Data Structures for Fast Polyhedral Intersections in 3D

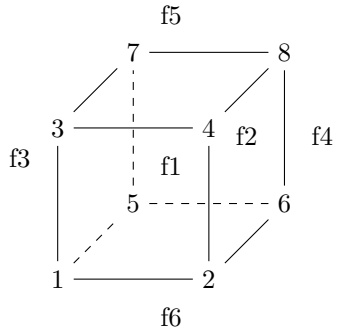
Joseph Donato* Daniel Shevitz* Rao Garimella*

1 Abstract

The operation of intersecting two polyhedra is fundamental to intersection based remap and interpolation[1]. This is done under the assumption that one of the two polyhedra is convex and we clip the other polyhedra with the planes representing the faces of the convex polyhedra. R3D[3] is a widely used library for such operation which implements the algorithm of Sugihara[4]. In this paper, we describe the implementation of R3D, its shortcomings, and propose a novel implementation named J3D which provides substantial performance improvements.

2 Introduction

Intersection based remap which conservatively interpolates values from a source mesh onto a target mesh heavily relies on polyhedral intersections. That is, for each target cell, intersecting source cells are determined and the target cell field value(s) is calculated by using the moments of each intersection as weights. Before discussing intersection routines, it is important to define and distinguish between the boundary representation (BREP) and the 1-skeleton representation of a polyhedra. A boundary representation describes a hierarchy between faces and the nodes on each face ordered in the outward normal direction. The 1-skeleton is a graph representation of the polyhedra in which each node has a list of nodes which are edge-connected to it which we also refer to as neighbors. Within the context of this paper however, we will also include the requirement that the neighbors of a node are ordered counter-clockwise relative to the exterior of the polyhedra (Table 1).



BREP	
face	vertices
f1	[1,2,4,3]
f2	[7,8,6,5]
f3	[3,7,5,1]
f4	[8,4,2,6]
f5	[3,4,8,7]
f6	[1,5,6,2]

1-skeleton	
vertex	neighbors
1	[3,5,2]
2	[4,1,6]
3	[7,1,4]
4	[8,3,2]
5	[7,6,1]
6	[8,2,5]
7	[8,5,3]
8	[4,6,7]

Table 1: A hexahedron with labeled vertices and faces. The left table demonstrates the corresponding BREP with faces and their nodes ordered in the outward normal direction. The right table demonstrates the corresponding 1-skeleton with nodes and their neighbors ordered counter-clockwise relative to the exterior of the the hexahedron.

Meshes typically represent their cells/polyhedra in the boundary representation and R3D converts them into a 1-skeleton. R3D implements the robust intersection algorithm by Sugihara[4] which is more natural in the 1-skeleton since only local information (neighbors) has to be updated. The intersection algorithm has two stages referred to as clip and cap. In the clip step, the edges which intersect the plane are identified, new vertices are introduced on such edges, and vertices above the plane are removed. After this however, we are left with open faces, or, faces which no longer form a closed cycle, as well as an open volume. In the cap step, the new nodes are linked together to construct the cycle or capping face to seal these open faces (Figure 1).

*Los Alamos National Laboratory.

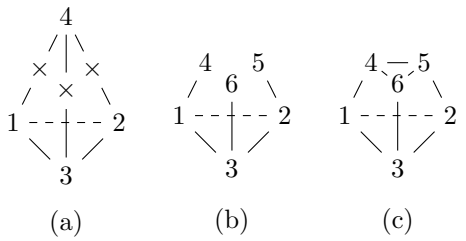


Figure 1: The clip and cap algorithm divided into three stages. (a) Points on the edges of the polyhedra which lie on the plane (denoted by ×) are identified. (b) Vertices above the plane are removed and the new vertices are labeled and included in the 1-skeleton. (c) The new vertices are linked to form the capping face.

3 R3D's Implementation

R3D represents its polyhedra as a 1-skeleton but with the additional restriction that every node has a valence of three to take advantage of loop unrolling by compilers. This representation proves to be disadvantageous when initializing from a BREP and computing moments. In the case of initialization, if a polyhdera is provided which has a maximum valence greater than three, then R3D has to duplicate vertices and introduce edges which hurts performance (Figure 2). To compute the moments, for each face, one must pick a starting vertex on the face and walk the face to construct triangular facets. Each of these facets are oriented counter-clockwise relative to the exterior of the polyhedra. In addition to the origin which is used as a reference point, these facets define tetrahedra to be integrated over[2]. This is very straightforward and performant in the BREP, but in the case of R3D, the neighbors of each node on the walk must be searched in order to ensure the next node in our walk is properly selected.

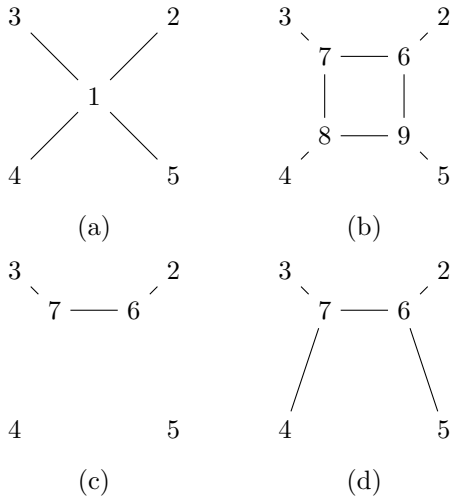


Figure 2: R3D's procedure (a)-(d) of duplicating nodes to enforce that all node have a valence of three. (a) Vertex 1 is identified to be a vertex with valence of four. (b) Vertex 1 is replaced with a cycle composing of four vertices 6, 7, 8, and 9 with coordinates equivalent to that of vertex 1. (c) Two edge connected vertices 8 and 9 from the cycle are removed. (d) The vertex clockwise from 8 in the cycle (7) is joined with 8's old neighbor (4). The vertex counter-clockwise from 9 in the cycle (6) is joined with 9's old neighbor (5).

4 J3D's Implementation

In our novel implementation J3D, we also represent the polyhedra in the 1-skeleton but without any restrictions on the valence of nodes. In addition, we also enrich the 1-skeleton with some data structures that eliminate the small searches described in the previous section. This becomes advantageous since degenerate vertices are no longer necessary in the initialization step and the memory footprint is substantially reduced as a result. However, with an arbitrary valence, additional data structures and algorithms are necessary to efficiently initialize the 1-skeleton, clip and cap, and compute moments. Given a face from a BREP and the ordered set of nodes forming that face, a directed edge is a pair of sequential nodes. From this, the data structures we introduce are two maps M_{prev} and M_{next}. M_{prev} maps a directed edge to the vertex preceding it in the face walk and M_{next} maps a directed edge to the vertex succeeding it in the face walk. Alternatively, $M_{prev}(A, B)$ can be viewed as mapping to the node following B in the counter-clockwise iteration of neighbors around A. In addition, $M_{next}(A, B)$ can alse be viewed as mapping to the node following A in the clockwise iteration of neighbors around B.

4.1 1-skeleton Initialization For the sake of initializing the 1-skeleton, we let V be the set of vertices in the 1-skeleton and we define a map $Start : V \to V$ which for a vertex v simply returns a neighbor of v. From here, the algorithm to initialize the 1-skeleton with counter-clockwise node ordering is as follows.

ALGORITHM 4.1. Counter-Clockwise Neighbor Initialization

Require: M_{prev}, $Start$.
Ensure: 1-skeleton G with counter-clockwise neighbor ordering.
  function NEIGHBORSINITIALIZATION(G, M_{prev}, S)
    for $v \in G.vertices$ **do**
      $G.neighbors[v].append(Start(v))$
      $v_{prev} \leftarrow M_{prev}(v, Start(v))$
      while $v_{prev} \neq Start(v)$ **do**
        $G.neighbors[v].append(v_{prev})$
        $v_{prev} \leftarrow M_{prev}(v, v_{prev})$
      end while
    end for
  end function

4.2 Cap Before defining the new algorithm for cap, it is important to note that all new vertices on a polyhedra after clip and cap always have a valence of three. Moreover, all new vertices immediately after the clip step will have only a single valid neighbor which is a vertex from the old polyhedra kept after clip (Figure 1b). We will set the first element in the list of neighbors of a new vertex to this valid neighbor.

ALGORITHM 4.2. Capping After Clip

Require: M_{next}
Require: 1-skeleton G with open faces.
Ensure: 1-skeleton G is a valid polyhedra with closed faces
  function CAP(G, M_{next})
    for $v_{new} \in G.new\_vertices$ **do**
      $prev \leftarrow v_{new}$
      $curr \leftarrow prev.neighbors[0]$
      $next \leftarrow M_{next}(prev, curr)$
      while $next \notin G.new\_vertices$ **do**
        $prev \leftarrow curr$
        $curr \leftarrow next$
        $next \leftarrow M_{next}(prev, curr)$
      end while
      $v_{new}.neighbors[1] \leftarrow next$
      $next.neighbors[2] \leftarrow v_{new}$
    end for
  end function

4.3 BREP Traversal for Moment Calculation The map M_{next} will also prove necessary for efficient moments computation. To compute the volume or the zeroth order moment for example, we traverse each face and construct triangular facets from a starting vertex on the face. Each facet is defined by three vertices A, B, and C that are oriented counter-clockwise relative to the exterior of the polyhedra. By using the origin O as a reference point, the volume of the tetrahedra formed by O, A, B, and C is the following[2].

$$vol_T(A, B, C) = \frac{1}{6}\begin{vmatrix} x_A & y_A & z_A \\ x_B & y_B & z_B \\ x_C & y_C & z_C \end{vmatrix}$$

With this, the volume of the whole polyhedra can be computed via the following algorithm

ALGORITHM 4.3. Volume Computation

Require: M_{next}
Require: 1-skeleton G
Ensure: Volume vol of the polyhedra.
  function BREPWALK(G, M_{next}, vol)
    $vol \leftarrow 0$
    $visited\_edges[G.num\_vertices][G.num\_vertices]$
    for $v \in G.vertices$ **do**
      for $n = 1 : G.num\_neighbors[v]$ **do**
        if not $visited\_edges[v][G.neighbors[v][n]]$ **then**
          $prev \leftarrow v$
          $curr \leftarrow G.neighbors[v][n]$
          $next \leftarrow M_{next}(prev, curr)$
          $visited\_edges[prev][curr] \leftarrow true$
          while $next \neq v$ **do**
            $vol \leftarrow vol + vol_T(v, curr, next)$
            $prev \leftarrow curr$
            $curr \leftarrow next$
            $next \leftarrow M_{next}(prev, curr)$
            $visited\_edges[prev][curr] \leftarrow true$
          end while
          $visited\_edges[curr][next] \leftarrow true$
        end if
      end for
    end for
  end function

5 Benchmarks

For our benchmarks we test against a set of polyhedra listed below. For each of these, we run 10^7 trials. For each trial we initialize the internal data structure from a BREP representation, clip with a random plane, then compute the volume. The runtimes listed below are the CPU time in seconds provided by HPCToolkit. The last column is the R3D runtime divided by the J3D runtime.

polyhedra	J3D	R3D	R3D/J3D
tetrahedra	3.16	3.66	1.15
triangular prism	4.46	4.62	1.03
cube	5.63	6.43	1.14
quadrilateral pyramid	3.62	10.1	2.79
pentagonal pyramid	4.70	10.9	2.31
hexagonal pyramid	5.32	13.1	2.46
octahedron	4.95	13.3	2.68
triangulated cube	14.5	61.7	4.25

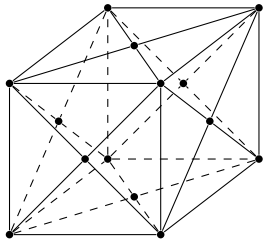


Figure 3: A visual representation of a triangulated cube to serve as a visual aid. Note that none of its vertices have a valence of three.

References

[1] A. Herring, C. Ferenbaugh, C. Malone, D. Shevitz, E. Kikinzon, G. Dilts, H. Rakotoarivelo, J. Velechovsky, K. Lipnikov, N. Ray, and R. Garimella, *Portage: A modular data remap library for multiphysics applications on advanced architectures*, Journal of Open Research Software, (2021).

[2] P. Koehl, *Fast recursive computation of 3d geometric moments from surface meshes*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 34 (2012), pp. 2158–2163.

[3] D. Powell, *r3d: Software for fast, robust geometric operations in 3d and 2d*, Los Alamos National Laboratory, (2015).

[4] K. Sugihara, *A robust and consistent algorithm for intersecting convex polyhedra*, Computer Graphics Forum, 13 (1994), pp. 45–54.