Boolean Operations on Surfaces in VTK Without External Libraries

Release 1.00

Cory Quammen, Chris Weigle, and Russell M. Taylor II

May 12, 2011

Department of Computer Science
The University of North Carolina at Chapel Hill

Abstract

We have written a set of classes than enable computation of boolean operations on surface meshes using only VTK classes. In addition to being compatible with the VTK license, our contribution preserves surface mesh topology to the extent possible in boolean operations and passes point data and cell data through to the output mesh where possible.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3262]

Contents

1 Introduction

Let A and B be sets. A *boolean operation* can be used to define a third set C. Three common boolean operations on sets are:

union

$$C = A \cup B = \{x \,|\, x \in A \text{ or } x \in B\}$$

intersection

$$C = A \cap B = \{x \,|\, x \in A \text{ and } x \in B\}$$

difference

$$C = A - B = \{x \,|\, x \in A \text{ and } x \notin B\}$$

In geometric modeling, a closed, orientable 2-manifold surface mesh in 3D M_A may be considered the boundary of an infinite set of 3D points A. Given a second surface mesh M_B with the same properties, a boolean operation may be applied to obtain a third surface mesh M_C that bounds the infinite set of points C resulting from the boolean operation on A and B. For brevity, we use the phrase "boolean operation on surfaces" to refer to the process of determining M_C from M_A, M_B, and a boolean operation. These simple operations can be used to define complex geometries.

Rather than somehow converting two mesh representations to a set of points, performing the boolean operations on those points, and reconstructing a third mesh from the resulting points, direct operations on the bounding geometry based on signed distance fields and mesh-to-mesh clipping can be used. This article describes an implementation of those methods to realize boolean operations on surfaces in VTK.

2 Previous Work

Lloyd previously contributed a VTK class for computing boolean operations on surface meshes [**?**]. His contribution wrapped the GNU Triangulated Surface Library (GTS) in a VTK class that performed conversion between the VTK mesh data structure and the GTS mesh data structure and vice versa. This work filled a need often requested by users on the VTK mailing lists.

Unfortunately, his work is unable to be incorporated into the main VTK repository because of the incompatible GTS license. Furthermore, any projects using Lloyd's contribution are subject to the licensing requirements of GTS. Our contribution is not subject to an incompatible license as it makes use of no external libraries or code.

Furthermore, our contribution extend's Lloyd's work by copying point and cell data from the input meshes to the output meshes appropriately, interpolating point data where necessary. When portions of two input meshes are combined into one mesh, the common point and cell data in both meshes is copied to the output.

3 Mesh-to-Mesh Signed Distance Calculation

The distance field $f(\mathbf{x})$ of a mesh is defined as the distance from $\mathbf{x}$ to the nearest point on the surface defined by the mesh. The signed distance field is similar to the distance field, but the sign of the field is negative for locations $\mathbf{x}$ inside the space bounded by the mesh and positive for outside locations. To determine whether $\mathbf{x}$ is inside or outside the mesh, a vector $\overrightarrow{\mathbf{v}} = \mathbf{x} - \mathbf{m}$ is defined where $\mathbf{m}$ is the nearest point on the mesh. The sign of the distance field is the same as the dot product of $\overrightarrow{\mathbf{v}}$ with the *angle-weighted pseudonormal* of the mesh at $\mathbf{m}$ when the mesh is a closed, orientable 2-manifold surfaces in 3D Euclidean space [**?**].

The angle-weighted pseudonormal is defined separately for faces, edges, and points in a mesh, and forms a discontinuous vector field over the surface defined by the mesh. For a face, it is simply the face normal. For an edge, it is the average of the normals for the faces that share the edge. For a point, it is the sum of the angle-weighted normals from each face where the weight for a face normal is the angle between the two edges of that face incident to the point.

We introduce the `vtkImplicitPolyData` class that defines a signed distance function given an input `vtkPolyData`. This class is a subclass of `vtkImplicitFunction`, so it can be used by classes that operate on `vtkImplicitFunction` objects (see, e.g., `vtkSampleFunction`). The method `double EvaluateFunction(double x[3])` is overridden to return the signed distance, and `void EvaluateGradient(double x[3], double g[3])` is overridden to return the angle-weighted pseudonormal.

3.1 vtkImplicitPolyData

The class `vtkImplicitPolyData` operates on a `vtkPolyData` input. Use the method

```
void SetInput(vtkPolyData *input);
```

to specify the `vtkPolyData` on which it should operate. The class also has several options that can be set. Use

```
void SetNoValue(double value);
```

to set the value returned by `double EvaluateFunction(double x[3])` when an error occurs. Likewise,

```
void SetNoGradient(double value[3]);
```

to set the gradient returned when an error is encountered. Finally,

```
void SetTolerance(double tolerance);
```

use to determine when the absolute value of the signed distance is close enough to zero to be considered zero.

3.2 vtkPolyDataDistance

To support distance-based boolean operations, we introduce another class, `vtkPolyDataDistance`, that computes the distance from points in the first input `vtkPolyData` to a second by evaluating the signed distance field from the second input using the `vtkImplicitPolyData` class. Optionally, the distance from points in the second input `vtkPolyData` to the first can also be computed. These distances are stored as a point data field named "Distance".

`vtkPolyDataDistance` has some options. Use the methods

```
void SetSignedDistance(int value);
void SignedDistanceOn();
void SignedDistanceOff();
```

to turn computation of the signed distance on or off. By default, this option is on. If it is off, the unsigned distance function is computed.

Use the methods

```
void SetNegateDistance(int value);
void NegateDistanceOn();
void NegateDistanceOff();
```

to enable or disable negation of the signed distance field. If the `SignedDistance` option is off, then this option has no effect. `NegateDistance` option is off by default.

The methods

```
void SetComputeSecondDistance(int value);
void ComputeSecondDistanceOn();
void ComputeSecondDistanceOff();
```

enables computation of the signed distance for the second input. Finally, a convenience method

```
vtkPolyData* GetSecondDistanceOutput();
```

can be called to get the second output.

4 Boolean Operations on Surfaces Using Signed Distance

Boolean operations can be computed with only the signed distance field for each mesh. The sign of the distance field at a point in the mesh corresponds to whether that point is inside (negative), outside (positive), or on (zero) the other mesh; the surfaces bounding the volumes produced by the boolean operations described in Section 1 can therefore be defined as:

union

: The set of cells in each mesh such that the distance from each cell point to the other mesh is greater than or equal to zero.

intersection

The set of cells in each mesh such that the distance from each cell point to the other mesh is less than or equal to zero.

difference

The difference is defined as the set of cells of M_A whose points are a non-negative distance from M_B combined with the cells of M_B whose points are a non-positive distance from M_A.

5 Clipping One Surface Mesh with Another

While it seems possible to use the class `vtkClipPolyData` together with `vtkImplicitPolyData` to extract the necessary portions of each input surface to produce the boolean surfaces, doing so would produce inaccurate results. `vtkClipPolyData` treats the implicit function as a single piecewise linear function within each cell of the `vtkPolyData` being clipped. In general, an implicit function is not guaranteed to be a single piecewise linear function in each cell, and the same is true for the signed distance field. The result is that clipped surfaces produced by `vtkClipPolyData` will not be clipped at the true zero level of the distance field. Consequently, the clipped portions of each input geometry will not match at the zero level where they should fuse seamlessly. Figure 1 shows an example where the zero level of the distance field (black) does not match the surface intersection (white).

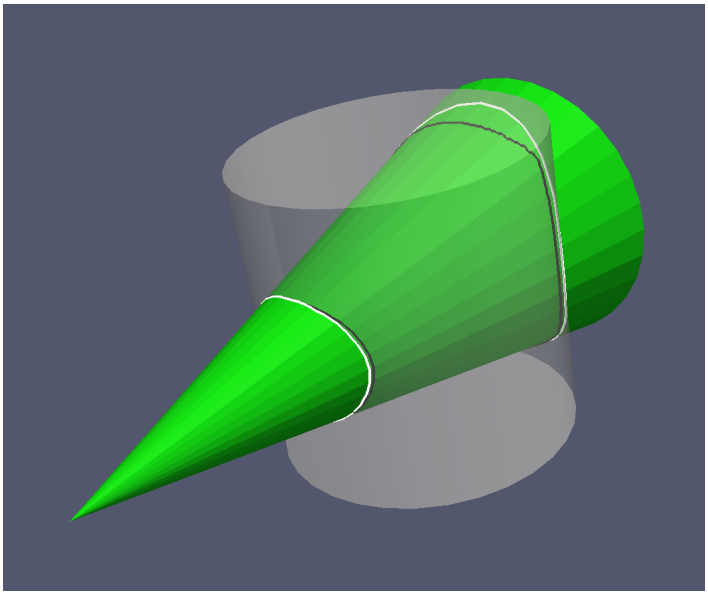

Figure 1: A case where the zero level of the distance field from the cylinder (black lines) computed on the cone does not match the intersection between the cylinder and the cone (white lines). Because of this mismatch, the class `vtkClipPolyData` is not suitable for accurate splitting of one surface mesh by another without further mesh processing.

Instead of clipping the input meshes based on the signed distance evaluated at mesh points, it is necessary to split each input mesh at the actual intersection of the two surfaces (hereafter referred to as the *surface intersection*). All points on the zero-level of the distance field are then completely defined by the points and edges of the surface intersection, and the surface intersection marks the accurate boundary between inside and outside portions of the mesh surface.

5.1 Identifying the Surface Intersection

Identifying the surface intersection between two surface meshes is the most complex part of the boolean operation algorithm. It involves identifying the intersection between each triangle in the first mesh with all triangles in the second mesh that intersect it. A triangle-triangle intersection test yields two endpoints (possibly the same) of the actual line segment defining the intersection when the triangles intersect. If the endpoints are the same, then the intersection line is degenerate, and it is not added to the set of intersection lines. The intersection between overlapping and co-planar triangles is more complicated; our work does not explicitly handle intersections between two coplanar triangles.

Oriented bounding box (OBB) trees are used to accelerate the identification of triangle-triangle intersections between meshes. Two `vtkOBBTree` objects are instantiated, one for each input mesh. The method `vtkOBBTree::IntersectWithOBBTree()` is then called on one OBB tree with the second OBB tree and a callback function passed as parameters. The callback function performs a more accurate triangle-triangle intersection test between the triangles in overlapping nodes from the two OBB trees and stores the line defining the intersection and other information in several data structures used later in the algorithm.

End points from the triangle-triangle intersections are stored in a `vtkPoints` object and the line cells are stored in a `vtkCellArray`. The points are merged using a `vtkPointLocator` object with a small tolerance, leading to many fewer connected components than if the line segments were stored as "line soup". Merging the end points also simplifies later operations. In addition, each end point is tested to see if it lies on an edge of the two input triangles. If so, a key-value pair is stored in a `std::multimap` where the key is the ID of the endpoint and the value is a 3-tuple consisting of the index of the triangle, the index of the triangle edge, and the index of the intersecting line.

The entries in the `std::multimap` described above are used to split the fully connected intersection lines to respect the topology of each input mesh. Because the end points of the intersection lines are merged during construction, they need to be split at locations where the mesh is split. For example, as shown in Figure **??**, a mesh representing a cube often has duplicate points at the corners to store three different normals, one for each face incident to the point. These normals enable the appearance of sharp edges on the mesh. If the surface intersection on a cube splits an edge that connects two of these corners, then two points should be inserted at the split point rather than one to ensure that two face normals are defined at the split point.

More generally, the end result of splitting the surface intersection should be that a copy of each line end point is produced for each set of triangles that share an edge. This is done by iterating over the entries of the edge point multimap. The first time a particular point ID is encountered in a multimap key-value pair, all other entries with the same key are checked to determine if the cell with the cell ID in the value shares the edge indicated by the first entry's value. If the entry is for a shared edge, then the entry is removed from the multimap and not processed any further. If not, the entry is left in the multimap for further processing.

If any multimap entries are left that have the current point ID as key, then that point lies on a split in the mesh and requires duplication. In this case, a copy of the point is inserted into the list of points for the intersection line data structure, and the first such entry in the multimap is read. All other entries in the multimap with the point ID as key are again examined to determine which share an edge with the first entry. Entries that do not share an edge are left in the multimap. For entries that share an edge, the point ID in the line referenced by the line ID stored in the entry is replaced with the ID of the newly duplicated point. The entry is then removed from the multimap. The process described in this paragraph is repeated if any entries are left that have the current point ID as key. Otherwise, the first multimap entry is removed.

During the above process, another auxiliary data structure is built. Every time a point is duplicated, a key-value pair is added to a map where the key is the point ID and the value is the cell ID from which the point

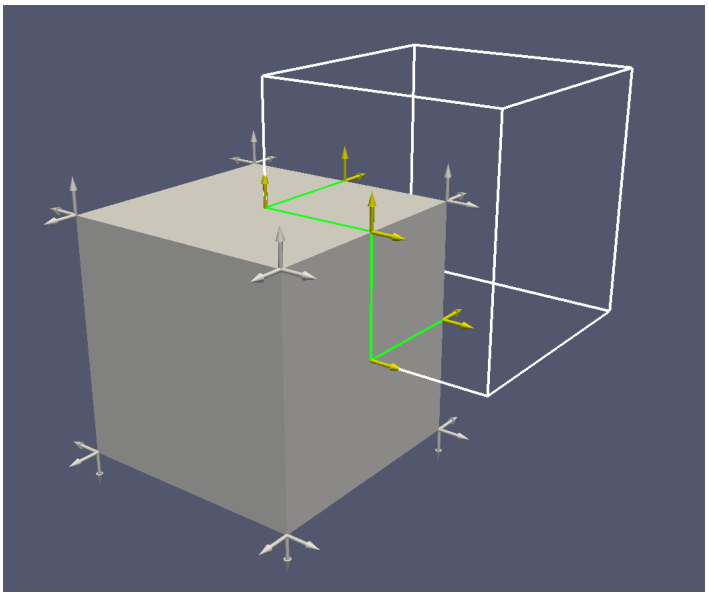

Figure 2: Splitting a mesh with duplicate normals. The box object has three duplicate points at each corner each of which has a different normal (gray arrows). The surface intersection between the box in the front (gray, solid) and the box in the back (white, outline) is shown in green. Points from the intersection lines that lie on the duplicate edges of the box are also duplicated by the mesh splitting procedure. The normal at a duplicated point, along with other point data, is interpolated from the cell that contains the intersection line that references that point.

data at the point should be interpolated. It does not matter which cell ID from the set of cells that share an edge is associated with a point on that edge because the interpolation results will be the same.

After the intersection lines are split, the points in the input mesh and their associated point data are copied to the points in the output mesh. The points from the intersection lines are then appended to the point set in the output mesh, and the point data is interpolated at those points.

5.2 Splitting the Mesh

After the intersection lines are split for a mesh, the next step is to identify candidate cells for splitting. A cell is a candidate cell in two cases:

1. The cell index is in the intersection line map, meaning that intersection lines lie inside the cell.
2. The cell is the neighbor of a cell identified by case 1.

The second case is important because one of the mesh-mesh intersection lines may have an endpoint on the edge of one cell, but no line that uses that endpoint on the cell neighbor across that edge. The cell neighbor needs to be split to avoid introducing a T-junction and therefore a hole in the output mesh.

Splitting proceeds on a cell-by-cell basis. For each cell that needs splitting, the following points and lines are gathered:

1. The three points that define the cell (green spheres in Figure 3).

2. The boundary lines of the cell (red lines in Figure 3).
3. The split intersection lines that lie in the cell (if any) (white lines in Figure 3).
4. The points that define the lines in 3 (purple spheres in Figure 3).
5. Points from neighboring cells that lie on a cell edge but which were not already added in 4 (orange sphere in Figure 3).

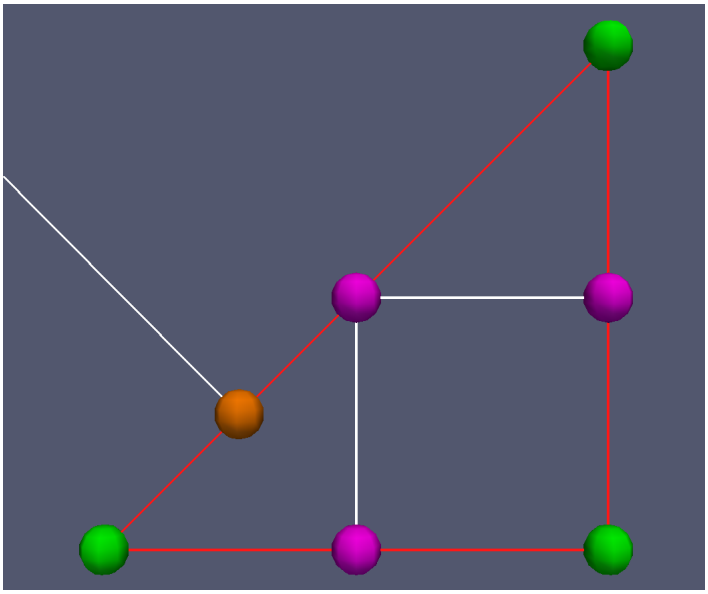

Figure 3: The points and lines required for splitting a cell.

The lines collected above are used to constrain a new triangulation of the points. In other words, these lines will be present in the output mesh. Special processing of the boundary lines is required to achieve the desired split. Specifically, boundary lines must be split at points that lie on them. Boundary line splitting is achieved by sorting points on the boundary lines according to the angle between the first edge of the cell and the vector formed by subtracting the cell center from the point. Adjacent points in the sorted list are then connected with lines, include a line between the last and first points in the list. Because the last boundary point is connected to the first, the absolute order of the boundary points is not important.

The `vtkDelaunay2D` class can be used to determine a triangulation from the cell, surface intersection lines, and boundary points and lines. A transformation of the points collected above that rotates them to the XY-plane is computed and set as the transform for the `vtkDelaunay2D` object. A new `vtkPolyData` object that contains the collected points and lines is defined and passed as both the input and constraint source to a `vtkDelaunay2D` object. The line point indices are renumbered to point to their locations in the new `vtkPolyData` object, and a map from the original indices is created to remap the output cell indices from the triangulation to the point indices in the output mesh. If a point in the surface intersection is within some small tolerance distance from a point in the surface mesh, the point in the surface mesh is used instead. This prevents unintentional splitting of the surface mesh.

5.3 vtkPolyDataIntersection

We have defined a new class `vtkPolyDataIntersection` that encapsulates the algorithm described above to compute the surface intersection between two surface meshes and optionally split the input meshes by the surface intersection. The surface meshes must consist of triangles. Two options are available for this filter:

```
void SetSplitFirstOutput(int value);
void SplitFirstOutputOn();
void SplitFirstOutputOff();

void SetSplitSecondOutput(int value);
void SplitSecondOutputOn();
void SplitSecondOutputOff();
```

These options specify whether the output meshes should be split. Each option is on by default.

6 Putting it All Together

There are two ways to use the classes we have developed to compute the surface mesh resulting from a boolean operation on two surface meshes: setting up a VTK pipeline or using a new class `vtkPolyDataBooleanOperationFilter`.

6.1 Setting up a VTK Pipeline for Boolean Operations on Surfaces

To compute a surface resulting from a boolean operation on surface meshes, the pipeline of classes shown in Figure 4 can be used to compute the desired surface. The two arrows between `vtkPolyDataIntersection` and `vtkPolyDataDistance` indicate that the two output surfaces from `vtkPolyDataIntersection` are set as the input to `vtkPolyDataDistance`. The two instances of `vtkThreshold` operate on different outputs of `vtkPolyDataDistance`.

6.2 vtkPolyDataBooleanOperationFilter

For convenience, we have included a single class that replicates the functionality class for computing boolean operations on surface meshes, `vtkPolyDataBooleanOperationFilter`, that encapsulates the functionality of the pipeline above. This class has three options. The desired boolean operation is specified with one of

```
void SetOperationToUnion();
void SetOperationToIntersection();
void SetOperationToDifference();
void SetOperation(int operation);
```

The default operation is union.

When computing the difference boolean operation, part of the resulting surface mesh will have reversed normals and a reversed order of cell points. The `ReorientDifferenceCells` option is used to enable reversal of the normals and reorientation of these cells and can be set using the methods

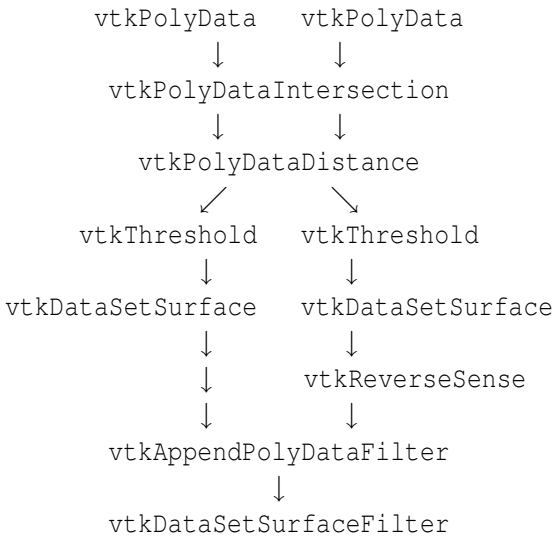

Figure 4: Possible pipeline for computing boolean operations on surface meshes. The instance of `vtkReverseSense` is optional and can be used when the boolean difference is desired to reorient the intersection portion of the second surface so that its normals face toward the outside of the resulting surface.

```
void SetReorientDifferenceCells(int value);
void ReorientDifferenceCellsOn();
void ReorientDifferenceCellsOff();
```

Lastly, the `Tolerance` sets a threshold on the absolute value of a distance field value below which the distance is considered to be zero.

7 Results

Results from boolean operations on various pairs of geometric objects are shown in Figure 5.

8 Limitations

The `vtkPolyDataIntersection` class operates only on triangulated surface meshes, and it does not properly handle intersections between coplanar triangles. When two triangles are determined to be coplanar, the intersection information is discarded. If this occurs, then the output of a boolean operation may not be exactly correct at the location of the coplanar triangles.

9 Software Requirements

The source code contributed with this article has been built against VTK git commit 9f132a45a7bece15d77ad74723c6378f2f29d869.

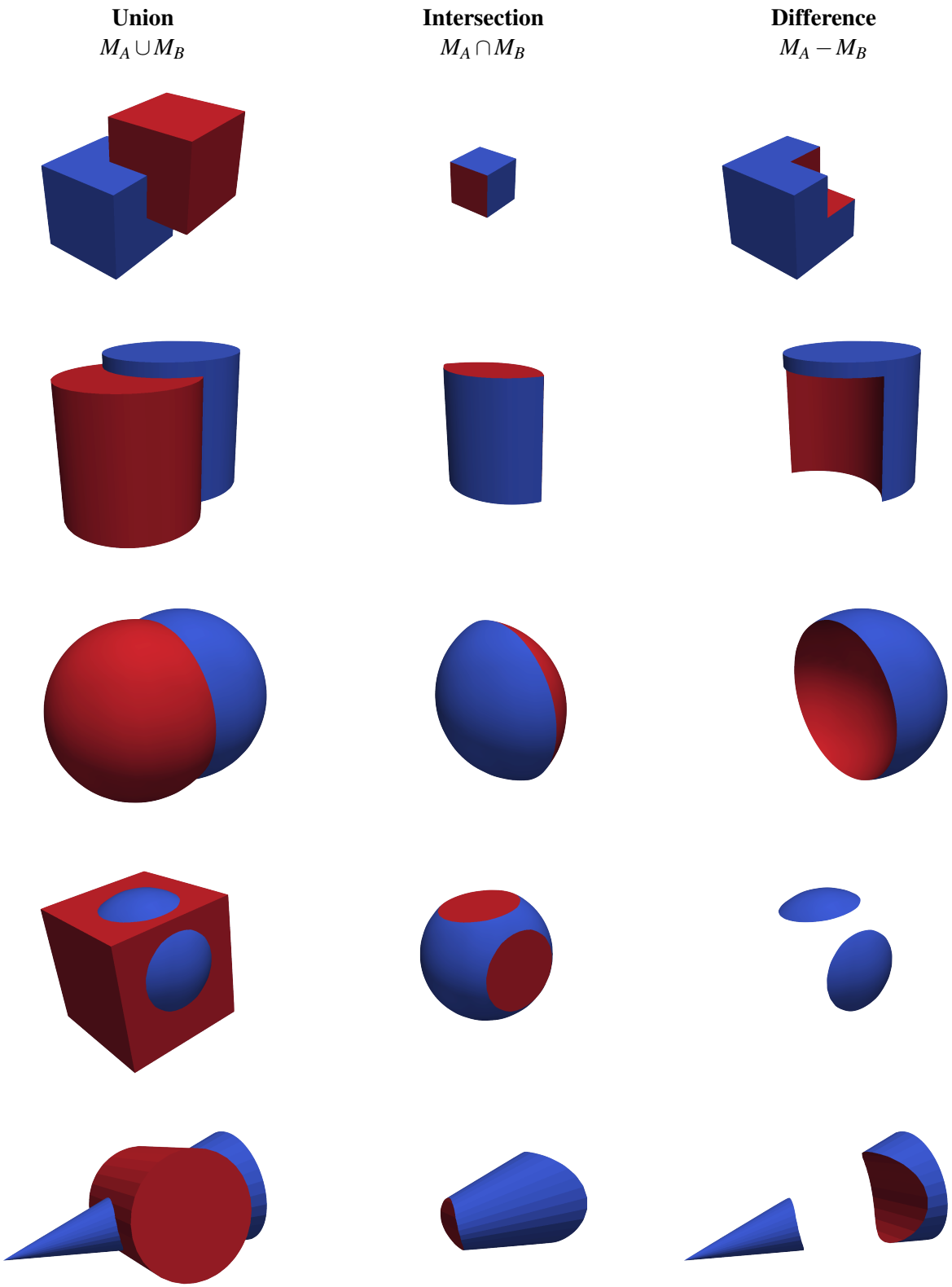

Figure 5: The blue parts of the surface come from the first input mesh M_A and the red parts come from the second input mesh M_B.