CERN

Summer Student Programme 2019

Generating Mesh Representations of VecGeom Solids

Prepared by
Murat Topak MSc in CS at EPFL

Supervisors: Andrei Gheata & Evgueni Tcherniaev

August 30, 2019

Contents

1. Introduction

The VecGeom [1] project aims at developing a high-performance library for geometry modelling, describing geometry in terms of 3D solid primitives. The library provides vectorized ray-solid intersection algorithms and other geometry-specific functionality that make use of fine-grained SIMD and SIMT parallelism.

This report covers the work done by me during the CERN Summer Student Programme between the dates 01.07.2019 and 30.08.2019. The project was initially to create mesh representations for VecGeom solids but then I had the chance to work on other problems too as will be explained in the next chapters.

2. First Sprint

2.1 Approach

The first problem was to come up with mesh representations for the numerous solids implemented in VecGeom library. A high level view of the solution is shown in Figure 1.

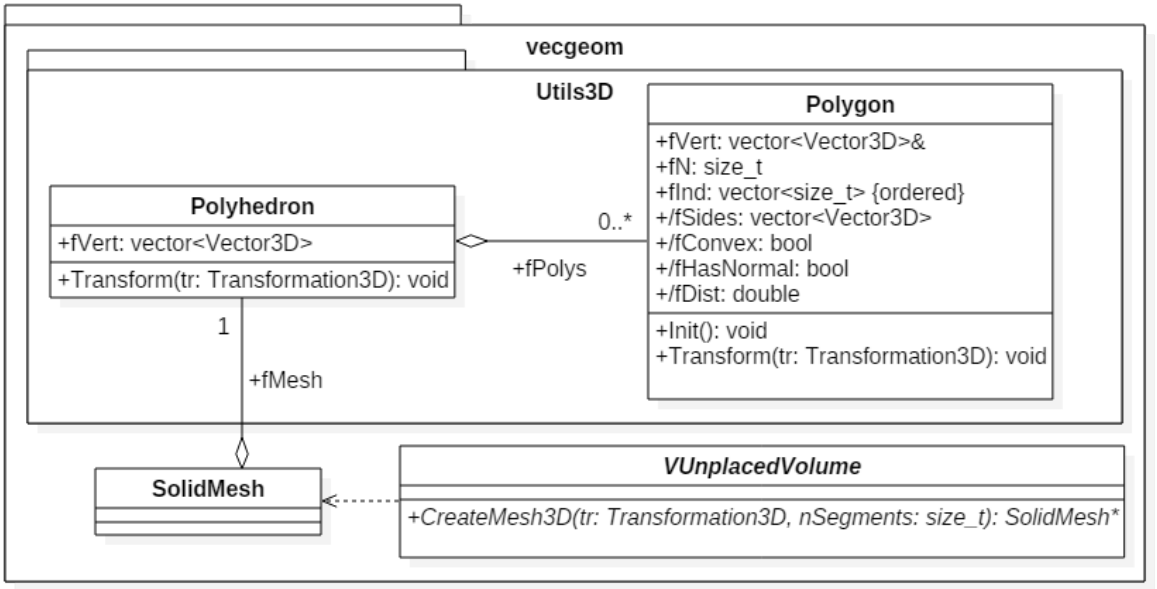

Figure 1: Class diagram of the approach.

When I started to work, *Polygon* and *Polyhedron* classes were exactly at the state shown in the figure, so all I had to do was to create a wrapper class called *SolidMesh* and declare the virtual method *CreateMesh3D* in the abstract parent class. Having this kind of relationship between classes, each derived solid may now implement its own *CreateMesh3D* to retrieve its mesh representation. The parameters to the method are a transformation object and a number specifying the accuracy of the model, only meaningful for smooth surfaces like a torus, sphere, cone, etc.

To explain the run-time relationship between the classes, an object diagram is given in Figure 2. In this diagram, it is clear that we use a vertex array shared by all polygons plus the polyhedron so as to save space. Having this kind of representation, each polygon only has to specify its indices. Then the coordinates may be retrieved by reading the corresponding indices in the vertex array.

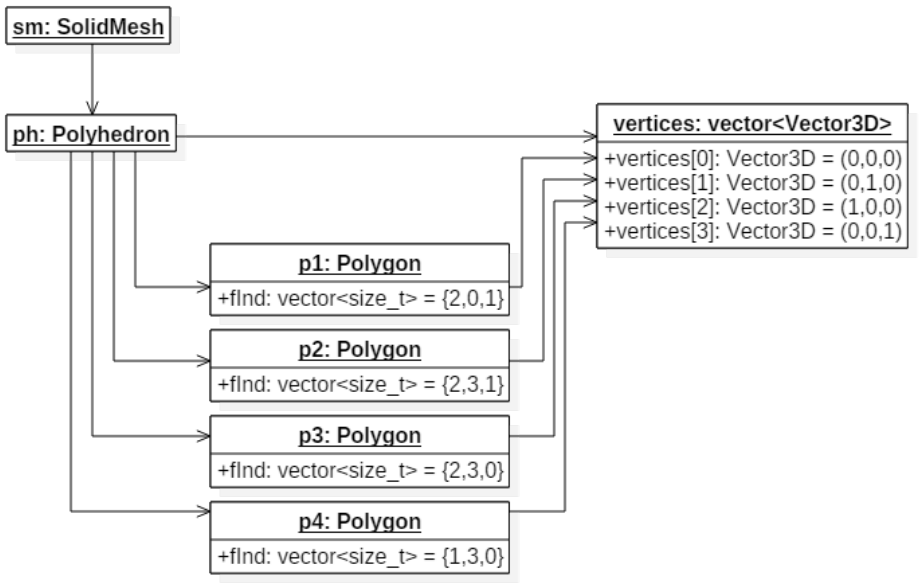

Figure 2: Object diagram of the approach.

The object diagram in the figure actually defines a tetrahedron when looked at carefully.

2.2 Results

The results followed the approach. I have started with the solids having an exact number of faces, and then moved to slightly complex ones which needed to parameter *nSegments*.

In the figures below you may check the generated meshes. The hard part of this task for me was to get used to all these classes, their internals, available methods, etc. so that I could retrieve the necessary information to draw the mesh.

Also, these solids are really flexible in their parameters so a torus for example can come in three shapes as in Figure 5. What makes this tricky is that degeneracies occur and need to be handled in some way. Degenaracies may exist if there are repeated vertices in the polygon or if there are collinear 3 points so that the polygon actually turns out to be a line, or even a point. This inevitably happens when generating points to form the mesh. For instance, one may think of an orb in Figure 4 as a stack of horizontal circles. Then you sample N points around each of these circles to draw the polygons but the problem is in the middle region the circle has a long radius while moving from middle to top and bottom this radius gets shorter, so as an extreme example, at the poles the same point is generated N times because the radius is zero there. Then it happens to be the case you end up with repeated vertices when trying to draw quads at the poles.

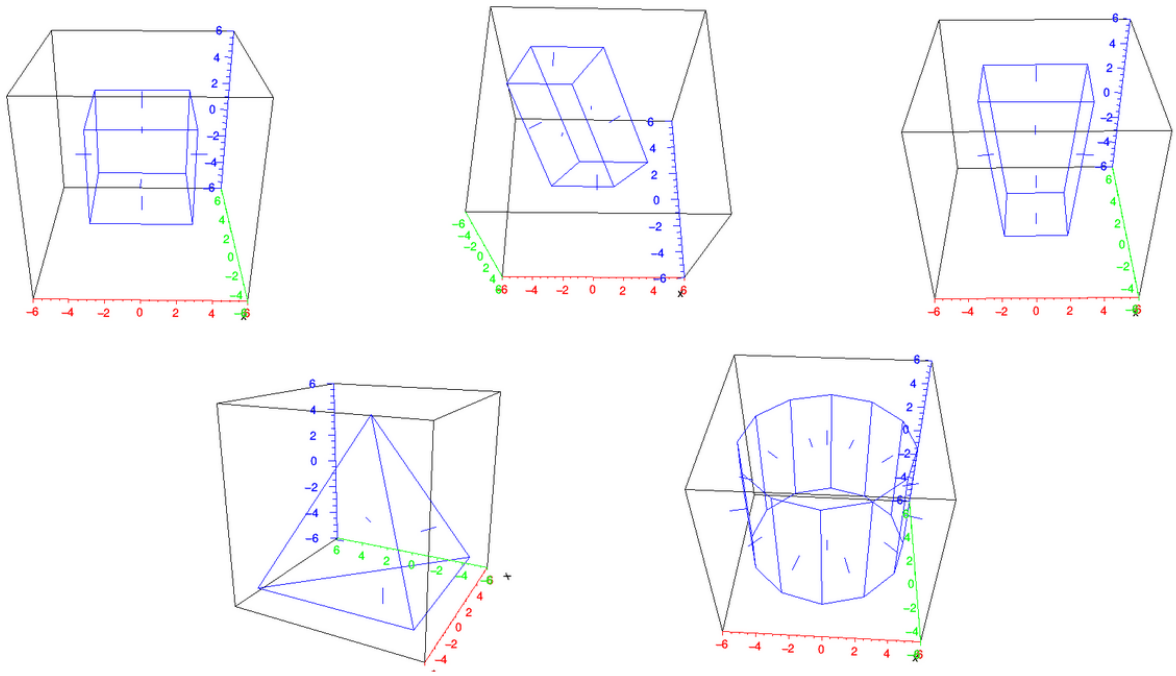

Figure 3: First round. Box, Parallelepiped, Trd, Tet, SExtruded.

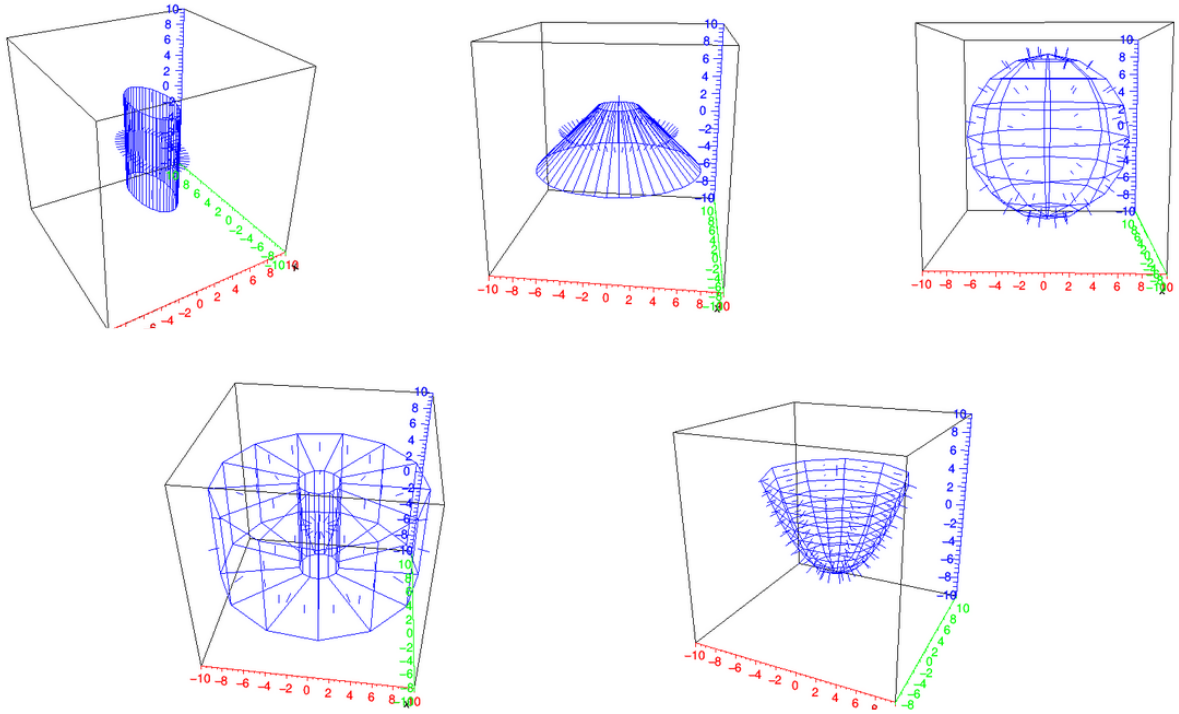

Figure 4: Second round. EllipticalTube, EllipticalCone, Orb, Tube, Paraboloid.

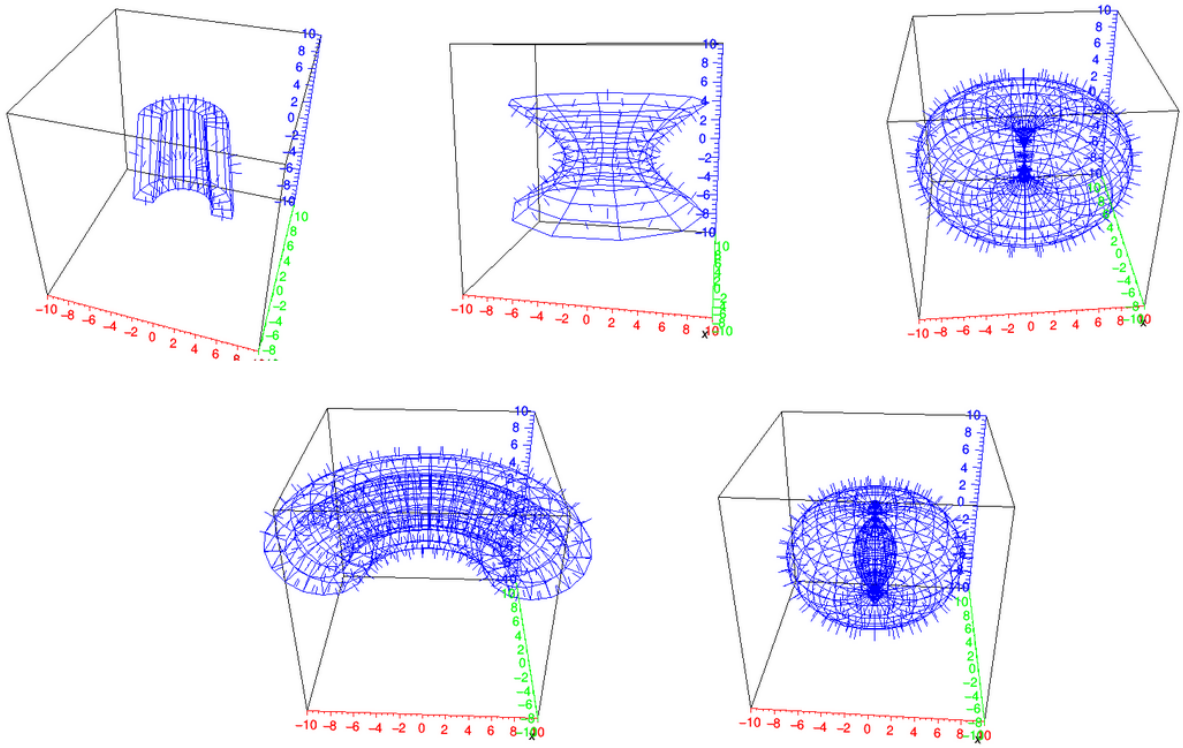

Figure 5: Third round. Cone, Hype, and three types of tori.

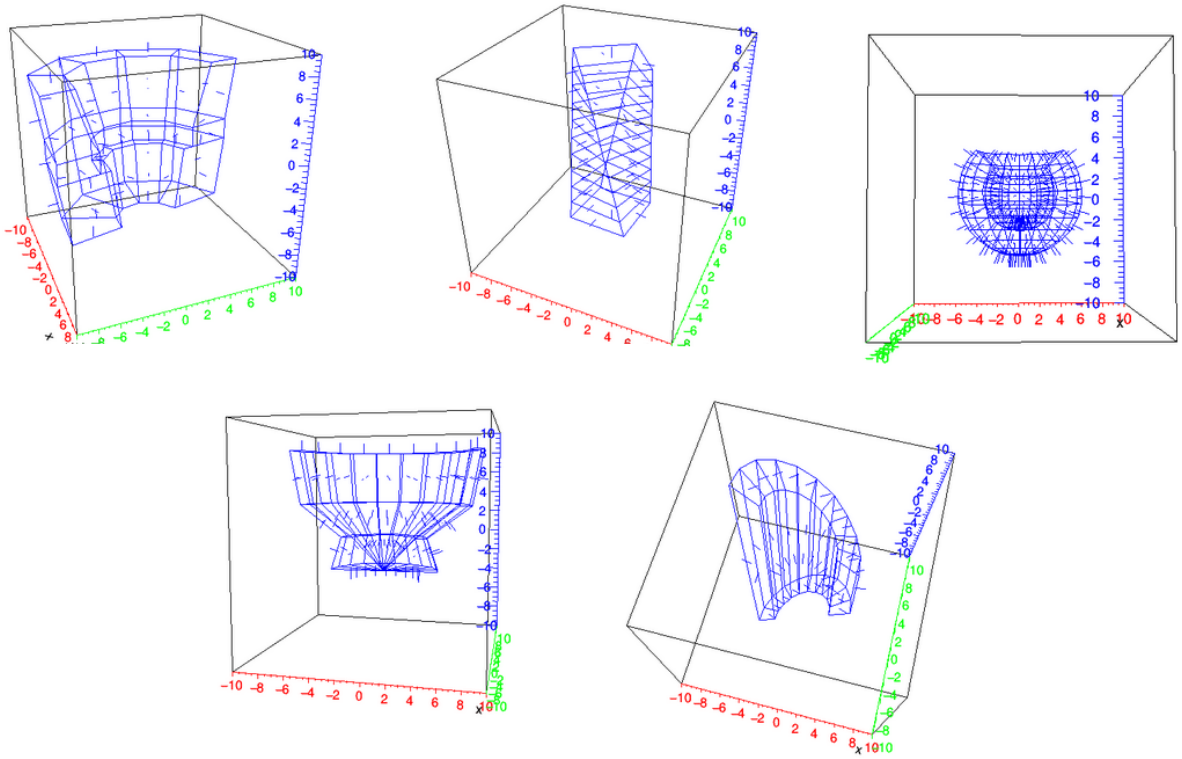

Figure 6: Fourth round. Polyhedron, GenTrap, Sphere, Polycone, CutTube.

3. Second Sprint

In the second part of the project, I dealt with problems related to polygons. Mainly, I implemented a point-in-polygon check [2] to see if a point is inside the polygon, an ear clipping triangulation to decompose a polygon into a set of triangles, and a polygon-polygon intersection algorithm to find the overlapping region between two polygons.

3.1 Approach

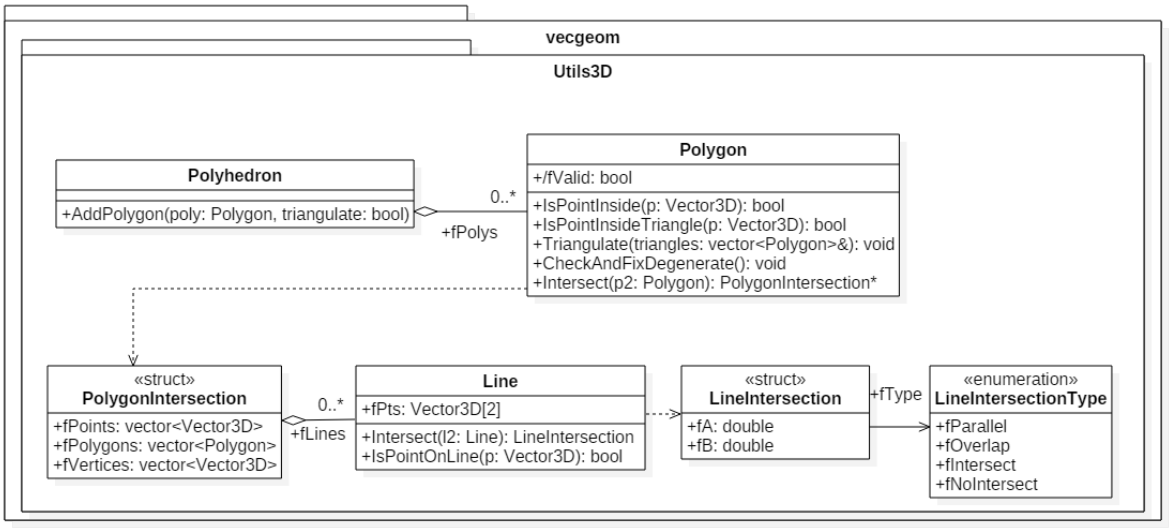

Figure 7: Class diagram of the approach.

My contribution to the *Utils3D* namespace is shown in Figure 7. A *Polygon* now also holds a boolean member called *fValid*, which is an indicator if the polygon is not actually a line or a point. Then there are methods mentioned earlier. In addition to those, also a point in triangle test, and a *CheckAndFixDegenerate* functionality has been implemented. The latter has its use for handling repeated vertices in polygons and setting *fValid* depending on the final number of vertices after repeated ones are deleted.

In the bottom part of the diagram, there are intersection structures such as *PolygonIntersection* and *LineIntersection*. Since a polygon-polygon intersection may result in a combination of points, lines, and polygons, the structure has the necessary members to support this varieties. For line intersections, on the other hand, we store the type of the result, plus two doubles to find the intersection point from each line. This is explained in the source code with ASCII drawings as in Figure 8.

```
/// Structure to hold line-line intersection results
///
/// In the case of fIntersect:
///                                    ^ lA.fPts[1]
///                                   /
///                                lA /
///                                  /
///                                 /              lB
///         <----------------------X----------------------->
///  lB.fPts[0]                   /                          lB.fPts[1]
///                              /
///                             /    X = lA.fPts[0] + fA * (lA.fPts[1] - lA.fPts[0])
///                            /     X = lB.fPts[0] + fB * (lB.fPts[1] - lB.fPts[0])
///               lA.fPts[0] v
///
///
///
/// In the case of fOverlap:
///
///                     lB.fPts[0]                      lB.fPts[1]
///         <-------------------<------------>------------->
///  lA.fPts[0]                              lA.fPts[1]
///
///
///                lB.fPts[0] = lA.fPts[0] + fA * (lA.fPts[1] - lA.fPts[0])
///                lB.fPts[1] = lA.fPts[0] + fB * (lA.fPts[1] - lA.fPts[0])
///
/// In the case of fNoIntersect and fParallel, fA and fB have no meaning.

struct LineIntersection{
    enum {fParallel, fOverlap, fIntersect, fNoIntersect} fType; ///< Intersection type
    double fA; ///< Used in calculating the intersection point (see the structure explanation)
    double fB; ///< Used in calculating the intersection point (see the structure explanation)
};
```

Figure 8: Explanation of the *LineIntersection* structure.

3.2 Results

Here the solutions to the above problems are given to show the work done. In Figure 9, random points are generated on the plane to test the point-inside-polygon (PIP) check algorithms. It seems there are not any problems for the general case as in the figure, but more tests with only the points on the edges or corners would be great to see how the algorithm performs for these special cases. PIP was implemented using crossing number algorithm, and PIT with cross and dot products. Simply, crossing number means for the point of interest a ray is shot in a direction and the number of crossings of this ray with the polygon is counted. If this number is odd then the point is inside, and outside otherwise.

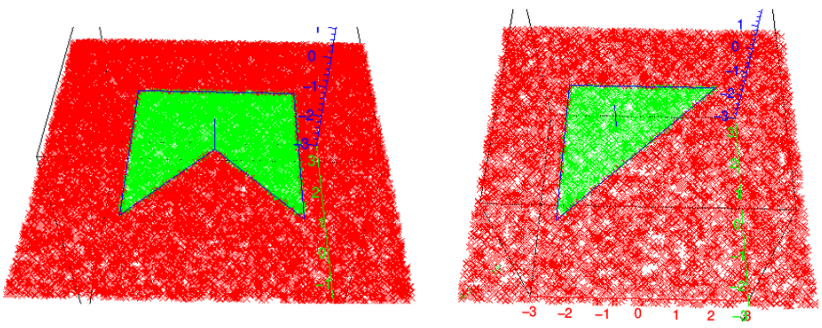

Figure 9: PIP and PIT tests.

The following figure shows the output of the triangulation algorithm on a simple example. Three red triangles are generated given the blue concave polygon.

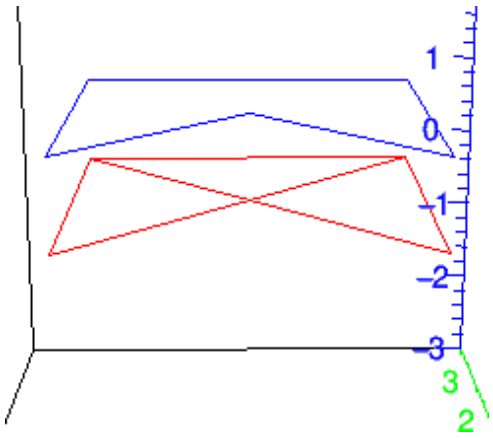

Figure 10: Polygon triangulation.

Then there comes the polygon-polygon intersection algorithm. There are two cases to consider if we label the two polygons of concern as *Subject* and *Clipper* and their normals as S_n, C_n:

- $\|\boldsymbol{S_n} \times \boldsymbol{C_n}\| \neq \mathbf{0}$: The result is a combination of lines and points.
- $\|\boldsymbol{S_n} \times \boldsymbol{C_n}\| = \mathbf{0}$: The result is a combination of polygons, lines, and points.

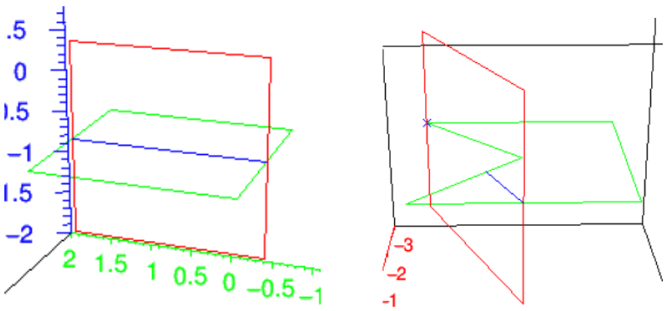

Figure 11: Subject and clipper polygons have different normals.

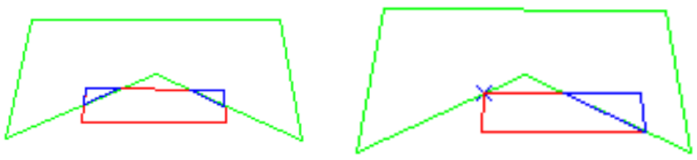

Figure 12: Subject and clipper polygons have the same normal.

Two cases are demonstrated in the Figure 11, 12. When both polygons have the same normal, it is the most complicated case and [3] is used to solve the problem.

4. Implementation notes

- There is a test file to reproduce the figures. Its name is TestMesh, and once run with no arguments it prints the usage.
- Right now the interface *CreateMesh3D* is protected against CUDA.
- Polygon-polygon intersection algorithm uses *std::unique* at one line and this is also problematic for CUDA, so this method is also protected.
- I made use of *kTolerance* defined in the library for *double* comparisons whenever the algorithm at hand was failing otherwise. However, not all the comparisons are made like this. One needs to consider this if any bug emerges.
- Fixing degenerate polygons right now only is for repeated vertices. That is, if the same vertex is specified two or more times like $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_3}, \boldsymbol{v_4}$, the algorithm fixes it to $\boldsymbol{v_1}, \boldsymbol{v_2}, \boldsymbol{v_3}, \boldsymbol{v_4}$. An improvement to this would be to catch and fix the colinear vertices so that no 3 points end up being colinear.
- The normal calculation in the library is problematic in the sense that it always finds the cross product between the first and second edge. However, if it happens that there is a concavity in those area, the normal we get is the opposite signed of what we are normally interested in. Or even worse, if there are collinear points in those area, then the normal is not defined at all.
- I experienced polygon intersection algorithm may return polygons with *cw* order although we use a *ccw* order. This leads to errors in debug mode as the normal for those polygons are defined in the reversed order. This can be fixed easily because while finding the intersection, the original normals to subject and clipper planes are already known, and then we may check the orders of indices of returned polygons by calculating their normals and comparing them to what we know already, eventually deciding if a reverse operation should be performed on the indices. At the moment, this is not a problem as I do not initialize the returned polygons. However, even if you want to initalize them, what should happen when the subject polygon has the normal $\boldsymbol{S_n}$, and the clipper polygon has the normal $-\boldsymbol{S_n}$? Which of the two normals should the intersection polygon inherit?
- There are a few solids left unimplemented: Tesselated, ScaledShape, GenericPolycone, and Boolean.
- Random polygon generation, and making use of PIP check in the intersection tests would be a nice step to validate the algorithm.

5. References

[1] "VecGeom / VecGeom," *GitLab*. [Online]. Available: https://gitlab.cern.ch/VecGeom/VecGeom.

[2] D. Sunday, *Inclusion of a Point in a Polygon*. [Online]. Available: http://geomalgorithms.com/a03-_inclusion.html.

[3] G. Greiner and K. Hormann, "Efficient clipping of arbitrary polygons," *ACM Transactions on Graphics*, vol. 17, no. 2, pp. 71–83, Jan. 1998.