *facet*

normal, *endfacet*, *outer loop*, *endloop*, and *vertex*. STL files stored in ASCII format can be very large so a binary format exists to address this issue. The binary format starts with an 80 character header which should not start with the keyword *solid* to prevent interpretation as an ASCII file. A 4 byte unsigned integer representing the number of triangles follows the header. Each triangle is in turn defined by 12 floating point numbers, x/y/z for the normal and x/y/z for the each vertex. Unlike the ASCII file, each triangle of in the binary file is then followed by two byte integer purposed as the *attribute byte count*. The STL specification does not document the use of the *attribute byte count* and only notes it should be set to zero. Many software packages take advantage of this fact and use the *attribute byte count* to store other information such as color and transparency [14]. A simple example for both formats is shown in Fig. 4.

Graphical Representation



v_1: $(20,\ -5,\ 5)$
v_2: $(-2,\ 10,\ 5)$
v_3: $(-5,\ -5,\ 10)$
N: $(v_2 - v_1) \times (v_3 - v_1)$
$= (75,\ 110,\ 375)$

ASCII Format

```
Solid example

  facet normal 7.50000e+01 1.10000e+02 3.75000e+02
    outer loop
      vertex 2.00000e+01 -5.00000e+00 5.00000e+00
      vertex -2.00000e+00 1.00000e+01 5.00000e+00
      Vertex -5.0000e+00 -5.00000e+00 1.00000e+01
    endloop
  endfacet

endsolid example
```

Binary Format

80 x UINT8 → Header
1 x UINT32 → Number of triangles in surface

3 x FLOAT32 → Normal
3 x FLOAT32 → First vertex
3 x FLOAT32 → Second vertex
3 x FLOAT32 → Third vertex
1 x UNIT16 → Attribute byte count

Figure 4: STL formats in detail

It is important to note that when STL files are used to represent real-world objects they are an approximation. The level of accuracy is dependent upon the number of faces that are used. Using a large number of faces increases the size of the data set and can cause longer processing time. A common example of this is rendering of 3D CAD models by the graphics card. While many CAD systems use boundary-spline representation to describe their models, the graphics card can still only draw the dot, and line, and triangle primitives. Most CAD systems have a system to control the triangle count in order to provide a balance between rendered accuracy and system performance. To demonstrate this principle, Fig. 5 shows a toroid with 200, 5,000 and 50,000 faces.

Another important note is that the STL format was created but not published nor maintained by 3D Systems. Efforts by this author to obtain a copy of the standard have not been responded to. The STL reader generated by this author and many others is based off of reverse engineering STL files created by other applications as well as non-published sources in respective communities.

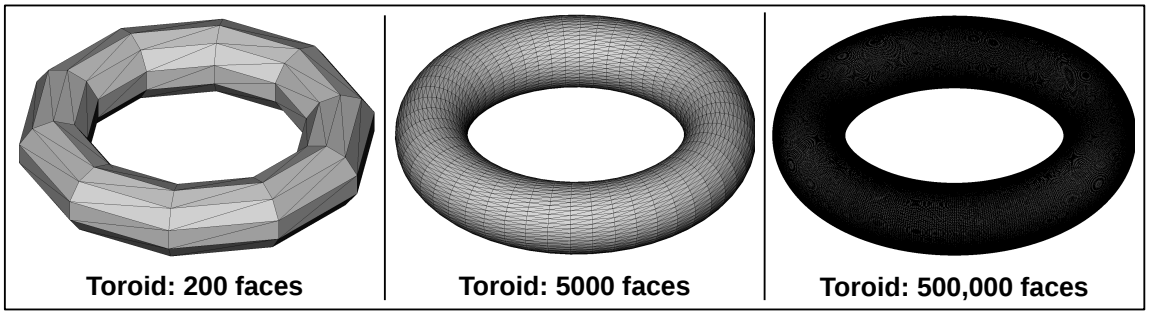


Figure 5: Toroid with varying face count

2.2 Additive Manufacturing File Format

The Additive Manufacturing File Format (AMF) is an attempt to overcome many of STL's shortcomings by creating an open standard that is Extensible Markup Language (XML) based. Specifically, AMF extends STL's capabilities by adding support for textures, color, materials, constellations (grouping of several bodies), and metadata. While AMF continues to store geometric data in a similar manner as STL, it also has the ability to support curved triangles [15].

Because AMF is an open specification and is XML based, it has the ability to support future innovation in the filed of 3D printing. As such, AMF would be a good candidate to add support for embedding components to 3D printing. While AMF will not be used in this thesis, it was mentioned to support the explanation of MBA.

2.3 Index Arrays and PyQtGraph's gl.MeshData object

Although the STL format is a convenient format to easily store and communicate a mesh for 3D printing, its structure is not ideal for complex mesh operations. Almost all mesh operations require manipulation of multiple facets or at minimum utilize information from surrounding facets. The STL format structure lacks any information pertaining shared edges, vertexes, or neighboring facets. While many methods for storing such information exists, a simple approach is to save all unique vertexes in one array, and to create second array that contains indexes referencing the vertex array to represent each facet. This model has also been extended to create arrays to store information about edges and to map vertexes to various grids to facilitate searches. Fig. 6 shows an implementation of this structure for a

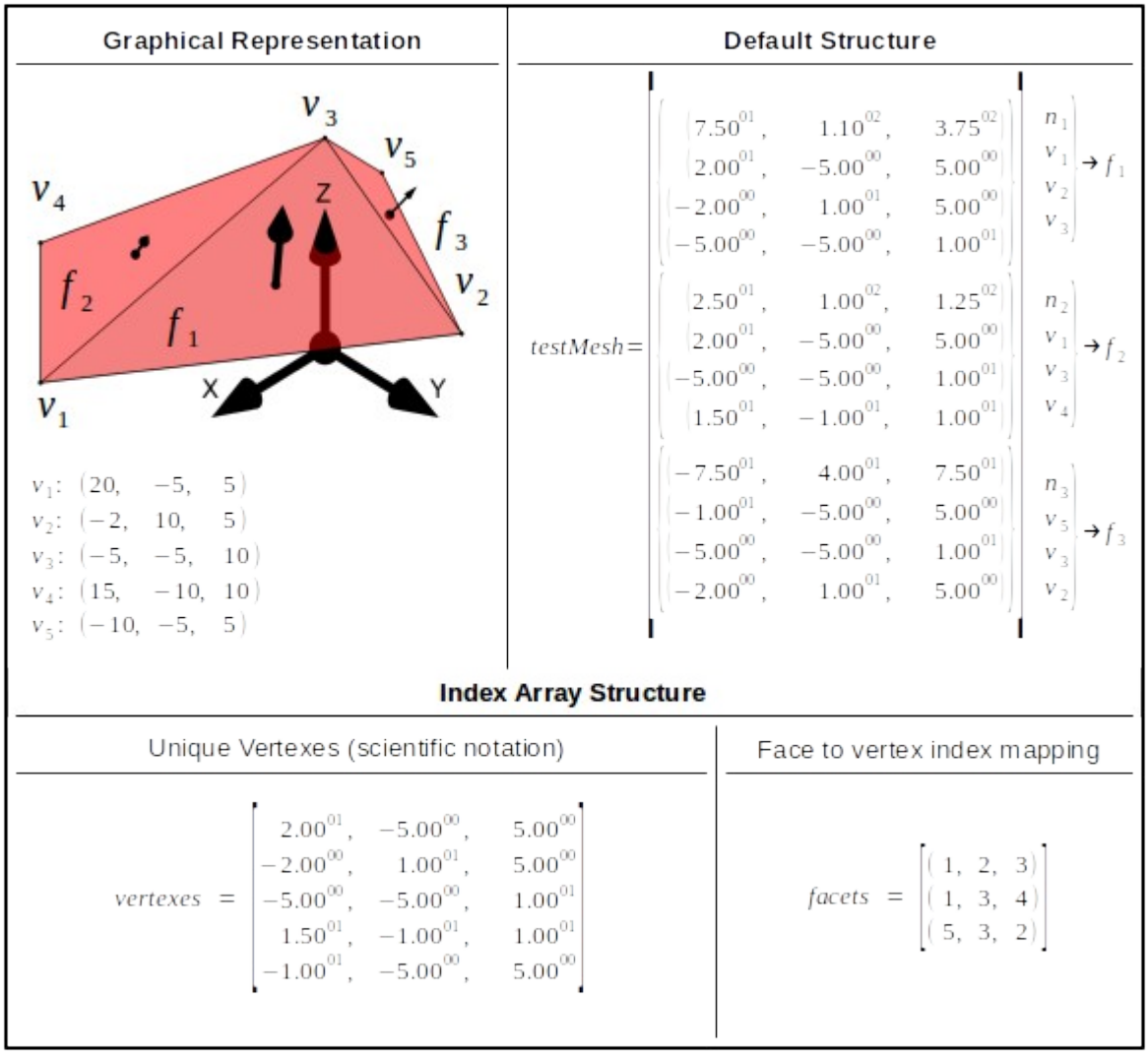


Figure 6: Representing meshes using index arrays

simple mesh.

It is important to note that the normal vector for each face has been disregarded. If use of the normal is often it may be saved to its own index array or combined with the faces array; however special attention to updating the appropriate face normals must be given when modifying vertex values or face vertex indexes.

The index array method of storing meshes is a borrowed effort from the graphics library

OpenGL and its attempt to increase efficiency between central processing unit (CPU) and graphics processor unit (GPU) [16]. Index arrays in this thesis have a dual purpose: to store meshes in a manner convenient for performing mesh operations and to be used with OpenGL's *glDrawArrays()* method to actually draw the shape to the display.

The OpenGL graphics library provides an application programming interface (API) to the GPU which enables the user to display 2D and 3D shapes through a processes called drawing. Unlike Microsoft's DirectX, the open source OpenGL is available on Windows, UNIX/LINUX, MAC, and embedded systems, as well as being accessible from Ada, C/C++, Fortran, Python, Perl, Java [16]. Although OpenGL is very powerful, it is considered to be a low-level tool-kit with a significant learning curve and requires a substantial investment to become proficient enough to produce sufficient applications. To overcome this barrier, a Python module by the name of PyQtGraph has been developed. PyQtGraph combines the powerful scripting language Python, OpenGL, and GUI toolkit Qt4 to create a “Scientific Graphics and GUI Library for Python” [17]. PyQtGraph excels at managing low-level OpenGL / Qt operations and allows engineers and scientists to quickly develop functional GUIs that are capable of real-time graphing of both 2D and 3D plots.

2.4 Python

Python is a programming language which is described as interpreted, powerful and fast, object orientated, and high-level. Because Python is easy to learn and supports many preexisting mature libraries written in C and C++, it has become a popular choice for many engineers and scientists. Python's principle of portability allows for rapid development of

solutions which utilizes community published modules. Modules are Python's way of organizing code into reusable chunks, much like libraries in C++. Fig. 7, is the ecosystem overview of Python(x,y), a popular collection of math and science related Python packages [18].

Variables

It is important to note that in Python, variables are simply names that refer to objects. This means that multiple names may be assigned to the same object, and that transferring one of the names to a different object will have no effect on the other name. For example, the names `a` and `b` may be assigned to the number 4, then `b` is be assigned to the number 5, but `a` will still be assigned to the number 4. Also, setting a variable equal to another variable will simply assign the first variable name to the object which the second variable is

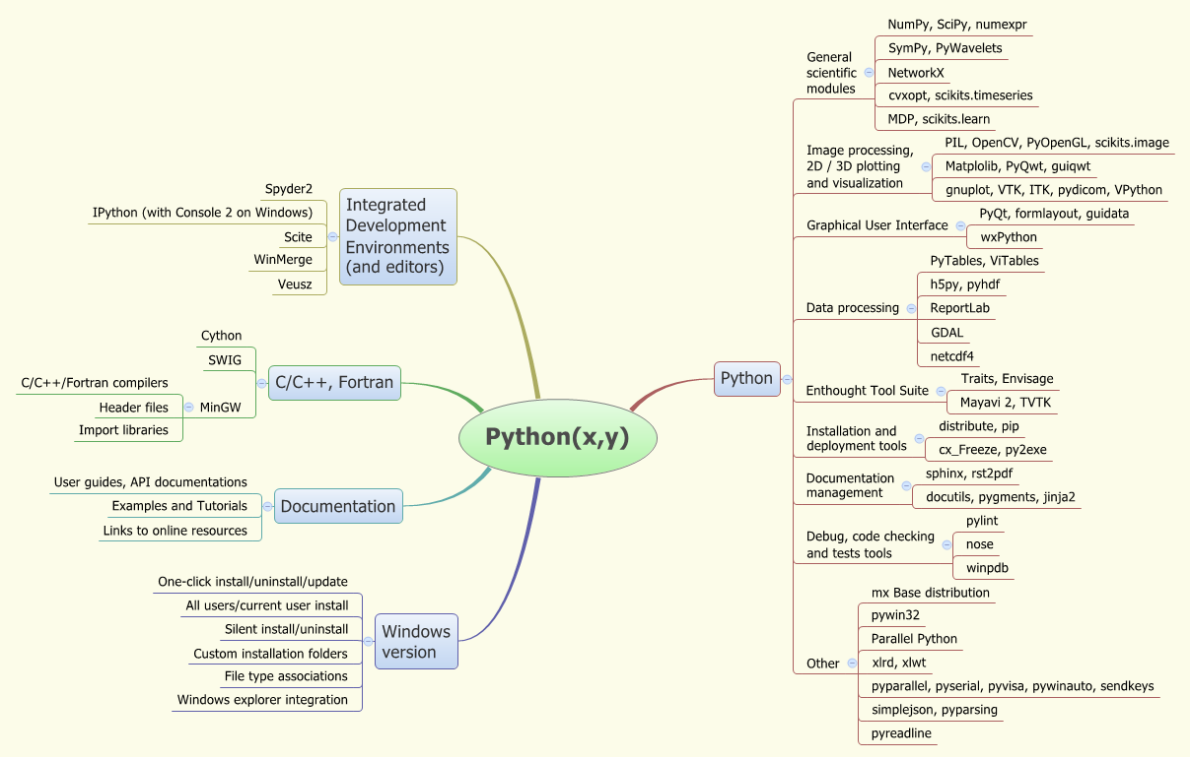


Figure 7: Python(x,y)'s overview of scientific and engineering Python modules

referencing [19]. A common analogy to help understanding is to visualize the variables as name stickers which may be moved around from object to object. Because variables are names there is no need to declare the type like in many languages such as C++ or VBA.

Indexing

The Python language indexes sequences starting at zero, meaning the valid indexes for a given sequence is **`0, 1, . . ., n-1`** where **`n`** is the number of elements in the sequence, which can be found by the **`.len()`** method.

Function Definitions

Python, like most languages, allows you to reuse code by defining functions. Excerpt 1 provides an example of a function declaration. Function definitions, or definitions for short, are initiated with the keyword **`def`** followed by a set of parenthesis which may optionally have input variables separated by commas, and lastly followed with a colon. Definitions may optionally return an object or a group of object and just like variables, the type returned is not declared in the function definition.

Excerpt 1: Function Definition Example

```
def foo(in1, in2):
    """
    An example function definition
    """
    bar = in1*2 + in2

    return bar
```

White Space

A critical concept in Python is the use of white space to control the structure of the code. An indentation of four spaces per sub-level is the standard. Please see the Function Definitions section for example of implementation.

Classes

Python is an object-oriented-programming language. Objects can have attributes (other objects) and methods (function definitions). Everything in Python is an object, this includes variables and the code that defines objects or function definitions. Objects are instances of class definitions. Class definitions start with the keyword **`class`** and the class name, followed by parenthesis which may contain names of other classes (for inheritance purposes), and lastly followed with a semi-colon. Excerpt 2 shows an example of a class definition. Python object, depending on the type, have built in methods whose name is

Excerpt 2: Class Definition Example

```
class Person():
    """
    An example class definition
    """

    __init__(self, name, age, sex):
        self.firstName = name.split()[0]
        self.lastName = name.split()[1]
        self.age = age
        self.sex = sex

    def timeLeft(self):
        """
        returns time left on earth based on US life expectancy
        """

        if sex == 'male':
            return 82.2 - self.age
        else:
            return 77.4 - self.age
```

generally surrounded by two underscores such as **__init__**, which is called when the object is initializes. In the example below, by defining **__init__**, we are overriding this built in function so we can set up the object as we see fit. We then define a method named **time_left()** which returns the expected remaining life in years based on US life expectancy for male and female. **self** refers to the object itself. This allows for saving object such as **age** so that it may be referenced later. Excerpt 3 shows an example of initializing an object, setting/referencing attributes, and calling a method using the dot, **.**, syntax.

Modules

A Module is Pythons technique of conveniently packaging code such as class and function definitions for later use. Modules are comprised of files, that when structured correctly through the use of folders, provide compact way to organize large program dependencies. One such example which is used extensively in the work is the PyQtGraph module. One class used iswhich is found inside of PyQtGraph's gl sub module. Like methods of classes, sub-modules and classes of sub-modules may be accessed through the dot syntax. Excerpt 4 shows an example of various methods of importing modules, sub modules and classes, as

Excerpt 3: Class Usage

```
#create instance of class Person known as author
author = Person(“Alexander Hanson”, 23, “Male”)

#the age is incorrect, change to 24
author.age = 24

#display the lastname
print(author.lastname)

#display years remaining
print(author.timeLeft())
```

Excerpt 4: Using Modules Example

```
import numpy as np
from geometry import GeoTuple, Vector
from pyqtgraph import opengl as gl

testArray = np.array([1,2,3])
testVector = Vector((1,1,2))

testMeshData = gl.MeshData()
```

well as creating instances objects form imported classes/modules.

Naming Conventions

While there is no official requirements for naming conventions in Python, this work has attempted to follow the widely accepted Python Enhancement Proposal (PEP) 8. An important part of PEP 8 is consistent naming conventions which has been reproduced here in Table 2 to facilitate readability of this work.

Table 2: Python's PEP 8 naming convention

Type	Convention	Example
Constants	ALLCAPS	`VERSION`
Variables	lowercase	`vertexColors`
Functions	lowercase	`create_triangle()`
Classes	CapWords	`Triangle()`
Modules	lowercase	`meshtools`

Other Rules	Convention	Example
Name is not to be accessed from outside of object	_single_leading_underscore	`_error_checker()`
Name conflict Python keyword	single_trailing_underscore_	`lamda_`
Name is unreadable	use_underscores_to_make_readable	`map_vertexes_to_grid()`

2.5 Alpha shapes

Alpha shapes are shapes created from a set of vertexes in either the 2D or 3D space which try to represent the original object. Alpha is a parameter that defines the maximum distance permitted between two vertexes that lie on the of the shape. The method starts with a Delaunay triangulation and uses the alpha parameter to eliminate edges to create what is known as a convex hull. An example of this is shown in Fig. 16. Alpha shapes are good for vertex sets that are uniformly distributed and avoid rapid directional changes in proportion the the alpha value. Because this paper focused on finding the exact perimeter and the removal of the third dimension causes large fluctuation of vertex distribution, the alpha shape solution is not accurate enough. The lower left of Fig. 9 shows is of the Stanford bunny, a common benchmark subject for geometric algorithms [20]. Three variations of detail 1 are shown, two alphas as wire-frame overlaid on the original and the perimeter edges in magenta as determined by the algorithm presented in this work. With an alpha

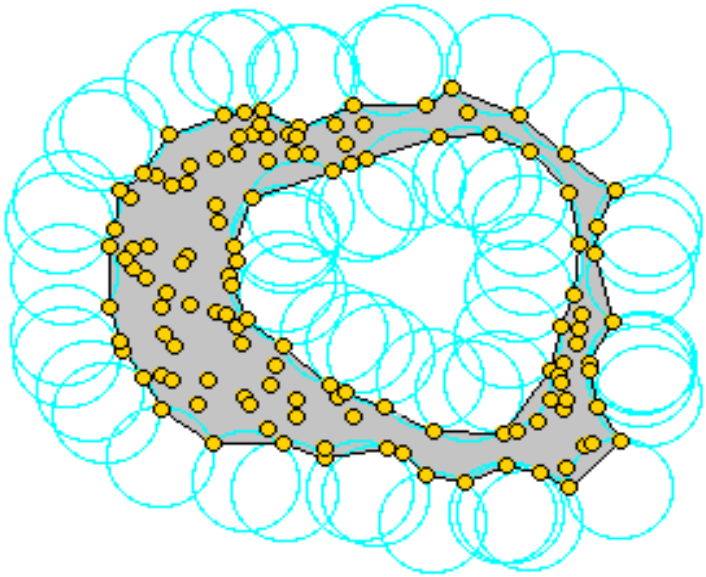


Figure 8: Creating the alpha, from [21]

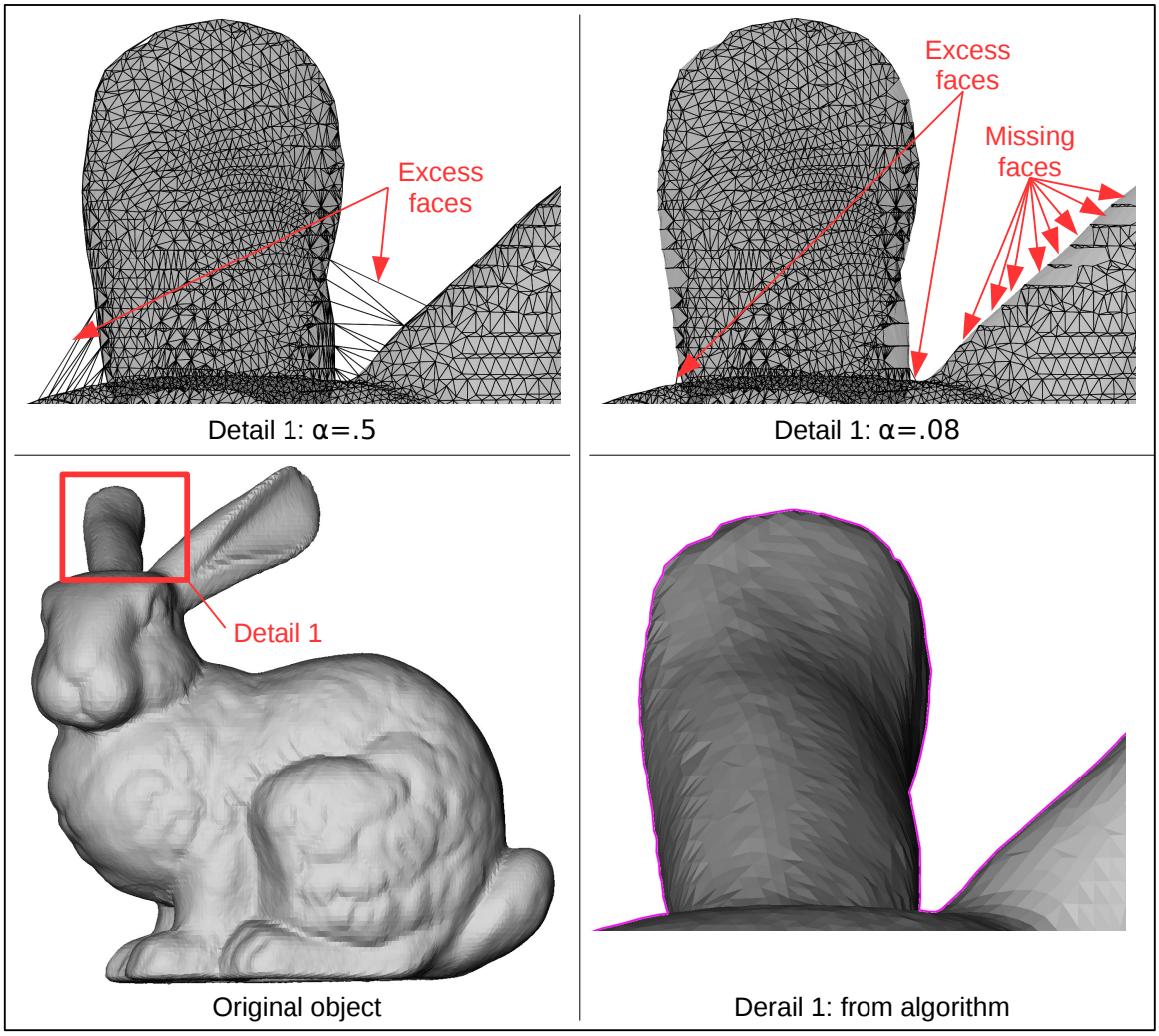


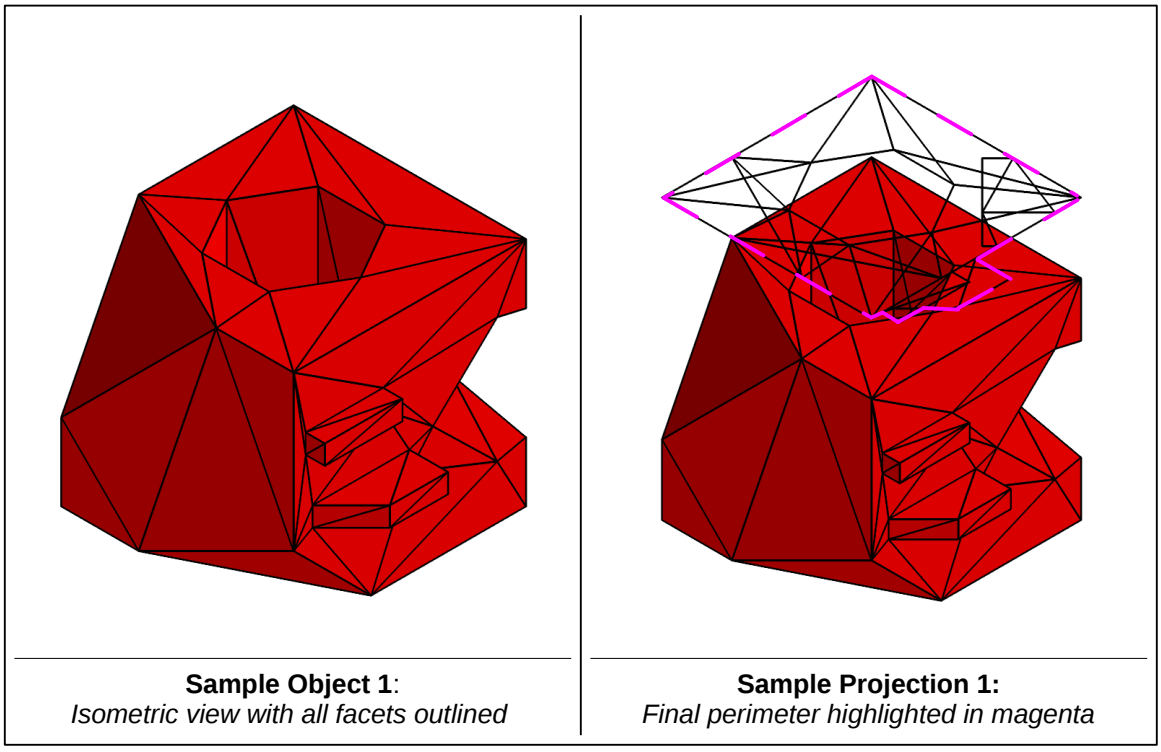
Figure 9: Alpha shape comparison

value of .5 (upper left) many areas of the bunny have excess faces, as shown here around the ears. Lowering the alpha value substantially is required to remove all excess faces. An alpha value of .08 (upper right) has removed all but two excess faces, but already many of the other faces have been removed. This characteristic of the alpha method is undesirable and not an issue in the method presented in this work as shown by the magenta line (lower right).

CHAPTER 3: SEQUENTIAL EDGE DETECTION METHOD

3.1 Overview

To produce the profile the proposed algorithm, Sequential Edge Detection Algorithm (SEDA) leverages not only vertex location but also edge configurations. Originally presented as Fig. 3 in chapter 1, Fig. Error: Reference source not found introduces the first sample object which will be used throughout the chapter to convey important features of the algorithm. The magenta-dashed line shows the perimeter edges that will be found by the algorithm. Sample object 1 started as a simple box with 8 vertexes, 18 edges, and 12 triangular facets. A through hole (hexagonal), a side pocket, two corner chamfers, and two side extrusions have been added to give a total of 52 vertexes, 156 edges, and 104 facets.



Sample Object 1:
Isometric view with all facets outlined

Sample Projection 1:
Final perimeter highlighted in magenta

Figure 10: Sample Object 1 - A simple box (left) with added features to increase complexity, and the simple projection (right) with detected perimeter edges.

Fig. 11 is a third angle projection of sample object 1 and details the added features. The view has been rendered with only the solid body faces outlined to help clarify the object's true shape. Feature (FTR) 1, a through hole orientated in the direction of the projection, is

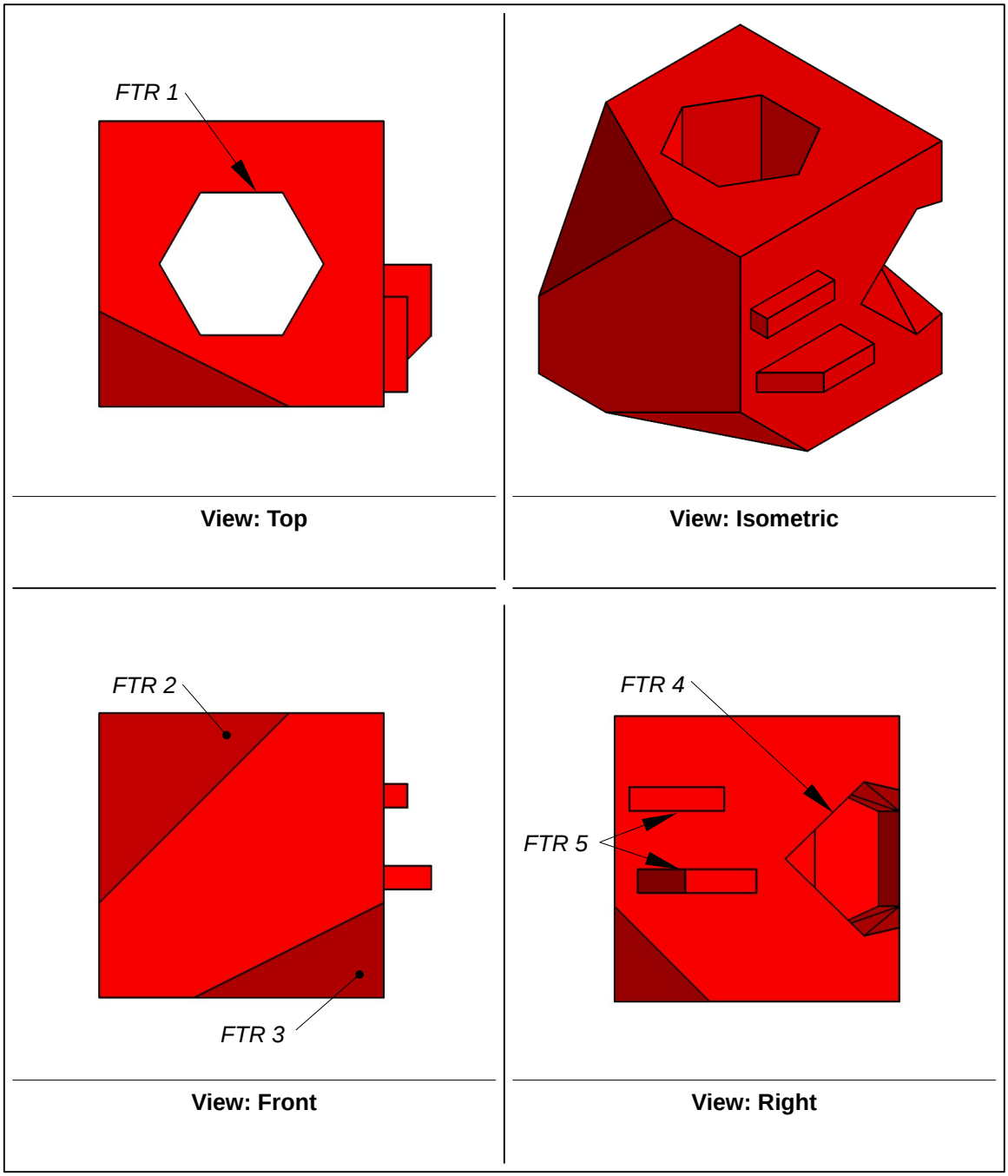


Figure 11: Sample object 1 – layout view

provided to tests the algorithm's ability to handle non-solid projections. FTR 2 and FTR 3 are corner chamfers to demonstrate choosing of a starting vertex and starting edge. FTR 4 is` a side pocket (hexagonal in shape) which tests the algorithm's capability to handle edges that lie on the same axis (of the projection). FTR 5 is made up of two overhanging bodies, that when projected, create intersecting edge, which is of critical importance to the algorithm.

The process of determining the profile starts with creating a simple projection of the original mesh, filtering the resulting projection through both vertex merging and face / edge removal, choosing a starting point of the perimeter, tracing edges along the perimeter, adding vertexes / edges when required, and detecting when complete. The described can be broken into two major events, per-processing and identification of perimeter edges, as show in Fig 3. Section 3.2 explains the preprocessing required for the algorithm while section 3.3 details the algorithm itself.

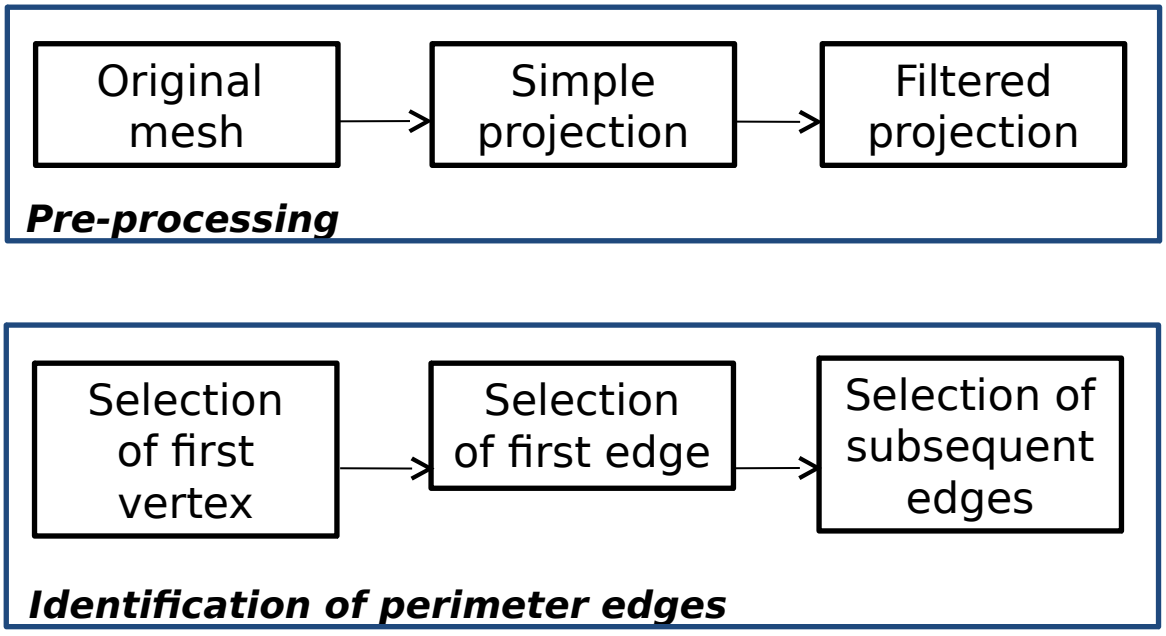


Figure 12: SEDA Overview

3.2 Preprocessing 2D projection

Removing the third dimension results in duplicate vertexes. This leads to duplicate edges duplicate faces, edges which no longer contain two unique vertexes, and faces which no longer contain three unique vertexes. All references to duplicate vertexes in faces and edges must be replaced with a reference to the unique vertex. After replacement of references, each face and edge must searched and removed if they themselves are a duplicate, or no longer occupy the x-y space. It is common for many projections to create clusters of vertexes which are the same vertex but because the nature of floating point integers register as different. It is therefore generally necessary to define duplicate vertexes based upon a distance parameter.

3.3 Perimeter algorithm

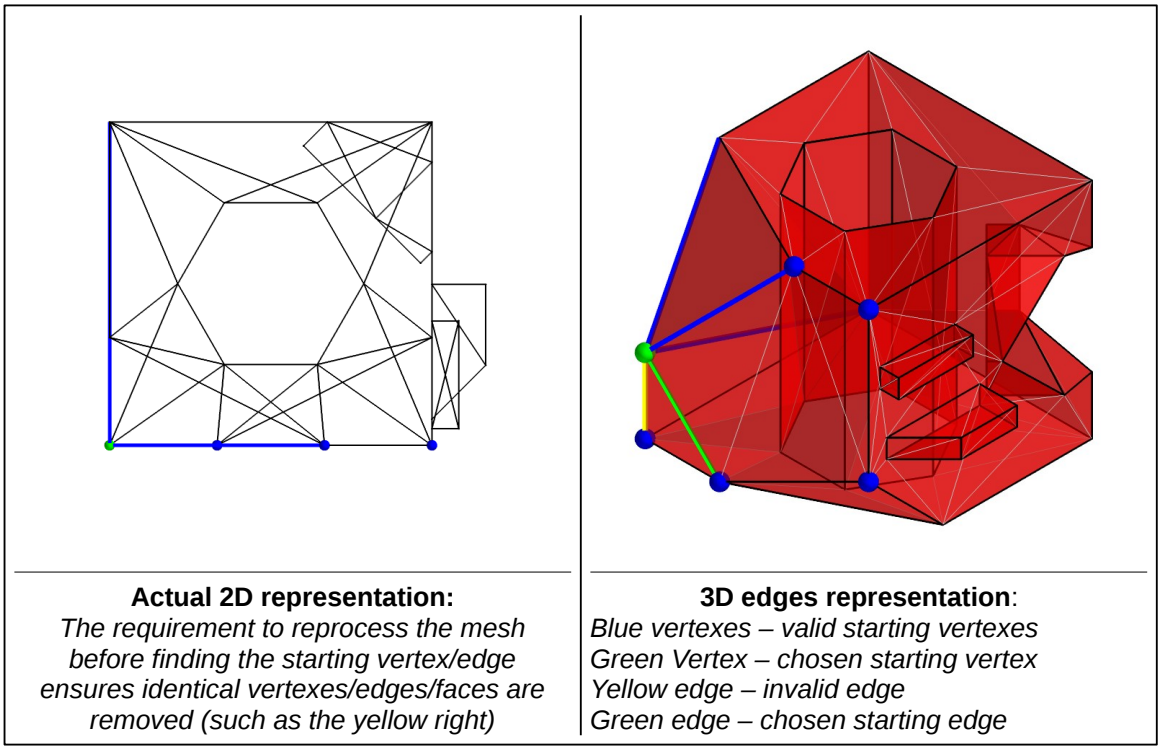
The algorithm is different in alpha-shape algorithms in that it attempts to find the exact perimeter by using edge information of the original object. The algorithm starts on a perimeter edge and by using simple geometric relations of surrounding edges, it selects the next edge, and continues this process until it reaches the starting edge. The following sections assume the projection is oriented orthogonal to the z axis.

Selection of starting vertex and starting edge

The first step is to find a starting edge. The starting edge is defined as the first edge that contains a vertex who's x component is equal to the global maximum x value and produces the minimum angle with the y-unit vector.

It is possible for multiple vertexes and multiple edges to fit the selection criteria, in this case,

the choice has no impact on the algorithm. The first edge that fits the criteria depends on the implementation. Fig. 13 highlights valid starting vertexes in blue, the chosen starting vertex in green, edges connected to the starting edge in blue, and starting edge in green. The right side of Fig. 13 maps the edges and vertexes from the 2D projection back to 3D space for visualization purposes only. The blue vertex connected to the yellow line and green vertex would have the same value after the projection and have been merged during the preprocessing of the mesh (as well as elimination of a face and edge). Please see section 21 for more information on mesh preprocessing. While all edges connected to the starting vertex of sample object 1 are valid perimeter edges, many object's starting vertex will have edges ending in the interior. Fig. 17 (left hand side) shows examples of edges and their angle with $\hat{j}$. Because the starting point must contain the greatest x component, the starting edge



Actual 2D representation:
The requirement to reprocess the mesh before finding the starting vertex/edge ensures identical vertexes/edges/faces are removed (such as the yellow right)

3D edges representation:
Blue vertexes – valid starting vertexes
Green Vertex – chosen starting vertex
Yellow edge – invalid edge
Green edge – chosen starting edge

Figure 13: Selection of starting edge and vertex

may have an angle with $\hat{j}$ that is at minimum 0° and at maximum angle of 180°.

Selection of subsequent edges; assuming no co-linear or intersecting edges

After the first edge has been determined, all of the edges of the perimeter may be found, one at a time, by calculating the cross product and the angle of each connected edge to the last chosen vertex / edge. The next edge is defined as the edges connected to the last vertex that has, in order of preference, 1) negative cross product and maximum angle with the last edge, 2) cross product and angle of zero, or 3) positive cross product and minimum angle. Fig. 14 (right hand side) shows the three cases when selecting the next edge. v_1 is the last vertex, v_0 is the second to last edges, and v_{2s} endpoints of edges connected to v_1. This method of selecting edges works for any projections with no overlapping or intersecting

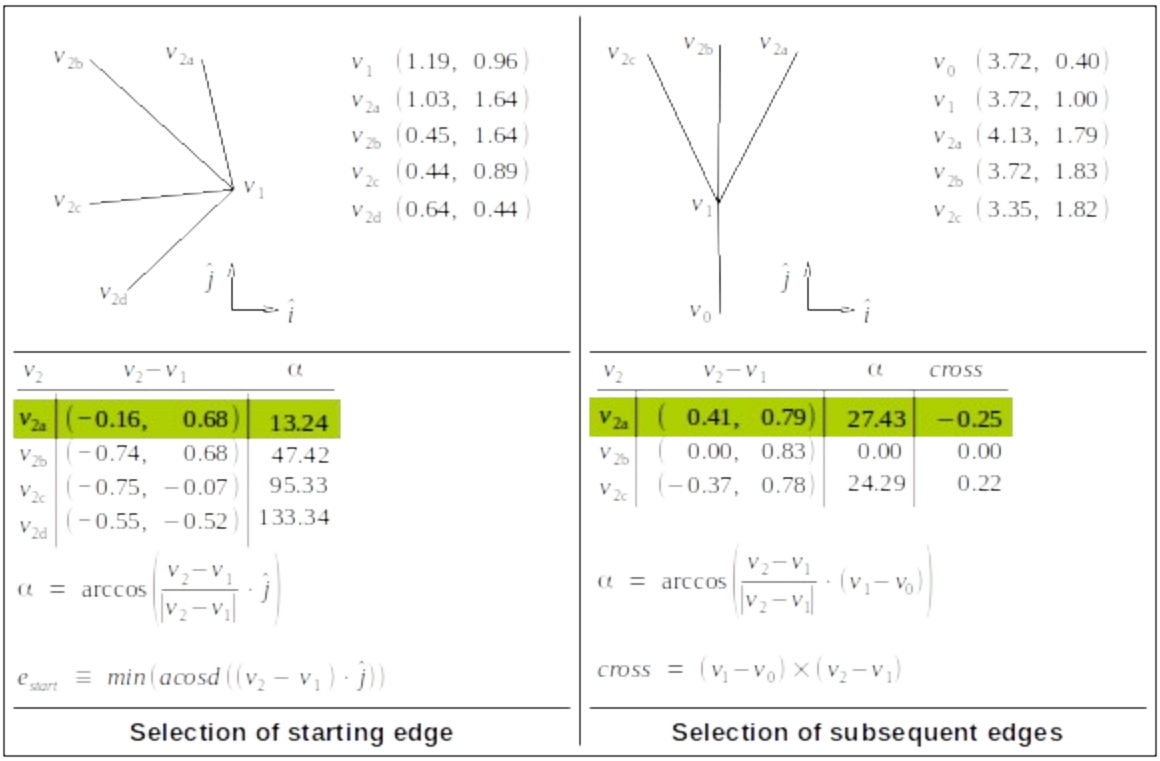


Figure 14: Selection of edges

edges.

Selection of subsequent edges; with collinear conflicts

Many objects' projections will produce edges which overlap or intersect other edges. Overlapping edges are caused by two edges, that when projected, are collinear, and where vertexes of one / both edges lie between the vertexes of the other. To ensure proper tracing of the perimeter the edge with the shortest length must be chosen. Note that this case only occurs when the edges connected to the last vertex have no negative cross product magnitudes. Fig. 15 (left hand side) is an example of collinear conflict. The previous edge is comprised of v_0 and v_1. Edges connected to v_1 are ended with v_{2a}, v_{2b}, and v_{2c}. The dashed gray line is an edge connected to v_{2a} which would ultimately be part of the perimeter. Both v_{2a} and v_{2b} have a cross product magnitudes of zero, but selection of

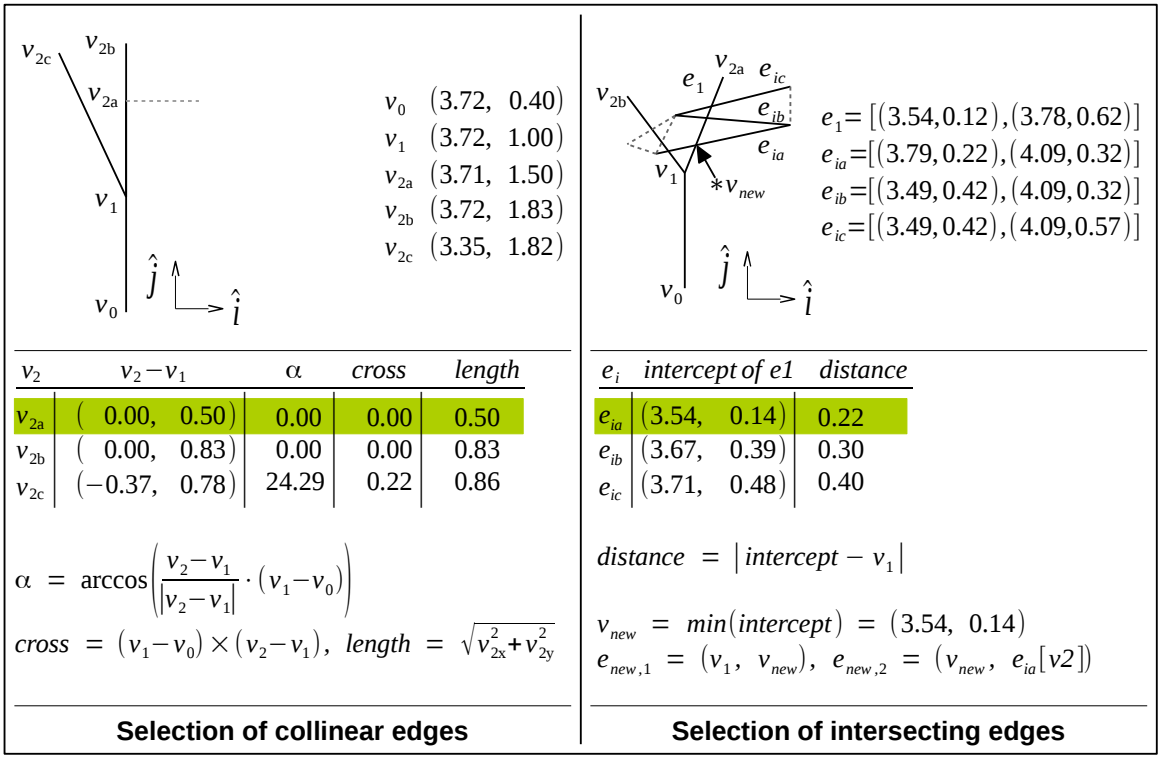


Figure 15: Selection of collinear and intersecting edges

v_{2b} would result in missing the dashed gray line. Selection of the collinear edge with the minimum length will ensure proper continuation of the perimeter.

Selection of subsequent edges; with intersection conflicts

Intersecting edges are caused from two discontinuous bodies, that in the projection, share some but not all space. FTR 5 (labeled in Fig. 11), produces intersections as seen in Fig 13, and Fig. 15 (right hand side) is an example of selecting the proper edge. The previous edge (Fig. 15) is comprised of v_0 and v_1. Edges connected to v_1 are ended with v_{2a} and v_{2b}. The next edge is first selected assuming no intersections occur as v_{2a}. If intersecting edges are found, a new vertex and two new edges are created and selected as perimeter edges. The new vertex, v_{new}, is chosen as the point of intersection which results in the first new edge, $e_{new,1}$, that has the shortest length. The second new edge, $e_{new,2}$, is from v_{new} to second vertex of the intersecting edge in question. Both $e_{new,1}$ and $e_{new,2}$ are added as perimeter edges, and the algorithm continues normally.

CHAPTER 4: PYTHON IMPLEMENTATION

The implementation presented in this paper is titled 'Mesh Projection Perimeter Processor' or in its package name as mesh_ppp. Table 3 overviews mesh_ppp's use of files, sub modules, Python Standard Library modules, and 3rd party modules. The author introduces four files within the mesh_ppp package: TriSurf.py, fileIO.py, tools.py, and mesh_perimeter_checker.py (MPP).

Table 3: mesh_ppp ecosystem

Refs.	Category	Name	Description
	mesh_ppp	TriSurf.py	Implements mesh filtering, perimeter algorithm, and mesh storage through pyqtgraph.opengl's MeshData
		fileIO.py	Handles reading of STL geometry data into np.array
		tools.py	Misc. tools, such as progress bars for the terminal
		perimeter_mesh_checker.py (PMC)	Script which implement pyqtgraph's GUI and OpenGL module to allow for verification of TriSurf methods
17		pyqtgraph	Handles 3D plotting with OpenGL and QT
22		Poly2Tri.py	Handles tessellation of perimeter mesh
23	Python 2.7 Standard Library	os	Assists TriSurf in displaying messages to the terminal
23		argparse	Assists PMC in parsing arguments from system terminal
23		distultils	Assists with distribution of mesh_ppp with pip
23		__future__	Provides compatibility with Python3 for print and division
23		time	Assists TriSurf with debugging
23		collections.defaultdict	Assists TriSurf with mapping hashable keys to iterables
23		itertools.combinations	Assists TriSurf with non-repeated combinations of iterables
23		struct.unpack	Assists fileIO with unpacking binary into byte data
24	3rd party	numpy (np)	Assists mesh_ppp with fast array structure and operations
25		termcolor.colored	Assists TriSurf with producing colored output to terminal

4.1 fileIO.py – Importing STL files

This file defines two methods, `_read_stl()` and `import_stl_file()`, the later which is designed to be called be other classes in mesh_ppp. The `import_stl_file()` method is called passing the file name of the desired STL file to read, and is responsible for checking that the extension of the file name ends in '.stl', calling `_read_stl()`, and returning the mesh geometry and number of triangles. The `_read_stl()` method opens the file into memory, reads the file header, number of triangles, and mesh geometry. The mesh geometry is saved into a numpy (np) array (see table 3), with an entry for each facet. Each entry is in turn comprised of four np arrays, each with three floating point number of 32 bit accuracy, and a two byte integer. The four arrays hold vectors that define the normal and the three points of the face, while the integer is the STL standard's non-specified n_attr field. The data from the returned np array may accessed through standard slicing or by naming the field with slicing. The valid field names are `'norm'`, `'v1'`, `'v2'`, `'v3'` and `'n_attr'`. The conversion of binary to byte is handled through np's `fromfile()` method.

See APPENDIX A: fileIO.py for source.

4.2 tools.py – miscellaneous classes

At the time of writing, tools.py defines a single class `ProgressBar()` which is used to display the progress of tasks, such as mesh filtering, to the terminal. The class is initiated with the name of the task, the length of the task, and the number of divisions to display in the progress bar. The progress bar may then be updated with the `update()` method by passing in the current progress of the task. Any printing to the terminal during use of the progress bar

will case irregularities in the display of the progress bar.

See APPENDIX B: tools.py for source.

4.3 TriSurf.py – implementation of method

TriSurf.py defines the TriSurf class which builds upon pyqygraph's gl.MeshData class's storage and indexing capabilities by defining methods necessary to implement the perimeter detection algorithm. The TriSurf class inherits from the pyqtgraph.opengl.MeshData class. This means all methods and attributes defined in MeshData are accessible to TriSurf. Of most significance is MeshData ability to be accepted by pyqtgraph's gl.GLMeshItem class, which is responsible for preparing mesh items to be drawn to the screen.

TriSurf is organized with **`@property`** methods appearing first, followed by **`_create_property()`** methods, then **`add_item_to_property()`** methods, then **`get_some_info()`** methods, then **`return_some_TriSurf_()`** methods, and finally by miscellaneous helper methods. The methods preceded with the **`@property`** decorator are designed to protect the attribute they return from being accidentally overwritten. These methods check to see if an attribute (generally named the same as the method but preceded with a single underscore character) is of type **`None`** and will call the corresponding create property method (generally named the same as the attribute but preceded with _create) if so. Finally, the method with return the attribute. The **`add_item_to_property()`** methods are used to ensure that when the respective attribute is updated, other attributes that are dependent upon the change are updated accordingly. The **`get_some_info()`** methods are

used to return data, generally np arrays, derived from attributes. The methods who's names are preceded by the word return create and return TriSurf instances using information from attributes or `get_some_info()` methods.

Many of the attributes are of type `defaultdict(list)` which provide mappings of vertexes and edges to grid locations. This type of dictionary uses hashable keys (such as a x/y cell tuple) and lists for the values (such as a list of vertexes residing in the cell). These mappings provide a fast convenient manner access of such information and play a vital role in filtering the mesh and determining the perimeter.

Initialization

Other important functions used from gl.MeshData are the initialization method `__init__()`, the `vertexes()` method, the `edges()` method, and the `faces()` method. The initialization method accepts mesh data formatted various manners including the vertex form that `import_stl_file()` returns. If it has not been done before, the `vertexes()` method will remove duplicate vertexes, create index mapping for both faces and edges, and then return the array of unique vertexes. The `edges()` and `faces()` methods return arrays which contain indexes to vertexes. The `vertexes()` method is called in TriSurf's initialization function to ensure calls to `edges()` and `faces()` do not result in any unexpected behavior.

Projection and filtering of mesh

The projection of the original mesh is handled by `return_projection()`. The method gets a copy of the vertexes in facet form with `vertexes(indexed='faces')`, sets all z

components to zero, and then returns a new TriSurf instance created with the modified set of vertexes.

Upon creation of the new projected TriSurf, vertexes which are 1e-14 from each other will merged and the resulting array of unique vertexes may be less in length then of the original TriSurf instance. Furthermore, when near vertical edges (parallel to z-axis) of the original TriSurf are projected the often result in clusters of vertexes. These clusters contain edges between vertexes of the cluster, and edges between clusters. Detection and handling of intersecting edges greatly increases the time required, and it is therefore necessary to further filter the projection. A fine grid was created (based upon the delta parameter) to map the space occupied by the model. Each vertex for each cell of the grid is then compared to all vertexes of neighboring cells. Any vertexes, as shown in Fig. 16, that are a distance of delta or less are merged. This is handled by the `return_filtered_mesh(delta=.01)` method

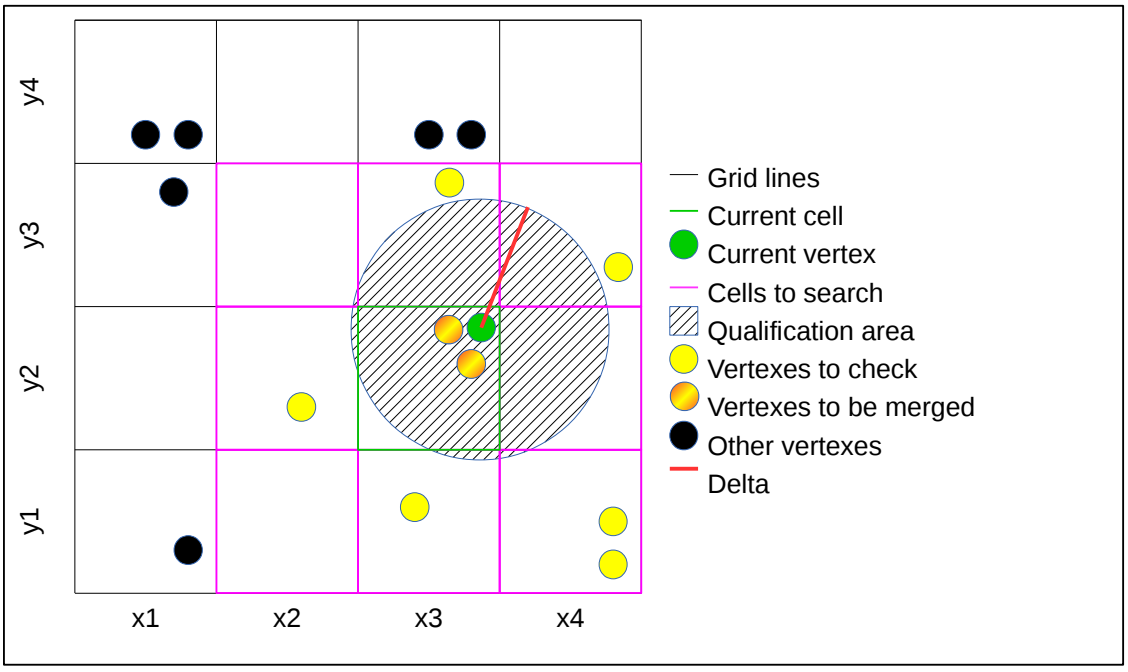


Figure 16: Vertex filtering method

which replaces vertex references in the faces array, removes invalid faces, and then returns a new TriSurf instance.

In order to reduce the number of vertexes that each vertex must be compared to, each vertex is assigned a cell location. This information is stored in two mappings, the first maps each vertex to a cell, and the second maps each cell to a list of vertexes. The first mapping, stored as `vertexes_to_cells` is created by `_create_vertexes_to_cells()`. Based upon the delta variable and information about the mesh's maximum width and height, this method determines appropriate cell sizes and assigns each vertex a cell location. For each vertex, the x cell location is set as $x_{col} = floor((vertex[0] - x_{min}) \ / \ x_{col\_size} + 1)$ and the y cell location as $y_{row} = floor((vertex[1] - y_{min}) / y_{row\_size} + 1)$. The calculated cell location is saved as a tuple to `vertexes_to_cells` with the vertex index as the key. Similarly, the `cells_to_vertexes` mapping stores the same information but uses the cell location as the key, and a list containing vertex indexes as the value.

The `return_filtered_mesh()` method starts by looping through each cell in the `cells_to_vertexes` mapping. For each cell a list of neighboring cells is compiled, and from this a list of all vertexes in the cell and neighboring cells is compiled. For each combination of vertexes if the difference of both the x and y components are less than delta, it is assume a duplicate vertex has been found. The defauldict(list) mapping `duplicate_to_unique_vertexes` is used to store this information. The first vertex of the combination is added as the value (unique vertex) with the second vertex as the key (duplicate vertex). To ensure the vertexes which are duplicates of other entries are not used

as uniques values, it is checked if the first vertex is a key of the mapping. If it is found as a key, it is set equal to the value of its entry (the unique vertex), and then mapped to the second vertex of the combination. As an example assume vertexes v1, v2, and v3 shall all be mapped together. The possible combinations are v1-v2, v2-v3, and v1-v3. First v1 will be the value and v2 will be the key. Secondly, v2 will be found as a key, so v1 (the value if v2) will be the value for the key v3. The third combination will have no effect because v1 is already found as the value to the key v3. This process continues until the for loop is complete.

Next, each vertex of each face is checked as a key of the duplicate vertex mapping, if an entry is found, the vertex is then replaced with the value. Any duplicate rows are then removed by the trisurf.py function **`get_unique_rows()`** (not a method of the TriSurf class). Each face is then checked to ensure that three unique vertexes exist. The final list of faces is then converted to vertex float form through **`get_vertexes_from_faces()`**. Finally, the **`return_filtered_mesh()`** method ends with returning a new TriSurf instance from the vertex array.

Determination of perimeter edges and vertexes

The method **`get_perimeter_edges_and_vertexes()`** is responsible for identifying and creating perimeter edges and vertexes. Because it is necessary to create vertexes and edges, the original set of vertexes and edges are duplicated and saved as **`modified_vertexes`** and **`modified_edges`**. All mappings and methods related to this modified set contain the word modified in their name. As perimeter vertexes and edges are identified, they are saved

to the two lists **perimeter_vertex_indexes** and **perimeter_edges** which contain indexes of the modified sets.

The method starts by selecting the starting point using the TriSurf property **x_max**. All edges which contain the starting point may be returned with **np.where()** method, which accepts a truth statement between the array to search and the value. To help choose the correct edge, the **get_connected_modified_edges_data** method has been written to return the vertex indexes, vertex floats, angle and cross product magnitude between the edge and the pivot edge, and the length of the edge. The method accepts a list of edge indexes (from np.where), and either a starting vertex index, or a pivot edge index. For selecting the starting edge, the starting vertex index argument is used and the angle and cross product magnitude of each edge will be with the y unit vector. Using the information returned, the starting edge may be chosen as the edge with the minimum length of the set of edges with the minimum angle.

All following edges are selected with the following process:

1. Determining all edges connected to the last vertex
2. Getting connected edge data with using the last edge as the pivot edge.
3. Select a candidate edge as:
    3.1. Negative cross product magnitude: minimum length edge of maximum angle set
    3.2. Zero cross product magnitude: minimum length edge
    3.3. Positive cross product magnitude: minimum length edge of minimum angle set
4. Check if the candidate edge is the beginning edge (algorithm will stop)

5. Save selected edge and vertex to the perimeter lists
6. Checking if intersecting edges exist
    6.1. Selecting the intersecting edge who's intersection point is the closest to the 2nd to last perimeter vertex
    6.2. Addition of the intersection point to modified vertex set
    6.3. Addition of the edge between the new vertex and the 2nd to last perimeter vertex to the modified edge set
    6.4. Identification of the vertex of the intersecting edge which has a negative cross product magnitude between v1 and v2 where v1 is the difference vector between itself and the 2nd to last perimeter vertex and v2 is the last edge of the modified edge set.
    6.5. Addition of the edge between the new vertex of step 6.2 and the identified vertex of step 6,4
    6.6. Removal of the last vertex and edge of the perimeter vertex / edge sets.
    6.7. Addition of the last two edges of the modified edges set to the perimeter edges set
    6.8. Addition of the last vertex of the modified vertexes set and the vertex identified in step 6.4 to the perimeter vertexes set
7. Repetition of step 6 until no intersections are found
8. Repetition of steps 1 through 7 until step 4 is true

Finding intersecting edges is handled by **`get_intersecting_modified_edges()`**. This method requires two arguments, the edge index you wish to find intersections of, and the starting vertex index (2nd to last perimeter vertex). This method leverages mappings of the modified vertexes and edges to a search grid much like the mappings to filter the mesh, however the delta parameter is based on the maximum x and y edge displacements. This grid is many orders of magnitudes coarser than the previous search grid and is used to isolate each edge with a finite set of neighboring cells that no intersecting edge can completely lie outside of. Fig. 17 shows examples of the three possible candidate edge (blue lines) configurations, completely contained in a single cell (left), starting and ending in two cells that share a common edge (middle), and starting and ending in two cell that share only a corner (right). Fig. 17 clearly shows that intersecting edges (red lines) do not have to start or end in a cell that the candidate edge resides in (outlined dashed blue). In fact, any edge that both starts and ends in cells that neighbor the start and end cells of the candidate edge, could result in a possible intersection. In this context, a neighboring cell is any cell that shares an edge or vertex, which gives 9, 12, or 14 cells to search depending on the configuration (shown as red hatching). Any line which is not completely contained in the neighboring

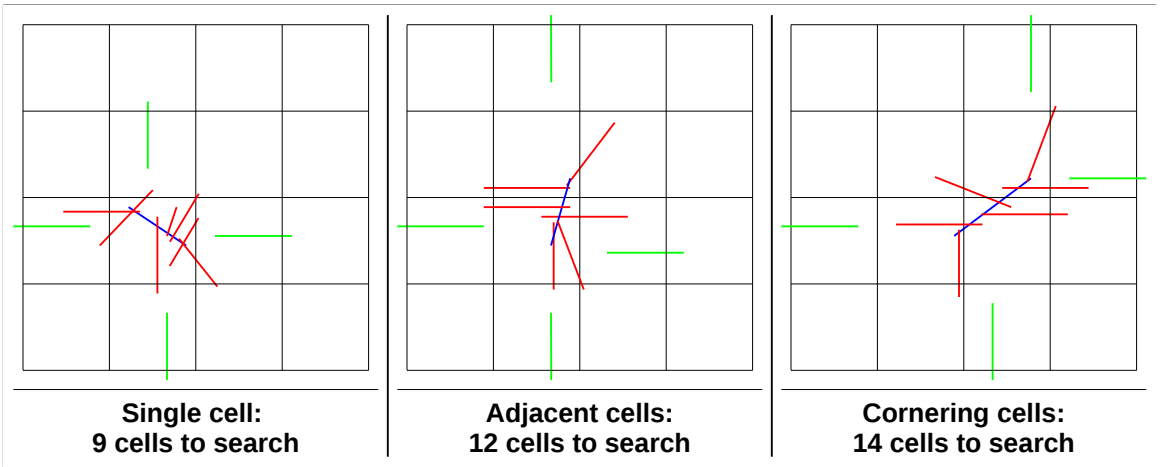


Figure 17: Grid search for intersecting edges

cells(green lines) are not possible intersections. The process is much the same as the mesh filtering, from the edge's cell location(s), a set of 9, 12, or 14 cells are compiled and with **`modified_cells_to_edges`** mapping a list of all possible intersecting edges is compiled. Using the method **`does_intersect(pivot_edge, edge)`** , each edge is checked for intersection, if an intersection exists then the intersection location, distance from the starting vertex, and angle is calculated. This information is then returned back to **`get_perimeter_edges_and_vertexes()`** .

Creation of perimeter mesh

The **`return_perimeter_mesh()`** method uses the list of perimeter vertexes returned by **`get_perimeter_edges_and_vertexes()`** to create a tessellation with the aid of **`Poly2Tri.Triangulator()`** . The **`triangles()`** method of the Poly2Tri class then returns the vertex representation of the perimeter mesh which is then used to return a new TriSurf instance.

See section APPENDIX C: TriSurf.py for source.

4.4 perimeter-mesh-checker.py – graphical verification

This file uses the pyqtgraph module to create a simple plot window and draw 3D geometry. It makes use of pyqtgraph's opengl plotting methods and is designed to be used from the terminal and accepts the following parameters:

•**file name** (required) – folder path and filename of desired STL file

•**--delta** or **-d** – mesh filtering parameter, accepts a single float

•**--original** or **-o** – flag that causes original object to be drawn to the screen

•**--projection** or **-p** – flag that causes the projection to be drawn with perimeter edges highlighted. Will also cause the perimeter mesh not to be drawn

An example of use:

```
python –i perimeter_mesh_checker.py ../Resources/sheep.stl –o –d .035
```

The -i python argument is required to prevent python from exiting after finishing drawing geometry (and to allow the user to interact with the scene).

See APPENDIX D: perimeter_mesh_checker.py for source.

CHAPTER 5: VALIDATION

5.1 Sample object 1: algorithm_test_piece.stl

The algorithm test piece, originally presented above, is shown in Fig. 18 to illustrate the major steps of the algorithm. The relatively simple example originally consisted 104 facets while the final perimeter mesh consists of 8 facets. Identification of the perimeter edges was

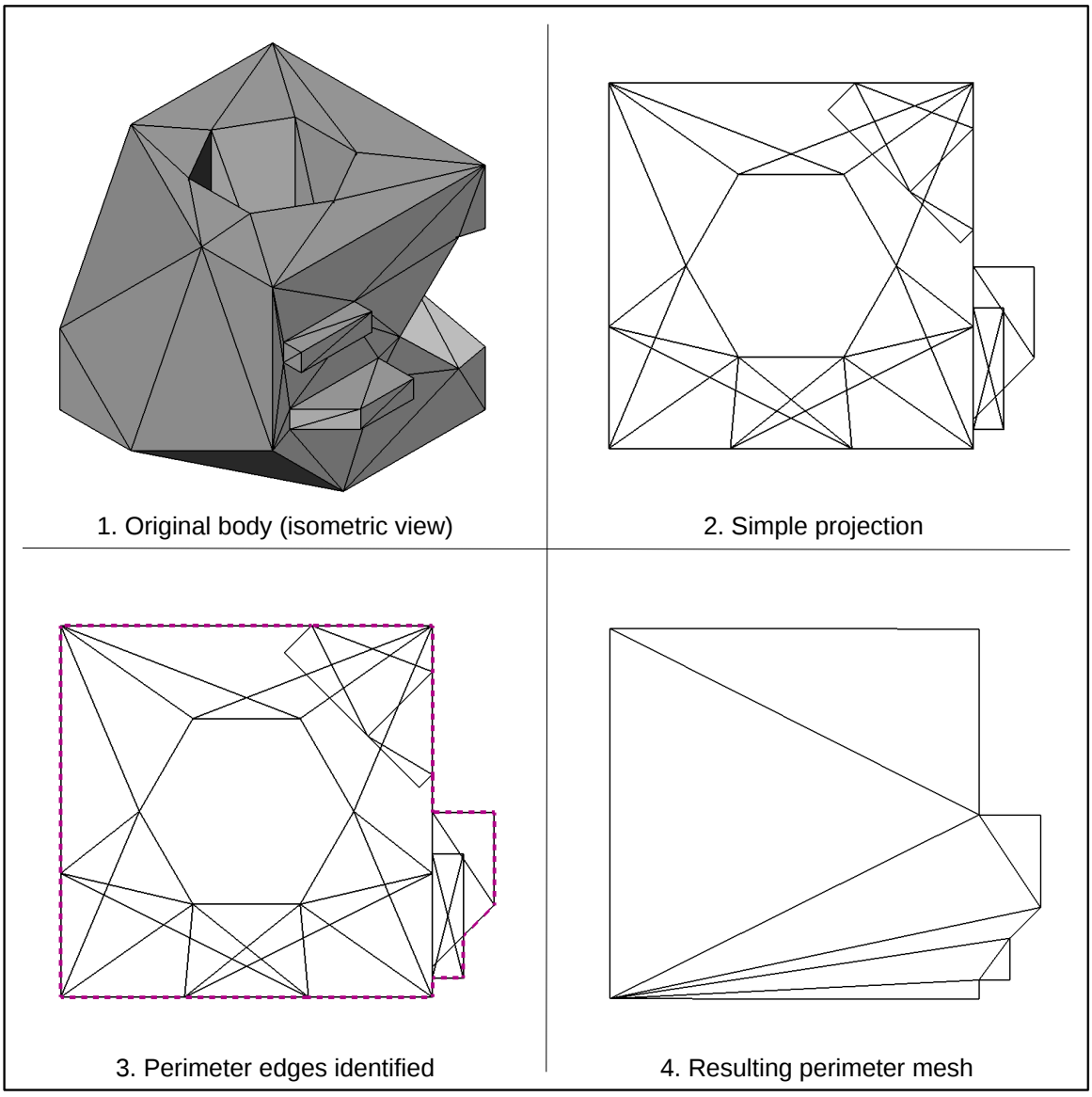


Figure 18: algorithm test piece

accomplished in .03 seconds. The perimeter mesh is a tessellation of the perimeter edges that results in the least number of facets (two less than the number of edges). The many non-organic features of this test piece helped develop the functionality of the algorithm such as the ability to handle edges that overlapped exactly, as well as having multiple valid starting points and edges.

5.2 Sample object 2: sheepWHITE.stl

The sheep is a computationally more complicated object which contains 162476 faces, 243714 edges, 81240 vertexes. The sheep is comprised of many intersecting spheres which provides a very organic test shape. Filtering the simple projection removes 24596 vertexes, and 36010 faces resulting in 162476 faces, 243714 edges, 81240 vertexes. This is a 22.2%,

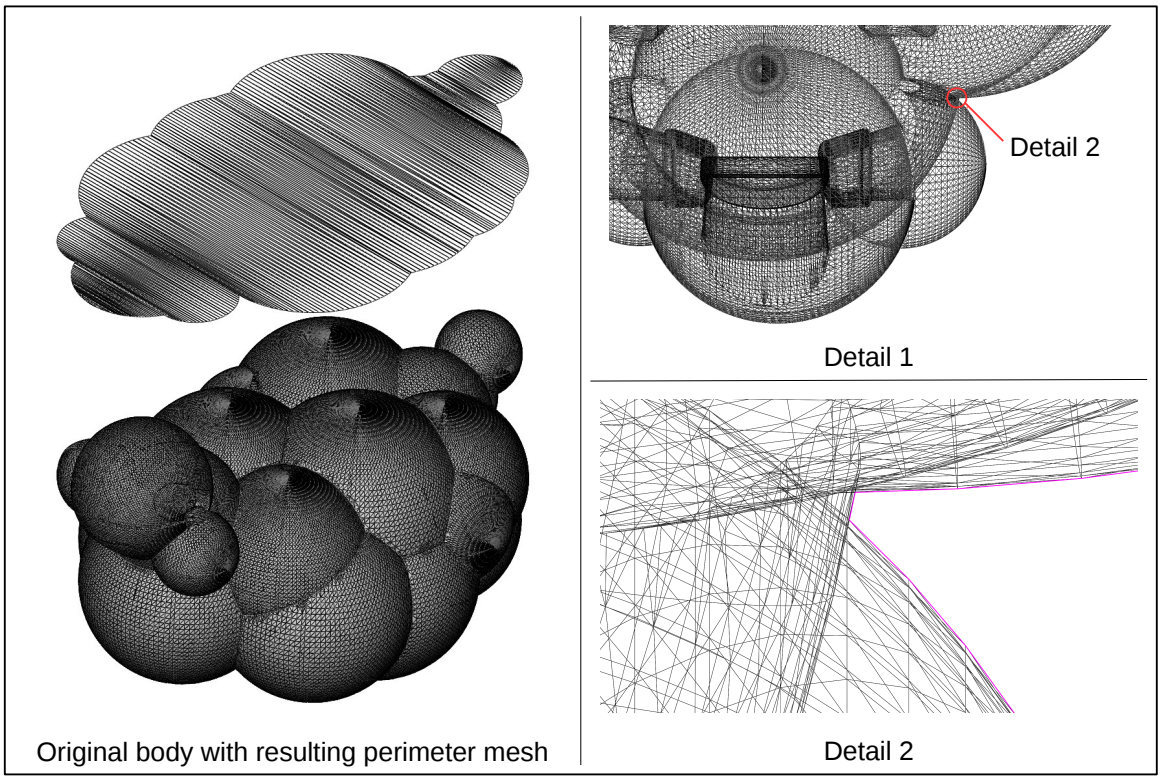


Figure 19: sheepWHITE

24.5%, and 28.4% reduction of faces, edges, and vertexes, respectively. The 429 perimeter edges were identified in 210 seconds. The left side of Fig. 19 shows the original object with the final perimeter mesh above it for comparison. Detail 1 is a view of the head, ear, and body. Detail 2 shows in detail an area of interest. This area shows the intersections of the ear and multiple body segments. This is an example when the algorithm must be capable of handling consecutive intersection. The purple line shows the perimeter edges as identified by the algorithm. The sheepWHITE object showed sensitivity to the delta parameter. The above was executed with a delta of .035.

5.3 Sample object 3: alpha-shape.stl

The alpha shape of sheepWHITE is shown in Fig. 20. As shown on the left side, with an

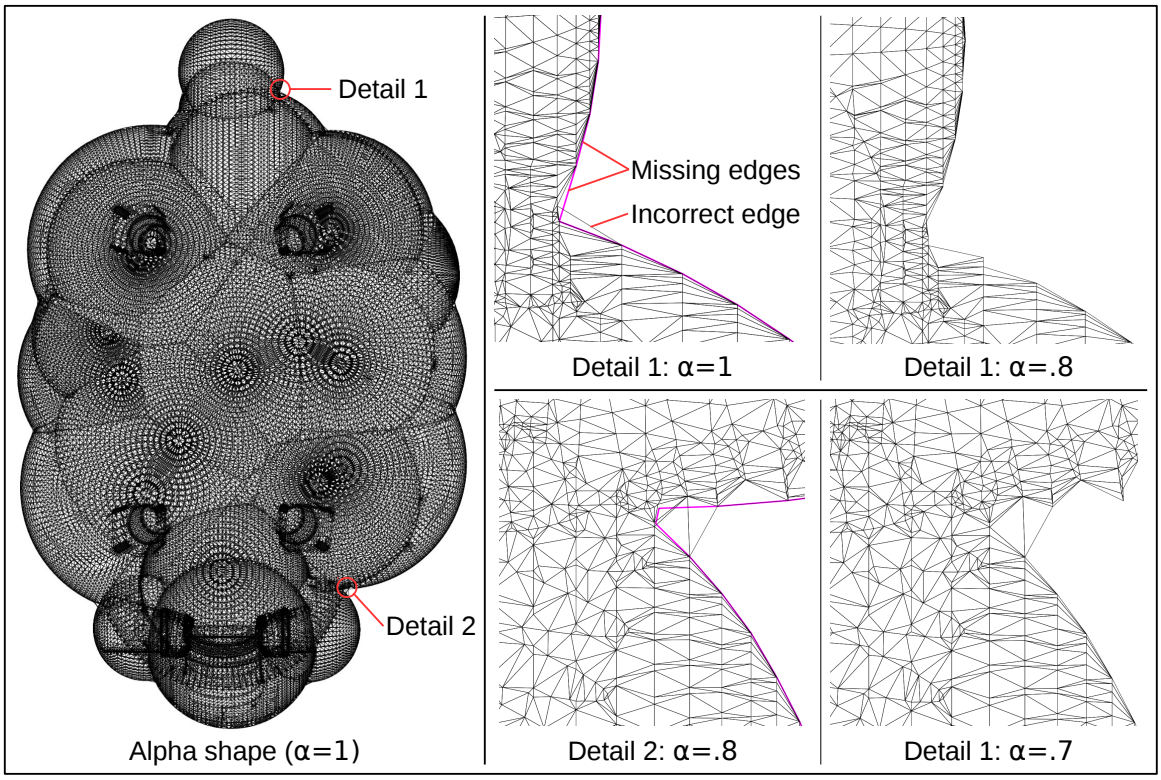


Figure 20: Alpha shape of sheepWHITE

alpha value of 1, the general shape agrees with the actual projection. The right side shows in greater detail the inexactness of the alpha shape. The intersection of the tail and body displayed in detail 1 clearly shows that perimeter edges are starting to be removed (labeled as missing edges) while edges outside of the actual projection (magenta line) remain (labeled as incorrect edge). With an alpha of .8 the edge lying outside of the projection is removed, while the problem of correct perimeter edges disappearing worsens. Detail 2 shows a similar situation. Comparing the difference in behavior of detail 1 and 2 shows the inherent limitation of the alpha method, the severity of concavity relative the edge size and alpha parameter. Had the alpha value been reduced even further to eliminate the other four edges lying outside of the actual projection, the number of missing edges would be in the hundreds. Because the alpha method originates from a convex hull tessellation, connection of the vertexes is not based on any of the original face data and as such the resulting edges of the alpha shape which lie interior to the original convex hull will many times not agree with the actual projection. This is especially true at areas of concavity because addition of vertexes is not possible as it is in the proposed exact method.

5.4 Sample object 4: arduino-micro.stl

A model of the popular Arduino Micro is shown in Fig. 21. This particular model is not a valid STL due to the fact there are many faces missing which results in a non-closed surface. As the right hand of the figure shows, the algorithm handles this situation perfectly. This is to be expected as the algorithm works on a project where the concept of a closed surface is no longer of concern.

Although this particular model took less than 30 seconds it is interesting to note that because the grid spacing for searching for intersecting edges is based of the max edge length, completion may have been faster if the software that created the STL file limited the maximum edge length. This is certainly not true for more organic shapes and may only true for some man-made designs where long edges that lie on the perimeter encounter many intersections. This is seen with the Arduino Micro with its headers partially overhanging the PCB.

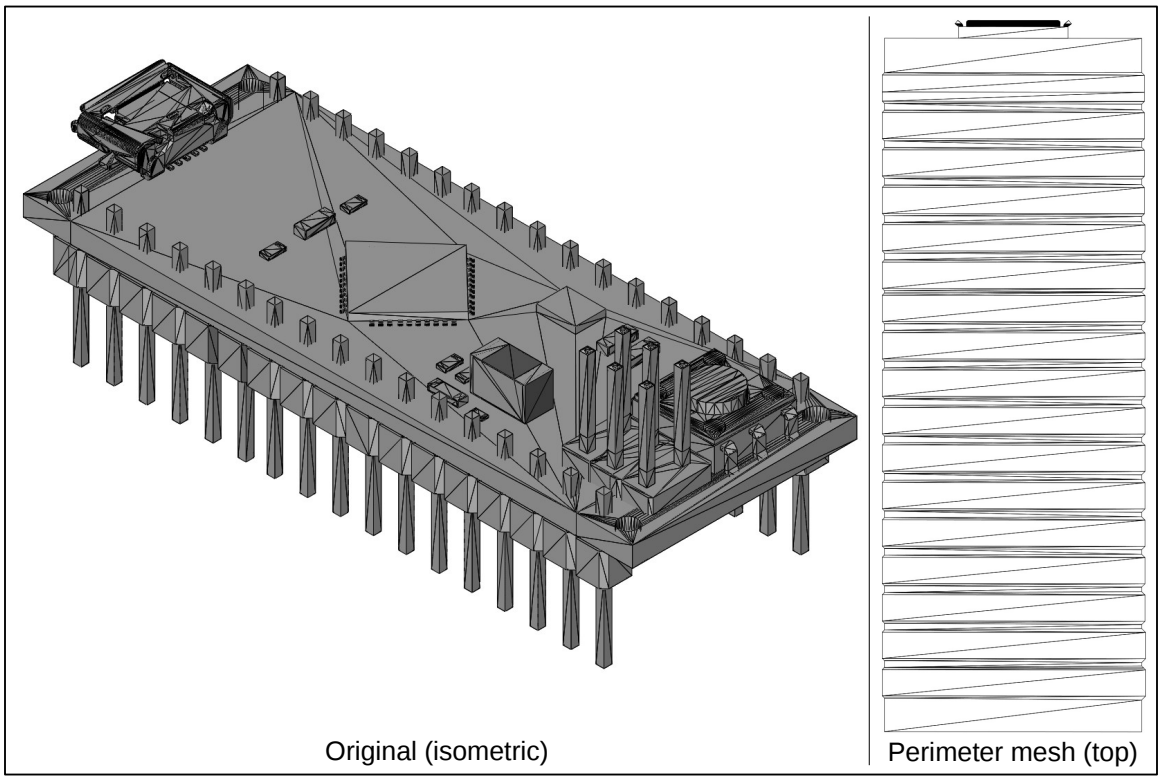


Figure 21: Arduino Micro

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

The algorithm presented in this work has achieved its main goal of determining the exact perimeter of a projecting of a 3D surface as defined by a STL object. Unlike the alpha shape algorithm which uses only vertex information, this algorithm utilizes how vertexes where originally organized into faces to ensure the end result is exact through the creation of new vertexes and edges. The process of identification of perimeter vertexes and edges is the chief concern of the algorithm and should be easily adaptable to other file formats (.obj, .step, etc), and methods of intermediate storage. Unlike the alpha method, this algorithm will work on all shapes and is not an approximation.

The motivation and necessity for this work resulted directly from the desire to embed components into 3D prints. This work is ready for such use, and should provide an adequate foundation for future embedding work.

6.1 Known Issues

Because mesh structures utilize floating point numbers, some caution must be exercised when comparing equivalence. This is particularly true when checking if two lines intersect and the cross product of two lines. For example, the **`does_intersect()`** method dismisses intersection tests that return a value less than 1e-10. This value may need to be adjusted depending on the precision of the machine.

6.2 Feature enhancements

The implementation presented with this work was used to develop the methods of the

algorithm and hindsight gives us insight on useful improvements. This section will help illustrate many possible improvements.

Speed

For integration with other work, such as further development of embedding components, it is advised to decrease the time required by the implementation. This is especially true for larger models which can take drastically longer, for example a model of a statue containing 198,530 triangles required 2 hours and 13 minutes to process. The development of this implementation prioritized speed of development and as such utilized Python which is known for this strength. There are two major areas that should be focused on for increasing efficiency: 1) methods for computing data (calculating intersections, etc.) and 2) methods for storing and accessing data.

The mappings (vertexes to cells, etc.) used the `defaultdict(list)` data type because of it allowed for a consistent method for accessing a variety of different type of data. Initial exploration into using generic python lists for some of the mappings has shown potential decrease in both creation and access of the data. `Numpy` arrays and other custom classes may also be viable solutions.

An Python module known as `Cython` allows calling of C/C++ functions and is popular among the python computational community [26]. Leveraging of `Cython` may greatly increase the overall speed of a Python implementation.

Package dependency

This implementation leveraged pyqtgraph's implementation of OpenGL components for storing the mesh data and greatly assisted in graphical debugging and verification. An implementation can be made much lighter by developing an independent method for storing the mesh data, or by integration into existing packages (such as ReplicatorG or other 3D printing software).

Automation of parameters

At this time delta, the parameter that controls mesh filtering, is the only user specified parameter. Set properly, this parameter can eliminate many redundant vertexes and edges, thus greatly reducing the time associated with edge intersection. Setting this parameter to greater values will eventually lead to substantial loss in accuracy of the representation, although it will continue to decrease the time required.

Investigating the effects of the delta value on a variety of test cases may help develop a more automatic approach to choosing the parameter. Because the mesh filtering technique focuses on merging vertexes, replacing vertex references, and removing duplicate or illegal faces/edges, the resulting form of the shape will not be altered, only the precise location of the vertexes. This means that the acceptable limits of delta should be proportional to the overall size of the original object.

Interactive Debugging

The implementation is only as strong as the collective set of objects that it has been tested against and it is probable that future test cases may reveal new issues. The dubbing

experienced with this work often involved a time consuming process of identifying the exact iteration the algorithm diverged from the perimeter and then scanning through a vast amount of data for the past three to five iterations. This process could be greatly assisted by being able to graphically select the suspected edge and display data on-screen of the 3D viewer.

REFERENCES

[1] K. V. Wong and A. Hernandez, "A Review of Additive Manufacturing", ISRN Mech. Eng., vol. 2012, no. 208760, pp. 1-10, 2012.

[2] J. P. Kruth and M.C. Leu and T. Nakagawa, "Progress in Additive Manufacturing and Rapid Prototyping", Int. Academy for Prod. Eng., vol. 47, no. 2, pp. 525-540, 1998.

[3] Arcam AB. (2012, February). Additive Manufacturing with Electron Beam Melting [Online]. Available: http://www.rm-platform.com/index2.php?option=com_docman&task=doc_view&gid=651&Itemid=5

[4] D. D. Gu and W. Meiners and K. Wissenbach and R. Poprawe, "Laser additive manufacturing of metallic components: materials, processes and mechanisms", Int. Materials Reviews, vol. 57, no. 3, pp. 133-164, 2012.

[5] L. M. Sherman, "Additive Manufacturing Materials for Real World Parts", Plastics Technology, vol. 2104, no. March, pp. 42-47, 2014.

[6] G. Q. Jin and W. D. Li, "Adaptive rapid prototyping/manufacturing for functionally graded material-based biomedical models", Int. J. Advanced Manufacturing Technology, vol. 2013, no. 65, pp. 97-113, 2012.

[7] C. R. Deckard, "Apparatus for production of three-dimensional objects by stereolithography", U.S. Patent US4863538, A. 1 May 2014.

[8] M. Mott and J.R.G. Evans, "Zirconia-alumina functionally graded material made by

ceramic ink jet printing", Materials Sci. and Eng., vol. 1999, no. A271, pp. 344–352, 1999.

[9] V. Ramachandran. (2013, July). 3D-Printed Foot Lets Crippled Duck Walk Again [Online]. Available: http://mashable.com/2013/07/02/3d-printed-duck-foot/

[10] B. Bregar, "Additive Manufacturing Hits $1.7B", Plastics News, vol. 24, no. 14, pp. 7, June, 2012.

[11] 3D Systems. (2014). 30 YEARS OF INNOVATION [Online]. Available: http://www.3dsystems.com/30-years-innovation

[12] D. Ma, F. Lin and C. K. Chua, "Rapid Prototyping Applications in Medicine. Part 2: STL File Generation and Case Studies", Int. J. of Advanced Manufacturing Technology, vol. 2001, no. 18, pp. 118-127, 2001.

[13] K. B. Guo and L. C. Zhang and C. Wang and S. H. Huang, "Boolean operations of STL models based on loop detection", Int. J. of Advanced Manufacturing Technology, vol. 2007, no. 33, pp. 627-633, 2007.

[14] Wikipedia. (2014). STL (file format) [Online]. Available: http://en.wikipedia.org/wiki/STL_(file_format)

[15] Standard Specification for Additive Manufacturing File Format Version 1.1, ASTM Standard 52915, 2013.

[16] The Khronos Group. (2014). OpenGL Overview [Online]. Available: http://www.opengl.org/about/

[17] PyQtGraph. (2014). Scientific Graphics and GUI Library for Python [Online]. Available: http://www.pyqtgraph.org/

[18] Pierre Raybaut. (2012). Python(x,y) Ecosystem Overview [Online]. Available: http://pythonxy.googlecode.com/files/python_2722.png

[19] Python Software Foundation. (2014). Python Execution Model [Online]. Available: https://docs.python.org/2/reference/executionmodel.html

[20] The Stanford 3D Scanning Repository. (2013). Stanford Bunny [Online]. Available: https://graphics.stanford.edu/data/3Dscanrep/

[21] Tran Kai Frank Da. (9, Oct 2014). CGAL 4.5 - 2D Alpha Shapes [Online]. Available: http://doc.cgal.org/latest/Alpha_shapes_2/index.html

[22] M. Green. (2014). Poly2Tri - A 2D constrained Delaunay triangulation library [Online]. Available: https://code.google.com/p/poly2tri/

[23] Python Software Foundation. (2014). The Python Standard Library [Online]. Available:

[24] Numpy Developers. (2014). Numpy Homepage [Online]. Available: http://www.numpy.org/

[25] K. Lepa. (2014). Termcolor [Online]. Available: https://pypi.python.org/pypi/termcolor

[26] Python Software Foundation. (2014). Cython 0.20.2 [Online]. Available: https://pypi.python.org/pypi/Cython/

APPENDIX A: fileIO.py

```
"""
Project:    mesh-ppp
Author:     Alex Hanson | hanson.alex@gmail.com
Date:       2/3/2013

This file handles all file input/output for pyEC3PO.

The following definitions are generally called by higher level
definitions in pyEC3PO such as class initialization of trisurf.py
"""

from __future__ import print_function
import numpy as np
from struct import unpack


def import_stl_file(file_name):
    """
    Determines file extension from file name and calls appropriate
    read definition.
    """
    try:
        _fileExtension = file_name.rsplit('.', 1)[1]
        if _fileExtension.lower() == 'stl':
            return _read_stl(file_name)
        else:
            print("Only .stl files are supported at this time!")

    except IndexError:
        print("Filename does not include recognizable extension:
*.stl")


def _read_stl(file_name):
    """
    Reads binary stl file (ascii not yet supported) and returns data
    as a tuple of 3 tuples.

    Data is returned as tuples to prevent accidental manipulation of
    original data. Data should be converted as necessary by calling
    definition, ex. initialization of pyEC3PO's TriSurf class

    TODO: Add support for ACSII stl.
    """
```

```
with open(file_name, 'rb') as f:
    header = f.read(80)

    if header[:5] == 'solid':
        raise ValueError("%s looks like an ASCII STL file!")

    else:
        _temp = f.read(4)
        n_tri = unpack('i', _temp)[0]

        tri0_dtype = np.dtype([
            ('norm', np.float32, (3,)),
            ('v1', np.float32, (3,)),
            ('v2', np.float32, (3,)),
            ('v3', np.float32, (3,)),
            ('n_atttr', '<i2', (1,))
        ])

        data = np.fromfile(f, dtype=tri0_dtype, count=n_tri)
        n_tri = n_tri

        print('\t' + str(n_tri), "triangles read")
        return data, n_tri
```

APPENDIX B: tools.py

```
"""
Project:    mesh-ppp
Author:     Alex Hanson | hanson.alex@gmail.com
Date:       2/25/2013

This module handles random tools that do not fit into other modules
"""
from __future__ import print_function
from __future__ import division
import os
import numpy as np

class ProgressBar(object):
    """
    The progress bar allows displaying of a tasks progression in the
    terminal by defining the task name, the task length and the number
    of divisions in the progress bar.

    The progress bar is updated by calling update with the current
    step count.
    """
    def __init__(self, task_name, length, divisions=10):
        """
        Initializes the progress bar by printing the task_name and
        containment brackets with divisions number of blanks, and
        backs the cursor divisions number of positions. The length is
        the quantity of steps in the process while the divisions is
        the resolution of the progress bar.

        The division_size is the size of each division of the progress
        bar relative to the length. It is used to determine the
        progress in integer form relative to the resolution of the
        progress bar.

        :param task_name: str
        :param length: int
        :param divisions: int
        """
        self.taskName = str(task_name)
        self.divisions = int(divisions)
        self.length = int(length)
        self.division_size = np.ceil(length/divisions)
        out = self.taskName + '['.ljust(divisions) + ']'
        self.progress = 0
```

```
        print(out, end="")
        print('\b'*divisions, end="")
        os.sys.stdout.flush()

    def update(self, step_count):
        """
        Called to notify the progress bar of the task's progress. The
        step count is compared to the the division_size, if no
        remainder exists, the progress is updated one step (unless
        progress == divisions).

        It is important to update the progress bar each and every
        time, because it updates the progress bar on the condition
        that no remainder exists.

        :param step_count: int
        """
        if step_count % self.division_size == 0:
            self.progress += 1

            if self.progress == self.divisions:
                print('] Done!', end='')
            else:
                print('.', end="")
                os.sys.stdout.flush()
```

APPENDIX C: TriSurf.py

```
"""
Project:    mesh-ppp
Author:     Alex Hanson | hanson.alex@gmail.com
Date:       2/5/2013

This file handles all file input/output

Defines the TriSurf class which extends the capability of gl.MeshData
by providing mesh filtering, projection,
perimeter algorithms, and supporting functions. Also defines some non
TriSurf functions used commonly with numpy
arrays and vectors.
"""

# python core modules
from __future__ import print_function, division
from collections import defaultdict
import time
import os
from itertools import combinations
import numpy as np
import pyqtgraph.opengl as gl
import Poly2Tri
from termcolor import colored
from tools import ProgressBar
from fileIO import import_stl_file

#clear terminal
os.system('clear')

#set np print options
np.set_printoptions(precision=3)

def get_unique_rows(a):
    """
    returns unique rows of an np.array
    :rtype : np.array
    :param a: np.array
    :return: np.array
    """
    col_count = a.shape[1]
    data_type = a.dtype.descr * col_count
    structured_array = a.view(data_type)

    unique_rows = np.unique(structured_array)
```

```
    unique_rows = unique_rows.view(a.dtype).reshape(-1, col_count)
    return unique_rows
def norm(v1):
    """
    returns a vector with length 1
    :param v1: np.array
    :return: np.array
    """
    return v1 / np.linalg.norm(v1)


def dangle(v1, v2):
    """
    Returns the angle between v1 and v2 (degrees)
    :param v1: np.array
    :param v2: np.array
    :return: float
    """
    v1_n = norm(v1)
    v2_n = norm(v2)

    dot = np.dot(v1_n, v2_n)
    if dot > 1:
        dot = 1
    elif dot < -1:
        dot = -1

    angle = np.arccos(dot)

    if np.isnan(angle):
        if (v1_n == v2_n).all():
            angle = 0.0
        else:
            angle = np.pi
    return angle * 180 / np.pi

def min_angle(v1, v2):
    """
    Returns the minimum angle between v1 and v2 (limits angle between
    0 and 90)
    :param v1:
    :param v2:
    """
    angle = dangle(v1, v2)
    if angle > 90:
        angle = abs(angle - 180)

    return angle
class TriSurf(gl.MeshData, object):
```

```
    """
    A class which extends the functionality of pyqtgraph's opengl
    MeshData

    Allows custom functions needed for pyEC3PO. At this time adds
    ability to find exact perimeter of 2D projection.
    """

    def __init__(self,
                 vertexes=None,
                 faces=None,
                 edges=None,
                 vertex_colors=None,
                 face_colors=None,
                 file_name=None):
        """
        attempts to create gl.MeshData attributes based upon inputs.
        If file_name is given then imported content will overwrite
        other inputs
        """

        gl.MeshData.__init__(self, vertexes=vertexes, faces=faces
                             edges=edges, vertexColors=vertex_colors,
                             faceColors=face_colors)

        # set aside variables to be defined later
        self._bounding_box = None
        self._edge_lengths = None

        self.filtering_delta = None
        self._cells_to_vertexes = None
        self._vertexes_to_cells = None

        self._modified_vertexes = None
        self._modified_edges = None
        self._modified_edge_lengths = None
        self._modified_edges_to_cells = None
        self._modified_cells_to_edges = None
        self._modified_vertexes_to_cells = None
        self._cell_groups_to_modified_edges = defaultdict(list)
        self._modified_x_row_size = None
        self._modified_y_col_size = None

        #import from file if file_name given
        if file_name is not None:
            short_file_name = file_name.split('/')[-1]
            text = "Importing " + short_file_name + ':'
            print(colored(text, 'green'))
            self._data, self._n_tri = import_stl_file(file_name)
```

```
            vertexes = np.zeros((self._n_tri, 3, 3))
            for i, subData in enumerate(self._data):
                vertexes[i, :, :] = [subData['v1'], subData['v2'],
                                     subData['v3']]
            self.setVertexes(verts=vertexes, indexed='faces')

        #index vertexes, edges, faces
        self.vertexes()
        self._create_bounding_box()
        print("\tTriSurf successfully created\n")

    @property
    def x_min(self):
        """
        Returns the index and value of the vertexes that has the
        minimum x component. If multiple vertexes contain the minimum
        x component, no order is guaranteed.
        """
        index = int(self.bounding_box[0][0])
        val = self.vertexes()[index]
        return [index, val]

    @property
    def x_max(self):
        """
        Returns the index and value of the vertexes that has the
        maximum x component. If multiple vertexes contain the maximum
        x component, no order is guaranteed.
        """
        index = int(self.bounding_box[0][1])
        val = self.vertexes()[index]
        return [index, val]

    @property
    def y_min(self):
        """
        Returns the index and value of the vertexes that has the
        minimum y component. If multiple vertexes contain the minimum
        y component, no order is guaranteed.
        """
        index = int(self.bounding_box[1][0])
        val = self.vertexes()[index]
        return [index, val]

    @property
    def y_max(self):
        """
        Returns the index and value of the vertexes that has the
```

```
        maximum y component. If multiple vertexes contain the maximum
        y component, no order is guaranteed.
        """
        index = int(self.bounding_box[1][1])
        val = self.vertexes()[index]
        return [index, val]



    @property
    def z_min(self):
        """
        Returns the index and value of the vertexes that has the
        minimum z component. If multiple vertexes contain the minimum
        z component, no order is guaranteed.
        """
        index = int(self.bounding_box[2][0])
        val = self.vertexes()[index]
        return [index, val]

    @property
    def z_max(self):
        """
        Returns the index and value of the vertexes that has the
        maximum z component. If multiple vertexes contain the maximum
        z component, no order is guaranteed.
        """
        index = int(self.bounding_box[2][1])
        val = self.vertexes()[index]
        return [index, val]

    @property
    def bounding_box(self):
        """
        Returns a np.array. Rows are x, y, z. Cols are min, max.
        :return: np.array()
        """
        if self._bounding_box is None:
            self._create_bounding_box()
        else:
            return self._bounding_box

    @property
    def edge_lengths(self):
        """
        Returns a list of each edge magnitude
        :return: list
        """
        if self._edge_lengths is None:
```

```
            self._create_edge_lengths()
            print("\tIndexing edge lengths")
        return self._edge_lengths

    @property
    def max_edge_length(self):
        """
        Returns the maximum edge length as a float
        :return: float
        """
        i_max = self.edge_lengths.argmax()
        return self.edge_lengths[i_max]

    @property
    def vertexes_to_cells(self):
        """
        Returns a defaultdict mapping of each vertex to a cell
        :return: defaultdict(list)
        """
        if self._vertexes_to_cells is None:
            print("\tIndexing vertexes_to_cells")
            self._create_vertexes_to_cells()
        return self._vertexes_to_cells

    @property
    def cells_to_vertexes(self):
        """
        Returns a defaultdict mapping of each cell to a list of
        vertexes with in the cell. used with mesh filtering
        :return: defaultdict(list)
        """
        if self._cells_to_vertexes is None:
            print("\tIndexing cells_to_vertexes")
            self._create_cells_to_vertexes()
        return self._cells_to_vertexes

    @property
    def modified_vertexes(self):
        """
        Returns the 2nd set of vertexes used to determine the exact
        perimeter as an np.array of floats (nv x 3)
        :return: np.array
        """
        if self._modified_vertexes is None:
            print("\tIndexing modified_vertexes")
            self._modified_vertexes = np.copy(self.vertexes())
        return self._modified_vertexes

    @property
```

```
def modified_edges(self):
    """
    Returns the 2nd set of edges used to determine the exact
    perimeter as an np.array of ints (ne x 2)
    :return: np.array
    """
    if self._modified_edges is None:
        print("\tIndexing modified_edges")
        self._modified_edges = np.copy(self.edges())
    return self._modified_edges

@property
def modified_edge_lengths(self):
    """
    Returns the edge lengths of the modified_edges set used to
    determine the exact perimeter as an np.array of floats
    (ne x 1)
    :return: np.array
    """
    if self._modified_edge_lengths is None:
        print("\tIndexing modified_edge_lengths")
        self._create_modified_edge_lengths()
    return self._modified_edge_lengths

@property
def modified_cells_to_edges(self):
    """
    Returns a defaultdict mapping of cell location as key and list
    of edges that start and or stop as value
    :return: defaultdict(list)
    """
    if self._modified_cells_to_edges is None:
        print("\tIndexing modified_cells_to_edges")
        self._create_modified_cells_to_edges()
    return self._modified_cells_to_edges

@property
def modified_edges_to_cells(self):
    """
    Returns a defaultdict mapping of each edge to a list of 2
    cells, one for each vertex
    :return: defaultdict(list)
    """
    if self._modified_edges_to_cells is None:
        print("\tIndexing modified_edges_to_cells")
        self._create_modified_edges_to_cells()
    return self._modified_edges_to_cells
```

```
@property
def modified_vertexes_to_cells(self):
    """
    Returns a defaultdict mapping of each vertex to a cell
    :return: defaultdict(list)
    """
    if self._modified_vertexes_to_cells is None:
        print("\tIndexing modified_vertexes_to_cells")
        self._create_modified_vertexes_to_cells()
    return self._modified_vertexes_to_cells

def _create_bounding_box(self):
    """
    Stores min / max x, y, z vertexes as np.array (2 x 3). Rows
    are for x, y, z and cols are for min, max
    :return: None
    """
    vertexes = self.vertexes()
    vertex_count = vertexes.shape[1]

    minimum_indexes = [col.argmin() for col in vertexes.T]
    maximum_indexes = [col.argmax() for col in vertexes.T]

    bbox = np.zeros([vertex_count, 2], dtype=int)
    bbox[:, 0] = minimum_indexes
    bbox[:, 1] = maximum_indexes

    self._bounding_box = bbox

def _create_edge_lengths(self):
    """
    Stores the edge lengths of the original edges set as a
    np.array of floats (ne x 1) to self._modified_edge_lengths
    :return: None
    """
    vertexes = self.vertexes()
    edges = self.edges()

    edge_vectors = vertexes[edges[:, 1]] - vertexes[edges[:, 0]]

    #with numpy 1.9 can be speed up with
    #np.linalg.norm(edge_vectors, axis=-1)
    self._edge_lengths = np.sum(np.abs(edge_vectors)**2,
                                axis=-1)**(1./2)


def _create_vertexes_to_cells(self):
```

```
        """
        Stores a mapping of each vertex to delta cell to assist in
        mesh filtering
        :return: None
        """
        vertexes = self.vertexes()
        delta = self.filtering_delta

        # pull out x/y_min, x/y_max
        x_min = self.x_min[1][0]
        y_min = self.y_min[1][1]
        y_max = self.y_max[1][1]

        # determine length x/y scope
        x_range = y_max - x_min
        y_range = y_max - y_min

        # number of x/y columns
        x_row_count = int(np.floor(x_range / (delta * 1.01)))
        y_col_count = int(np.floor(y_range / (delta * 1.01)))

        #width height of row / col
        x_col_size = x_range / x_row_count
        y_row_size = y_range / y_col_count

        #save to default dict
        #TIME: .52 sec
        vertexes_to_cells = defaultdict(list)
        for i1, vertex in enumerate(vertexes):
            x_row = int(np.floor((vertex[0] - x_min)/ x_col_size + 1))
            y_col = int(np.floor((vertex[1] - y_min)/ y_row_size + 1))
            vertexes_to_cells[i1].append((x_row, y_col))

        self._vertexes_to_cells = vertexes_to_cells

    def _create_cells_to_vertexes(self):
        """
        Stores a mapping of each mini cell and the vertexes contained
        in said cell
        :return: None
        """

        #TIME: .81
        cells_to_vertexes = defaultdict(list)
        for vertex, cells in self.vertexes_to_cells.items():
            cells_to_vertexes[cells[0]].append(vertex)

        self._cells_to_vertexes = cells_to_vertexes
```

```
def _create_modified_edge_lengths(self):
    """
    Stores the edge lengths of the modified_edges set as a
    np.array of floats (ne x 1) to self._modified_edge_lengths
    :return: None
    """
    vertexes = self.modified_vertexes
    edges = self.modified_edges

    edge_vectors = vertexes[edges[:, 1]] - vertexes[edges[:, 0]]

    #with numpy 1.9 can be speed up with
    #np.linalg.norm(edge_vectors, #axis=-1)
    self._modified_edge_lengths = np.sum(np.abs(edge_vectors)**2,
                                         axis=-1)**(1./2)

def _create_modified_vertexes_to_cells(self):
    """
    Stores a mapping of each vertex to a cell as a defaultdict as
    self._modified_vertexes_to_cells
    :return: None
    """
    vertexes = self.modified_vertexes

    #calculate x/y cell of each vertex
    #TIME: 1.64
    for i1, vertex in enumerate(vertexes):
        self.add_modified_vertex_to_cells(i1)

def _create_modified_edges_to_cells(self):
    """
    Stores a mapping of each edge to two cells as a defaultdict
    :return: None
    """
    edges = self.modified_edges

    #TIME: 3.88
    edges_to_cells = defaultdict(list)
    for i1, edge in enumerate(edges):
        cell_1 = self.modified_vertexes_to_cells[edge[0]][0]
        cell_2 = self.modified_vertexes_to_cells[edge[1]][0]
        edges_to_cells[i1].append(cell_1)
        edges_to_cells[i1].append(cell_2)

    self._modified_edges_to_cells = edges_to_cells
```

```
def _create_modified_cells_to_edges(self):
    """
    Stores a mapping of each each cell and each edge that starts
    and or stops in given cell as a defaultdict
    :return: None
    """
    cells_to_edges = defaultdict(list)
    for edge, cells in self.modified_edges_to_cells.items():
        cells_to_edges[cells[0]].append(edge)
        cells_to_edges[cells[1]].append(edge)

    self._modified_cells_to_edges = cells_to_edges

def add_color(self):
    """
    Adds color data to gl.MeshData simply based upon the normal of
    each face
    :return:
    """
    colors = np.ones([self._n_tri, 4], dtype=float)
    colors[:, 0:3] = self._data['norm']
    self.setFaceColors(colors)

def add_modified_vertex_to_cells(self, vertex_index):
    """
    Handles adding a vertex-cell mapping. Checks first if this
    function has ever been called before by checking
    self._modified_x_row_size
    :param vertex_index: int
    """
    vertexes = self.modified_vertexes
    edges = self.modified_edges
    # pull out x/y_min, x/y_max
    x_min = self.x_min[1][0]
    y_min = self.y_min[1][1]
    y_max = self.y_max[1][1]

    #check if row/col sizes have been calculated
    if self._modified_x_row_size is None:
        #determine max x components and y components to help
        #determine col size
        x_components = np.zeros(edges.shape[0], dtype=float)
        y_components = np.zeros(edges.shape[0], dtype=float)

        for i1, edge in enumerate(edges):
            v_x = vertexes[edge[1]][0] - vertexes[edge[0]][0]
            v_y = vertexes[edge[1]][1] - vertexes[edge[0]][1]

            x_components[i1] = abs(v_x)
```

```
            y_components[i1] = abs(v_y)

        max_x_comp = x_components[x_components.argmax()]
        max_y_comp = y_components[y_components.argmax()]

        # determine length x/y scope
        x_range = y_max - x_min
        y_range = y_max - y_min

        #number of x/y columns
        x_row_count = int(np.floor(x_range / (max_x_comp * 1.01)))
        y_row_count = int(np.floor(y_range / (max_y_comp * 1.01)))

        #determine actual x_col_size, y_row_size correcting for
        #floor()
        self._modified_x_row_size = x_range / x_row_count
        self._modified_y_col_size = y_range / y_row_count
        self._modified_vertexes_to_cells = defaultdict(list)

    x_row = int(np.floor((vertexes[vertex_index][0] - x_min) /
                         self._modified_x_row_size + 1))

    y_col = int(np.floor((vertexes[vertex_index][0] - y_min) /
                         self._modified_y_col_size + 1))

    self._modified_vertexes_to_cells[vertex_index].append((x_row,
                                                           y_col))

def add_modified_vertex(self, xf, yf, zf):
    """
    Adds a new vertex to the modified_vertexes array. Manages
    other actions, such as addition of vertex to mappings.
    :param xf: float
    :param yf: float
    :param zf: float
    """
    #add vertex
    self._modified_vertexes = np.append(self._modified_vertexes,
                                        [(xf, yf, zf)], axis=0)

    #add vertex to vertex to cell mapping
    vertex_index =  self._modified_vertexes.shape[0]-1
    self.add_modified_vertex_to_cells(vertex_index)

def add_modified_edge(self, v1i, v2i):
    """
    Adds a new edge to the modified_vertexes array. Manages other
    actions, such as addition of edge into mappings.
```

```
        :param v1i: int
        :param v2i: int
        """
        vertexes = self.modified_vertexes

        #add edge
        self._modified_edges = np.append(self.modified_edges,
                                         [[v1i, v2i]],
                                         axis=0).astype('i')

        #add length
        length = np.linalg.norm(vertexes[self.modified_edges[-1][1]] -
                                vertexes[self.modified_edges[-1][0]])

        self._modified_edge_lengths =
            np.append(self.modified_edge_lengths, [length], axis=0)

        #map edge to cell
        edge_index = self._modified_edges.shape[0]-1
        edge = self._modified_edges[-1]
        cell_1 = self.modified_vertexes_to_cells[edge[0]][0]
        cell_2 = self.modified_vertexes_to_cells[edge[1]][0]
        self._modified_edges_to_cells[edge_index].append(cell_1)
        self._modified_edges_to_cells[edge_index].append(cell_2)

        #map cell to edge
        self._modified_cells_to_edges[cell_1].append(edge_index)
        self._modified_cells_to_edges[cell_2].append(edge_index)

    def get_vertexes_from_faces(self, faces):
        """
        Returns mesh as np.array of floats (nf x 3 x 3)
        :param faces: np.array
        :return: np.array
        """
        vertexes = self.vertexes()

        mesh = []
        for face in faces:
            v1 = vertexes[face[0]]
            v2 = vertexes[face[1]]
            v3 = vertexes[face[2]]
            face_vertexes = np.array([v1, v2, v3])

            mesh.extend([face_vertexes])

        return np.array(mesh)

    def get_connected_modified_edges_data(self,
```

```
                                    edge_indexes,
                                    pivot_edge_index=None,
                                    starting_vertex_index=None):
    """
    Returns a matrix of data for a supplied list of edge indexes.
    The pivot_edge_index is the edge index of the edge that the
    other edges are being compared to i.e. angle, and crossMag
    starting_vertex_index is used for the special case of
    selecting the first edge of the perimeter.
    :param edge_indexes: np.array
    :param pivot_edge_index: int
    :param starting_vertex_index: int
    :return: edge_info_data_type
    """
    vertexes = self.modified_vertexes
    edges = self.modified_edges

    edge_info_data_type = [('edgeIndex', 'u8'),
                           ('v1i', 'u8'),
                           ('v2i', 'u8'),
                           ('v1f', '3float64'),
                           ('v2f', '3float64'),
                           ('angle', 'float64'),
                           ('crossMag', 'float64'),
                           ('length', 'float64')]

    # determine if mode and set v1
    is_connected = False
    if pivot_edge_index is not None:
        is_connected = True
        # remove pivot Edge if is_connected
        edge_index_to_remove = np.where(edge_indexes ==
                                        pivot_edge_index)[0]

        edge_indexes = np.delete(edge_indexes,
                                 edge_index_to_remove)

        # find v1 - check v1 of first edge
        if edges[edge_indexes][0][0] in edges[pivot_edge_index]:
            v1 = edges[edge_indexes][0][0]
        else:
            v1 = edges[edge_indexes][0][1]
    else:
        v1 = starting_vertex_index

    # save edge index
    edge_data = np.zeros(len(edge_indexes),
                         dtype=edge_info_data_type)
```

```
edge_data['edgeIndex'] = edge_indexes

# save vertexes indexes, sorted if is_connected
if is_connected or starting_vertex_index is not None:
    edge_data['v1i'] = v1
    v2_indexes = np.zeros(len(edge_indexes))
    for i1, edge_index in enumerate(edge_indexes):
        edge = edges[edge_index]
        if edge[0] == v1:
            v2_indexes[i1] = edge[1]
        else:
            v2_indexes[i1] = edge[0]

    edge_data['v2i'] = v2_indexes

else:
    edge_data['v1i'] = edges[edge_indexes][:, 0]
    edge_data['v2i'] = edges[edge_indexes][:, 1]

#save vertex floats
edge_data['v1f'] = [vertexes[i] for i in edge_data['v1i']]
edge_data['v2f'] = [vertexes[i] for i in edge_data['v2i']]

#save edge length
edge_data['length'] = [self.modified_edge_lengths[i] for i in
                       edge_data['edgeIndex']]

#save angle with pivot edge (or y unit vector)
v21 = edge_data['v2f'] - edge_data['v1f']
if pivot_edge_index is None:
    #y vector
    v_pivot = np.array([0, 1, 0])
    edge_data['angle'] = [dangle(vEdge, v_pivot) for vEdge in
                          v21]
else:
    v_p1 = vertexes[v1]
    if edges[pivot_edge_index][0] == v1:
        v_p0 = vertexes[edges[pivot_edge_index][1]]
    else:
        v_p0 = vertexes[edges[pivot_edge_index][0]]
    v_pivot = v_p1 - v_p0
    edge_data['angle'] = [dangle(v_pivot, vEdge) for vEdge in
                          v21]

cross = np.zeros(len(edge_indexes))
#cross with pivot edge
for i1, vEdge in enumerate(v21):
    _cross = np.cross(v_pivot, vEdge)[2]
```

```
            if abs(_cross) < 1e-14:
                _cross = 0
            cross[i1] = _cross

        edge_data['crossMag'] = cross
        return edge_data

    def get_intersecting_modified_edges(self,
                                        edge_index,
                                        starting_vertex_index):
        """
        Returns all edge indexes that intersect edge_index of the
        modified_edges set.
        :param edge_index:
        :param starting_vertex_index:
        :return: np.array
        """
        vertexes = self.modified_vertexes
        edges = self.modified_edges
        edges_to_cells = self.modified_edges_to_cells
        cells_to_edges = self.modified_cells_to_edges
        pivot_edge = edges[edge_index]
        v0i = starting_vertex_index

        # find all cells to search
        cell_1, cell_2 = edges_to_cells[edge_index]

        cell_1_neighbors = set()
        for i1 in np.arange(cell_1[0] - 1, cell_1[0] + 2):
            for i2 in np.arange(cell_1[1] - 1, cell_1[1] + 2):
                cell_1_neighbors.add((i1, i2))

        cell_2_neighbors = set()
        for i1 in np.arange(cell_2[0] - 1, cell_2[0] + 2):
            for i2 in np.arange(cell_2[1] - 1, cell_2[1] + 2):
                cell_2_neighbors.add((i1, i2))

        cells_to_search =
            tuple(sorted(cell_1_neighbors.union(cell_2_neighbors)))

        # make a list of all edges in cells_to_search
        if cells_to_search not in self._cell_groups_to_modified_edges:
            edge_set = set()
            for cell in cells_to_search:
                edge_set.update(cells_to_edges[cell])

            self._cell_groups_to_modified_edges[cells_to_search] =
                list(edge_set)
```

```
x1, y1 = vertexes[pivot_edge[0]][0:2]
x2, y2 = vertexes[pivot_edge[1]][0:2]
intersecting_edges = []
intersecting_angles = []
intersecting_points = []
intersection_distances = []

# find intersections and save relevant information
for _edge_index in
    self._cell_groups_to_modified_edges[cells_to_search]:

    edge = edges[_edge_index]
    x3, y3 = vertexes[edge[0]][0:2]
    x4, y4 = vertexes[edge[1]][0:2]

    #ensure 2 edges don't start or stop at the same point
    p1_truth = (x1, y1) == (x3, y3) or (x1, y1) == (x4, y4)
    p2_truth = (x2, y2) == (x3, y3) or (x2, y2) == (x4, y4)

    if not (p1_truth or p2_truth):
        #ensure the two lines are not parallel
        if not ((x3 - x4) * (y1 - y2) == (y3 - y4) * (x1 -
                x2)):

            intersection = self.does_intersect(pivot_edge,
                                               edge)

            #calculate angle of intersecting,
            if intersection:
                v1f = vertexes[pivot_edge[1]] - \
                              vertexes[pivot_edge[0]]

                v2f = vertexes[edge[1]] - vertexes[edge[0]]
                angle = dangle(v1f, v2f)
                intersecting_angles.extend([angle])
                intersecting_edges.extend([_edge_index])

                x_top = (x1*y2 - y1*x2) * (x3-x4) - \
                        (x1-x2) * (x3*y4 - y3*x4)

                x_bottom = (x1-x2) * (y3-y4) - (y1-y2) * \
                           (x3-x4)

                y_top = (x1*y2 - y1*x2) * (y3-y4) - \
                        (y1-y2) * (x3*y4 - y3*x4)

                y_bottom = (x1-x2) * (y3-y4) - (y1-y2) *  \
                           (x3-x4)
```

```
                    x_intercept = x_top / x_bottom
                    y_intercept = y_top / y_bottom

                    intersecting_points.append(
                        np.array([x_intercept, y_intercept]))

                    _v = np.array([x_intercept, y_intercept]) - \
                                  vertexes[v0i][0:2]

                    intersection_distance = np.linalg.norm(_v)
                    intersection_distances.extend(
                        [intersection_distance])

        out = np.array(intersecting_edges), \
            np.array(intersecting_angles), \
            np.array(intersecting_points), \
            np.array(intersection_distances)

        return out

    def get_perimeter_edges_and_vertexes(self):
        """
        Determines the perimeter edges and vertexes of a projected
        (filtered) mesh
        :return: list
        """
        t_1 = time.clock()
        print(colored("Finding perimeter:", 'green'))
        perimeter_vertex_indexes = list()
        perimeter_edges = list()

        self._create_modified_edge_lengths()
        self._create_modified_edges_to_cells()
        self._create_modified_vertexes_to_cells()
        self._create_modified_cells_to_edges()

        # select stating vertex
        v0i = [self.x_max[0]]
        perimeter_vertex_indexes.extend(v0i)

        #Connected Edges to starting vertex
        connected_edges = np.where(self.modified_edges ==
                                   perimeter_vertex_indexes[-1])[0]

        #determine starting edges, save relevant data for later use
        connected_edge_data = \
            self.get_connected_modified_edges_data(
                connected_edges, starting_vertex_index=v0i)
```

```
#determine min angle and edges with min angle:
_index = connected_edge_data['angle'].argmin()
_min_angle = connected_edge_data[_index]['angle']
_index = np.where(connected_edge_data['angle'] == _min_angle)
edges_with_min_angle = connected_edge_data[_index]

#save starting edge
edge_index = edges_with_min_angle['length'].argmin()
perimeter_edges.append(
    edges_with_min_angle[edge_index]['edgeIndex'])

#save 2nd vertex
if edges_with_min_angle['v2i'][edge_index] == \
    perimeter_vertex_indexes[-1]:

    perimeter_vertex_indexes.append(
        int(edges_with_min_angle['v1i'][edge_index]))

else:
    perimeter_vertex_indexes.append(
        int(edges_with_min_angle['v2i'][edge_index]))

#loop through perimeter
for i1 in np.arange(0, 500):

    #Connected Edges
    connected_edges = \
        np.where(self.modified_edges == \
            perimeter_vertex_indexes[-1])[0]

    #retrieve edge data
    connected_edge_data = \
        self.get_connected_modified_edges_data(
            connected_edges, perimeter_edges[-1])

    #find which cases we have
    neg_case = np.where(
                    connected_edge_data['crossMag'] < 0)[0]

    zero_case = np.where(connected_edge_data['angle'] == 0)[0]

    pos_case = np.where(
                    connected_edge_data['crossMag'] > 0)[0]

    #determine best edges / vertex
    #for counter-clockwise method, favor neg cross magnitude
    if len(neg_case) > 0:
```

```
        case_edges = connected_edge_data[neg_case]
        best_index = case_edges['angle'].argmax()

    elif len(zero_case) > 0:
        case_edges = connected_edge_data[zero_case]
        best_index = case_edges['length'].argmin()

    elif len(pos_case) > 0:
        case_edges = connected_edge_data[pos_case]
        best_index = case_edges['angle'].argmin()

    next_edge = case_edges['edgeIndex'][best_index]
    next_vertex = case_edges['v2i'][best_index]

    #check if we're done
    if next_vertex == perimeter_vertex_indexes[0]:
        perimeter_vertex_indexes.append(next_vertex)
        perimeter_edges.append(next_edge)
        steps = len(str(len(perimeter_edges)-1))
        print("\b" * steps + \
              colored(str(len(perimeter_edges)),
              "red"), "in ", time.clock() - t_1, " sec\n")

        break

    #save best edge
    perimeter_vertex_indexes.append(next_vertex)
    perimeter_edges.append(next_edge)
    if len(perimeter_edges) == 2:
        print("\tEdges Detected:", '2', end='')
        os.sys.stdout.flush()

    elif len(perimeter_edges) > 2:
        steps = len(str(len(perimeter_edges)-1))
        print("\b" * steps + str(len(perimeter_edges)),
              end="")

        os.sys.stdout.flush()

    count = 0
    flag_intersects = True
    while flag_intersects:
        #check for edges that intersect the new edge was i1+1
        out = self.get_intersecting_modified_edges(
                  perimeter_edges[-1],
                  perimeter_vertex_indexes[-2])

        intersecting_edges, intersecting_angles, \
```

```
intersecting_points, intersection_distances = out

#pick best intersecting edge
if len(intersecting_edges) > 0:

    #choose intersecting edge that results in min
    #intersection distance

    _index = intersection_distances.argmin()
    intersecting_edge_index =
        intersecting_edges[_index]

    if count > 10:
        print(colored(i1, "red"))
        break

    #add intersection point to modified_vertexes
    intersection_point = intersecting_points[_index]
    self.add_modified_vertex(intersection_point[0],
                             intersection_point[1],
                             self.z_max[1][2])
    vertexes = self.modified_vertexes

    #add edge from new vertex to
    #perimeter_vertex_indexes[-2]
    self.add_modified_edge(
        perimeter_vertex_indexes[-2],
        vertexes.shape[0]-1)

    #determine which vertex is to outside of
    #intersection
    e_1i = \
        [self.modified_edges[
            intersecting_edge_index][0],
            perimeter_vertex_indexes[-2]]

    e_2i = \
        [self.modified_edges[
            intersecting_edge_index][1],
            perimeter_vertex_indexes[-2]]

    e_1f = vertexes[e_1i[0]] - vertexes[e_1i[1]]
    e_2f = vertexes[e_2i[0]] - vertexes[e_2i[1]]

    e_pivot = vertexes[self.modified_edges[-1][0]] - \
              vertexes[self.modified_edges[-1][1]]

    v1_cross = np.cross(e_1f, e_pivot)
```

```
                    v2_cross = np.cross(e_2f, e_pivot)

                    _iei =  intersecting_edge_index
                    if v1_cross[2] < 0:
                        new_edge = [self.modified_edges[_iei][0],
                                    vertexes.shape[0]-1]

                    elif v2_cross[2] < 0:
                        new_edge = [self.modified_edges[_iei][1],
                                    vertexes.shape[0]-1]

                    elif v1_cross[2] == 0:
                        new_edge = [self.modified_edges[_iei][0],
                                    vertexes.shape[0]-1]

                    elif v2_cross[2] == 0:
                        new_edge = [self.modified_edges[_iei][1],
                                    vertexes.shape[0]-1]

                    #add second edge
                    self.add_modified_edge(new_edge[0], new_edge[1])
                    #remove original chosen vertex and edge
                    perimeter_vertex_indexes.pop()
                    perimeter_edges.pop()

                    #add vertex and edges to perimeter set
                    perimeter_vertex_indexes.append(new_edge[1])
                    perimeter_vertex_indexes.append(new_edge[0])
                    perimeter_edges.append(
                        self.modified_edges.shape[0]-2)

                    perimeter_edges.append(
                        self.modified_edges.shape[0]-1)

                    count += 1
                else:
                    flag_intersects = False
        print('\n')
        return [perimeter_vertex_indexes, perimeter_edges]

    def return_projection(self):
        """
        create a simple 2D projection by setting the z components to
        zero
        :return: TriSurf
        """

        print(colored("Creating projected 2D mesh:", 'green'))
```

```
    vertexes = self.vertexes(indexed='faces') * 1
    vertexes[:, :, 2] = self.z_max[1][2]

    return TriSurf(vertexes=vertexes)


def return_filtered_mesh(self, delta=.01):
    """
    Creates a filtered mesh based upon filter parameter delta
    :param delta: float
    :return: TriSurf
    """
    print(colored("Creating filtered mesh:", 'green'))
    self.filtering_delta = delta
    vertexes = self.vertexes()
    faces = self.faces()

    #loop through all cells, find neighboring cells to search,
    #find
    #similar vertexes

    duplicate_to_unique_vertexes = defaultdict(list)
    progress_bar = ProgressBar("\tFiltering Vertexes ",
                               len(self.cells_to_vertexes),
                               divisions=30)

    for i1, cell in enumerate(self.cells_to_vertexes):
        progress_bar.update(i1)
        # determine cells to search

        neighboring_cells = set()

        for i2 in np.arange(cell[0] - 1, cell[0] + 2):
            for i3 in np.arange(cell[1] - 1, cell[1] + 2):
                neighboring_cells.add((i2, i3))

        cells_to_search = tuple(sorted(neighboring_cells))

        #make list of all vertexes to search
        vertexes_to_search = set()
        for _cell in cells_to_search:
            if _cell in self.cells_to_vertexes:
                cell_vertexes = self.cells_to_vertexes[_cell]
                for vertex in cell_vertexes:
                    vertexes_to_search.add(vertex)

        #for each vertexes, search all other vertexes distance
        vertexes_to_search = list(sorted(vertexes_to_search))
```

```
    if len(vertexes_to_search) > 1:
        for v1i, v2i in combinations(vertexes_to_search, 2):
            v1f = vertexes[v1i]
            v2f = vertexes[v2i]

            #much faster than np.linalg.norm(v1f-v2f)
            if abs(v1f[0] - v2f[0]) < delta and \
                abs(v1f[1] - v2f[1]) < delta:
                #map v2f to v1f
                #if v1i is a 'bad vertexes', set equal to its
                #good vertexes

                if v1i in duplicate_to_unique_vertexes:
                    v1i = duplicate_to_unique_vertexes[v1i][0]

                if v1i not in
                    duplicate_to_unique_vertexes[v2i]:

                    temp =  duplicate_to_unique_vertexes
                    temp[v2i].append(v1i)
                    duplicate_to_unique_vertexes = temp

# replace face vertex indexes
print(" -> ",
      colored(len(duplicate_to_unique_vertexes), "red"),
      'removed')

print("\tRepairing faces, ", end="")
flattened_faces = faces.flatten()
for i1, face in enumerate(flattened_faces):
    if face in duplicate_to_unique_vertexes:
        flattened_faces[i1] = \
            duplicate_to_unique_vertexes[face][0]

corrected_faces = flattened_faces.reshape([len(faces), 3])

#remove duplicate rows
unique_faces = get_unique_rows(corrected_faces)

#remove faces that don't have 3 unique vertexes
count = 0
valid_faces = []
for face in unique_faces:
    truth_12 = face[0] == face[1]
    truth_23 = face[1] == face[2]
    truth_31 = face[2] == face[0]

    if truth_12 or truth_23 or truth_31:
```

```
                count += 1
            else:
                valid_faces.extend([face])

        valid_faces = np.array(valid_faces)
        print(colored((faces.shape[0] - valid_faces.shape[0]), "red"),
              "removed")

        print("\tRebuilding mesh data")
        #Convert vertexes (NV,3) and faces (NF,3) to vertexes(NF,3,3)
        final_vertexes = self.get_vertexes_from_faces(valid_faces)

        return TriSurf(vertexes=final_vertexes)

    def return_perimeter_mesh(self):
        """
        Creates a mesh from the perimeter_vertexes returned from
        self.get_perimeter_edges_and_vertexes(). Tessellation handled
        by Poly2Tri.Triangulator(). See Poly2Tri for credits.
        :return:
        """

        [perimeter_vertexes, _] = \
            self.get_perimeter_edges_and_vertexes()

        print(colored("Creating perimeter mesh:", 'green'))

        vertexes = self.modified_vertexes
        perimeter_lines = vertexes[perimeter_vertexes]

        #tesselate perimeter mesh
        tessellation = Poly2Tri.Triangulator(perimeter_lines)
        new_vertexes = np.array(tessellation.triangles())
        new_vertexes_3d = np.ones((new_vertexes.shape[0], 3, 3),
                                  dtype=float)*(self.z_max[1][2]*1.5)

        new_vertexes_3d[:, :, 0:2] = new_vertexes

        return TriSurf(vertexes=new_vertexes_3d)

    def does_intersect(self, edge1, edge2):
        """
        Returns true if lines intersect. Comparing if p3/p4 is
        _is_counter_clockwise of p1,p2 (one must be, one must not be)
        and if p4 is _is_counter_clockwise of p1,p3 and p2,p3 (one
        must be, one must not be)
        """

        vertexes = self.modified_vertexes
```

```
        a = vertexes[edge1[0]]
        b = vertexes[edge1[1]]
        c = vertexes[edge2[0]]
        d = vertexes[edge2[1]]

        truth_1 = self._is_counter_clockwise(a, c, d) != \
            self._is_counter_clockwise(b, c, d)

        truth_2 = self._is_counter_clockwise(a, b, c) != \
            self._is_counter_clockwise(a, b, d)

        if truth_1 and truth_2:
            _temp = [self._intersection_helper(a, c, d),
                     self._intersection_helper(b, c, d),
                     self._intersection_helper(a, b, c),
                     self._intersection_helper(a, b, d)]

            for val in _temp:
                if abs(val) < 1e-14:
                    return False

        return truth_1 and truth_2

    @staticmethod
    def _is_counter_clockwise(a, b, c):
        """
        determines point c relative location to point a and b
        :param a: np.array
        :param b: np.array
        :param c: np.array
        :return: boolean
        """
        return (c[1]-a[1]) * (b[0]-a[0]) > (b[1]-a[1]) * (c[0]-a[0])

    @staticmethod
    def _intersection_helper(a, b, c):
        """
        Used for debugging
        :param a: np.array
        :param b: np.array
        :param c: np.array
        :return: float
        """

        return (c[1]-a[1]) * (b[0]-a[0]) - (b[1]-a[1]) * (c[0]-a[0])

    @staticmethod
    def intersection(vertex_1, vertex_2, vertex_3, vertex_4):
```

```
    """
    returns the intersection point of two vectors, (vertex_1,
    vertex_2) and (vertex_3, vertex_4)
    :param vertex_1:
    :param vertex_2:
    :param vertex_3:
    :param vertex_4:
    :return:
    """
    # dump e1
    x1 = vertex_1[0]
    y1 = vertex_1[1]
    x2 = vertex_2[0]
    y2 = vertex_2[1]

    # dump e2
    x3 = vertex_3[0]
    y3 = vertex_3[1]
    x4 = vertex_4[0]
    y4 = vertex_4[1]

    # calc m1
    m1_top = y2 - y1
    m1_bottom = x2 - x1
    if m1_bottom == 0:
        m1 = 'inf'
    else:
        m1 = m1_top / m1_bottom

    #calc m2
    m2_top = y4 - y3
    m2_bottom = x4 - x3
    if m2_bottom == 0:
        m2 = 'inf'
    else:
        m2 = m2_top / m2_bottom

    #calc b1
    if m1 == 'inf':
        #Vertical line, no b
        b1 = None
    elif m1 == 0:
        #Horizontal line, b = x
        b1 = y1
    else:
        b1 = y1 - (m1 * x1)

    #calc b2
```

```
        if m2 == 'inf':
            #Vertical line, no b
            b2 = None
        elif m2 == 0:
            #Horizontal line, b = x
            b2 = y3
        else:
            b2 = y3 - (m2 * x3)

        #calc intercept
        if m1 == 'inf' and m2 == 0:
            #vertical and horizontal
            x_intercept = x1
            y_intercept = y3
        elif m1 == 0 and m2 == 'inf':
            #horizontal and vertical
            x_intercept = x3
            y_intercept = y1
        elif m1 == 'inf':
            x_intercept = x1
            y_intercept = m2 * x_intercept + b2
        elif m2 == 'inf':
            x_intercept = x3
            y_intercept = m1 * x_intercept + b1
        else:
            x_intercept = (b2 - b1) / (m1 - m2)
            y_intercept = m1 * x_intercept + b1

        return x_intercept, y_intercept


if __name__ == "__main__":
    pass
```

APPENDIX D: perimeter_mesh_checker.py

```
"""
Project:    mesh-ppp
Author:     Alex Hanson | hanson.alex@gmail.com
Date:       5/15/2014

This is a test script to graphically verify the perimeter

call from terminal like
python -i perimeter_mesh_checker.py /folder/filename.stl -d .02 -o
"""

from pyqtgraph.Qt import QtGui
import pyqtgraph.opengl as gl
from trisurf import TriSurf

import argparse

#Command line argument parsing
description = """
Creates finds the perimeter and creates a mesh of a binary stl file
"""

parser = argparse.ArgumentParser(
    formatter_class=argparse.RawDescriptionHelpFormatter,
    description=description)

parser.add_argument('file_name',
                    metavar='File',
                    type=str,
                    help="Directory/filename of binary STL file to"
                        "process")

parser.add_argument('-delta',
                    '-d',
                    metavar='float',
                    type=float,
                    default=.01,
                    help="Mesh filtering parameter, typically ~ .00"
                        " - .00")

parser.add_argument('-original',
                    '-o',
                    dest='flag_o',
                    action='store_true',
                    help="displays the original object")
```

```
parser.add_argument('—projection',
                    '-p',
                    dest='flag_p',
                    action='store_true',
                    help="displays the filtered projection")

#parse args
args = parser.parse_args()

#setup app
app = QtGui.QApplication([])
w = gl.GLViewWidget()
w.show()
w.setWindowTitle('Perimeter checker')
w.setCameraPosition(distance=100)

#add grid item
g = gl.GLGridItem()
g.scale(2, 2, 1)
w.addItem(g)

#create original object
base = TriSurf(file_name=args.file_name)

#add original object
if args.flag_o:
    base.add_color()
    m1 = gl.GLMeshItem(meshdata=base,
                       computeNormals=False,
                       drawFaces=True)

    #m1.setGLOptions('additive')
    w.addItem(m1)

#ceate projection
projection = base.return_projection()
filtered_projection =
projection.return_filtered_mesh(delta=args.delta)

#add projection:
if args.flag_p:
    m4 = gl.GLMeshItem(meshdata=filtered_projection,
                       computeNormals=False,
                       drawFaces=False,
                       drawEdges=True,
                       edgeColor=[0, 0, 1, 0])

    w.addItem(m4)
```

```
    perimeter_vertexes, perimeter_Edges = \
        filtered_projection.get_perimeter_edges_and_vertexes()

    vertexes = filtered_projection.modified_vertexes
    perimeter_lines = vertexes[perimeter_vertexes]
    m6 = gl.GLLinePlotItem(pos=perimeter_lines,
                           color=[1, 0, 0, 1],
                           width=5)

    w.addItem(m6)

else:
    #return perimeter mesh
    perimeter_mesh = filtered_projection.return_perimeter_mesh()

    m7 = gl.GLMeshItem(meshdata=perimeter_mesh,
                       computeNormals=False,
                       drawFaces=False,
                       drawEdges=True,
                       edgeColor=[1, 1, 1, 0])

    w.addItem(m7)



# #add starting vertex
# l_start = np.zeros((2, 3), dtype=float)
# l_start[0] = filtered_projection.vertexes()[perimeterVertexes[0]]
# l_start[1] = l_start[0]
# l_start[1, 2] += .05
# m3 = gl.GLLinePlotItem(pos=l_start, color=[1, 0, 0, 1], width=5)
# w.addItem(m3)

#add y_unit
#l_y_unit = np.zeros((2,3),dtype=float)
#l_y_unit[0] = l_start[0]
#l_y_unit[1] = l_y_unit[0]
#l_y_unit[1,1] = l_y_unit[1,1]+.5
#m4 = gl.GLLinePlotItem(pos=l_yunit, color=[1,0,0,1], width=1)
#w.addItem(m4)
```