SolidMesh – Data Structure for Global Mesh Operations

Gábor Fábián[1]

[1]ELTE Eötvös Loránd University, Budapest, Hungary, insomnia@inf.elte.hu

Abstract. In this paper, we design a data structure for representing polyhedra that allows the efficient execution of global operations. Our research is motivated by the following contradiction. In computer-aided design or modeling tasks, we generally represent surfaces using edge-based data structures such as Winged edge, Half-edge, or Quad-edge. In contrast, real-time computer graphics represent surfaces with Face-vertex meshes, since for surface rendering, there is no need for the explicit representation of edges.

In most cases, when mainly local modifications are used (e.g. vertex split, edge flip, face removal), traditional Winged edge and Half-edge data structures perform well. However, for global operations (affecting a large number of vertices, edges, and faces), the advantages of edge-based data structures seem to diminish. We will show a novel data structure for the representation of triangle meshes, that is based on the concept of Face-vertex meshes. We will discuss, what additional topological information needs to be stored to obtain fast traversal algorithms without representation of edges. In this paper, we compare the proposed data structure with the industry-standard Half-edge data structure in several aspects.

Keywords: triangle mesh, mesh processing, topology, subdivision, Half-edge
DOI: https://doi.org/10.14733/cadaps.2025.586-599

INTRODUCTION

Topological data structures play an important role in computer graphics and computer-aided design. They provide efficient and effective means for representing and manipulating complex geometric structures. These data structures heavily rely on the topological properties of triangulations of surfaces. This knowledge is used to quickly respond to queries regarding certain connectivity information.

The first topological data structure was the Winged-edge, which enables the store of polyhedra in a way that allows efficient execution of complex geometric operations [2]. The central element of the data structure is the edge since storing a constant number of geometric entities is required to fully understand the local neighborhood relations of edges, unlike, for example, faces where the number of vertices per face can vary.

Doubly connected edge list or Half-edge data structure extends the concept of the Winged-edge structure, the key idea is to represent each edge as a pair of directed Half-edges, providing efficient navigation between

© 2025 U-turn Press LLC, http://www.cad-journal.net

adjacent faces, edges, and vertices [14] [5] [13]. It is perhaps not an exaggeration to state that the Half-edge data structure is an industrial standard. Libraries as widely used as OpenMesh [4] and CGAL [10] utilize it for implementing mesh operations.

Another edge-based representation is the Quad-edge, where by swapping the pairs of vertices and faces attached to edges, we can easily determine the dual of a mesh, therefore it can aid in problems such as calculating the Voronoi diagram of a polyhedron [9].

The Render dynamic mesh is a representation that combines aspects of both the Face-vertex and Winged-edge data structures, hence edges also play an important role here. The Render dynamic mesh was specifically designed for the rapid execution of subdivision algorithms, but it may not be efficient for performing local manipulations [16]. There are some other other concepts, a very nice collection and comparison of boundary representations of solid geometries can be found in [18].

As can be seen, most popular topological data structures explicitly include edges, but the edges are the ones that are completely unnecessary for rendering. It is reasonable to question whether edges can be omitted from a topological data structure. In this article, we present an approach that omits edges, closely related to Face-vertex meshes, providing complete topological information with less overhead than edge-based representations.

TOPOLOGICAL BACKGROUND

In our work, we considered polyhedrons defined by triangular faces. The surface of a polyhedron is a surface also in terms of topology, i.e. it is an orientable compact 2-manifold, and there are some direct consequences of this fact, that are explicitly used in mesh representation. It is important to emphasize that in this study we only dealt with closed surfaces. In topological terms, a closed surface is a 2-manifold *without boundary*. Later on, we will not highlight this, but when we mention 2-manifolds, we are thinking of 2-manifolds without boundaries.

The Euler-formula for polyhedron states that if a polyhedron has n vertices, ℓ edges, and m faces then

$$\chi = n - \ell + m = 2(1-g),$$

where χ is the Euler-characteristic of the polyhedron, g is the genus of its surface [11]. Roughly speaking, the genus determines the fundamental topological surface with which our polyhedron is topologically equivalent. The classification theorem of connected orientable compact 2-manifolds states that there are only 2 fundamental surfaces, the sphere $g = 0$ and g tori "glued together". Since the genus of most of the objects around us is quite small (much less than n, ℓ, m), we can assume that $\chi \approx 0$. Another classic result of topology is the triangulation theorem of 2-manifolds, which states that every 2-manifold is topologically equivalent to the polyhedron of a 2-dimensional simplicial complex, in which every 1-simplex is a face of exactly two 2-simplices. The direct consequence of this theorem is, that in each polyhedron each edge corresponds to exactly 2 triangles, and each triangle corresponds to exactly 3 edges, i.e.

$$2\ell = 3m \iff \ell = \frac{3}{2}m.$$

Using this formula:

$$0 \approx \chi = n - \ell + m = n - \frac{3}{2}m + m = n - \frac{1}{2}m \iff m \approx 2n,$$
$$0 \approx \chi = n - \ell + m = n - \ell + \frac{2}{3}\ell = n - \frac{1}{3}\ell \iff \ell \approx 3n.$$

As a consequence, if we assume that a polyhedron consists of only triangular faces, then it has approximately $2n$ faces and $3n$ vertices.

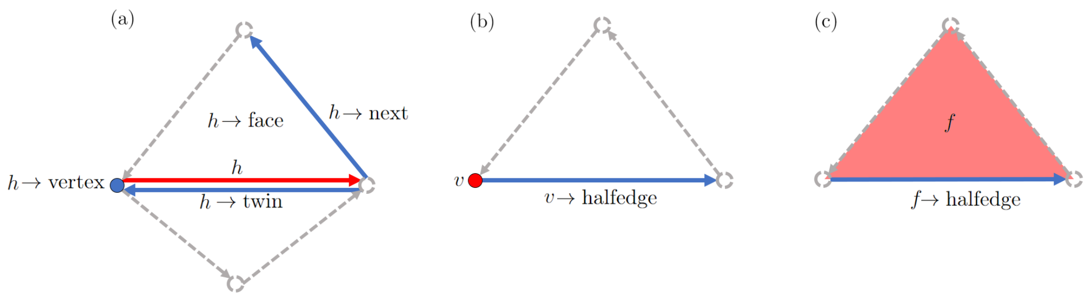

Figure 1: The local neighborhood of (a) an h Half-edge; (b) a v vertex; (c) an f face in Half-edge data structure.

Another important topological concept is the orientability of a surface (2-manifold). Surfaces of real-world objects are orientable, and we often utilize the orientability of a surface in data structures representing it. Well-known counterexamples are the Möbius strip and the Klein bottle, which are non-orientable surfaces. Instead of providing the precise definition of orientability, it suffices for us to recall the orientability of surfaces made up of polygonal faces. These surfaces are orientable if and only if there exists a consistent orientation of the normal vectors of the faces. In our minds, let us choose an orientation for each face, i.e., let us orient the edges of the face. This can be done in two ways for each face. The two different orientations of vertices define two different unit normals. It can be shown, the orientation of normals is consistent if and only if the orientation of the shared edge of two adjacent faces is opposite on these two faces. We call the surface orientable if there exists such a consistent orientation of the normal vectors. Summarizing the above, when designing a data structure for storing closed orientable surfaces with polygonal faces, we take advantage of the following.

- If a mesh is a compact 2-manifold, then each edge is shared by exactly two faces.
- If a mesh is orientable, the orientation of the shared edge can be opposite on the two adjacent faces.
- If we assume that a polyhedron with n vertices consists of only triangular faces and has a low genus, then it has approximately $2n$ faces and $3n$ vertices.

HALF-EDGE DATA STRUCTURE

The Half-edge data structure is suitable for storing orientable 2-manifolds (with or without boundary) defined by polygonal faces. As we will soon see, this representation effectively takes advantage of the properties mentioned earlier. Let us assume, that each edge is shared by exactly two faces (2-manifold), and the orientation of the common edge is opposite on these two faces (orientability). Due to this, each edge is split in two obtaining the so-called half-edges, the central elements of the data structure. For each half-edge h, we store the starting vertex ($h \rightarrow \mathrm{vertex}$), the associated face ($h \rightarrow \mathrm{face}$), its oppositely oriented pair ($h \rightarrow \mathrm{twin}$), and the next half-edge ($h \rightarrow \mathrm{next}$), as you can see on Fig. 1 (a). Faces and vertices store only a reference to a corresponding half-edge to which they are connected, see Fig. 1 (b) and (c). Most queries related to neighboring information, which may be required in a modeling task, can be performed in constant time in this data structure.

The storage cost can be estimated as follows. Let us suppose again that our polyhedron has n vertices, ℓ edges, and m triangles. The coordinates of vertices are represented by floating point numbers, while for each adjacency information, an integer is required to store the index of the referenced geometric entity. Since both integer and floating point numbers typically occupy 4 bytes of storage, we do not differentiate between their storage cost, it is sufficient to count the number of data needed to be stored per vertex. The actual storage requirement of the implementation can be obtained by multiplying the storage cost by 4 bytes. For Half-edge data structure, we have to store the following.

- Half-edges (vertex, face, next, twin): $2 \cdot \ell \cdot 4 = 8\ell$;
- Vertices (coordinates, half-edge): $n \cdot (3 + 1)$;
- Faces (half-edge): $m \cdot 1$.

The total storage cost of the data structure assuming $\chi \approx 0$:

$$8\ell + 4n + m \approx 8 \cdot 3n + 4n + 2n = 30n.$$

SOLIDMESH DATA STRUCTURE

The SolidMesh data structure was originally created for being able to perform attach and detach operations quickly, especially for real-time simulations where the user can break, slice, or glue objects together. It was clear that the vertex and index array required for rendering do not contain enough topological information to perform such tasks. On the other hand, we found that the Half-edge data structure is not suitable for the purpose either, because after attaching or detaching, additional costly operations have to be performed to create the new vertex and index arrays that define the new meshes. This is the reason why we developed our own data structure, which is enough close to the computer graphics representation of meshes, yet contains enough topological information to perform complex mesh operations. On the other hand, the SolidMesh implementation of attach and detach operations requires introducing additional advanced algebraic operations, which are beyond the scope of this article.

One of the main reasons for the popularity of edge-based data structures is perhaps that, the part of the surface can be described by a fixed number of geometric entities per edge (vertex, edge, face). If we store a fixed amount of data for each edge, there is no need to use dynamic arrays or dynamic lists within the class describing the edges. Therefore, when designing a new, face-based data structure, it is crucial to store a fixed amount of data for each face (and vertex), similarly in the case of edge-based data structures. When designing our data structure, we formulated the following requirements.

1. The representation should based on the vertex and index arrays used by the GPU.
2. Edges should not be explicitly represented.
3. A fixed amount of data should be stored for faces and vertices.
4. Global operations should be performed quickly.

Condition 1. and 2. imply, that the central elements of our data structure are necessarily faces. Condition 3. can not be fulfilled, unless each face has the same number of vertices. Since every polygonal face can be decomposed into triangles, we choose triangular faces.

The efficient execution of local operations occurring during modeling tasks was not a crucial consideration. We would like to prepare the data structure for global operations where the entire vertex and index arrays need to be traversed (detach, cut, split, smooth, subdivide, etc.). The Half-edge data structure allows efficient execution of local modifications, geometric information of a neighborhood of an edge is encapsulated into half-edges. In the implementation of our data structure, we did not create a new face or vertex class, which would achieve similar encapsulation. Instead, we added some extra (one- and multi-dimensional) arrays containing all the necessary geometric information to the vertex and index arrays. The following section describes how our data structure is created, which is suitable for storing and manipulating orientable compact 2-manifolds defined by triangular faces. It is important to note that the surface of a solid geometry is a connected orientable compact 2-manifold, and conversely, each non-self-intersecting connected orientable compact 2-manifold defines a solid. Our data structure is designed specifically for storing and manipulating the surfaces of such solid geometries, where we extensively utilize the previously discussed topological properties. Therefore, we will refer to this representation as SolidMesh in the subsequent discussions.

Maybe the simplest approach is to define the SolidMesh data structure using functions with finite domains. Let us suppose, that $I = \{0, \ldots, n-1\}$, $J = \{0, \ldots, m-1\}$ and $\tau = \{0, 1, 2\}$. Then the common representation of a mesh in computer graphics is a (V, T) pair, where the V vertex array and the T index array can be defined by the following functions: $V : I \to \mathbb{R}^3$ and $T : J \times \tau \to I$. The I, J sets refer to the indices of the vertices and triangles, the τ set is responsible for storing the order of vertices within a triangle. τ is similar to the ring of integers modulo 3, we defined the addition for any $k \in \mathbb{Z}$ as

$$\forall j \in \tau \ \ \forall k \in \mathbb{Z} \ : \ j \oplus k := (j + k) \mod 3.$$

We found, that if we define our data structure with the following functions, we achieve a similarly strong topological descriptive capability as in the case of the Half-edge structure.

- $V : I \to \mathbb{R}^3$: the vertex coordinates. $V(i) \in \mathbb{R}^3$ defines the position of the i-th vertex.
- $T : J \times \tau \to I^3$: the index triplets of the faces. Let us suppose that the j-th triangle of our surface is spanned by the $V(p), V(q), V(r)$ vertices $(p, q, r \in I)$. This fact can be formulated as follows:

  $$T(j, 0) = p \ \wedge \ T(j, 1) = q \ \wedge \ T(j, 2) = r.$$

  The edges are determined implicitly by T. We will assume that the k-th edge of the j-th triangle is a directed edge from $V(T(j,k))$ to $V(T(j, k \oplus 1))$ for $k = 0, 1, 2$.
- $A : J \times \tau \to J$: the triangle adjacents of the triangles. $A(j, k) = p$ if and only if the j-th triangle is adjacent to the p-th triangle, and their shared edge is the k-th edge of the j-th triangle.
- $\alpha : J \times \tau \to \tau$: the edge index of the adjacent triangle. $\alpha(j, k) = q$ if and only if the shared edge between triangles j and $A(j, k)$ is the q-th edge of the adjacent ($A(j, k)$-th) triangle. Now consider an edge from $T(j, k)$-th vertex to $T(j, k \oplus 1)$-th vertex. By definition, the $\alpha(j, k)$-th edge of $A(j, k)$-th triangle is the same edge between the two adjacent triangles. Since the orientation of the shared edge is opposite, one edge's starting vertex coincides with the other edge's end vertex, and vice versa, i.e.

  $$T(j, k) = T(A(j, k), \ \alpha(j, k) \oplus 1), \ \text{ and } \ T(j, k \oplus 1) = T(A(j, k), \alpha(j, k)).$$

- $B : I \to J$: face index for a vertex. $B(i) = j$ means that the $V(i)$ vertex is a vertex of the j-th triangle.
- $\beta : I \to \tau$ vertex index for a vertex. $\beta(i) = q$ means that the $V(i)$ vertex is the q-th vertex of the $B(i)$-th triangle, i. e.

  $$T(B(i), \beta(i)) = i.$$

- $d : I \to \mathbb{N}$: degree of vertices. The $V(i)$ vertex has exactly $d(i)$ vertex neihgbors.

Formally, our mesh representation is an $(V, T, A, \alpha, B, \beta, d)$ tuple. Since (V, T) is the common representation of a surface in computer graphics, V and T can be considered as the vertex array and the index array, which we pass to the GPU for rendering. A visualization of domain and range sets of the functions defining a SolidMesh data structure can be seen in Fig. 3. Most of the functions refer to the topology of the mesh, V is the only function that stores geometric information. In Fig. 2 you can see an illustration of the connectivity information

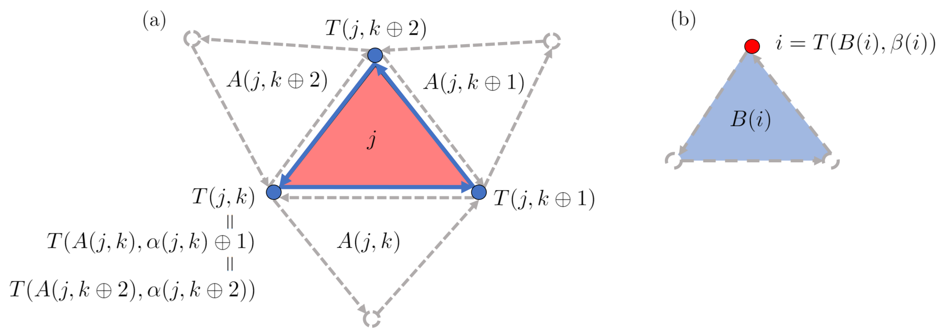

Figure 2: The local neighborhood of (a) the j-th triangle; (b) the i-th vertex in SolidMesh data structure.

Geometry

Topology

$$\begin{tikzcd}
& \mathbb{N} & J & \\
\mathbb{R}^3 & I \arrow[l, "V"'] \arrow[u, "d"] \arrow[ur, "B"] \arrow[dr, "\beta"'] & & J \times \tau \arrow[ll, "T"'] \arrow[ul, "A"'] \arrow[dl, "\alpha"] \\
& & \tau &
\end{tikzcd}$$

Figure 3: Visualization of the of $V, T, A, \alpha, B, \beta, d$ function with their domain and range sets.

stored in our data structure.

The question arises as to why exactly this additional information is stored for individual vertices/triangles. There may be many other options, but we have found that this information is necessary in order to be able to answer each connectivity query efficiently. Let us take a closer look at Fig. 3. In the set I we store the indices of the vertices, and in J the indices of the triangles. Since we have identified the edges with the edges of triangles, the set $J \times \tau$ can correspond to the set of edges. When we define a mapping between two sets, it means that we reach from one geometric entity to another. Based on these, we can draw the following conclusions:

- T assigns vertices to edges/triangles,
- A assigns triangles to edges/triangles,
- A with α assigns edges to edges,
- B assigns triangles to vertices,
- B with β assigns edges to vertices.

As we can see, the α and β functions are not interesting in themselves, only as a supplement to the A and B functions. By using α and β, we are able to store a reference to a specific edge. Therefore we don't have to go around the edges of a triangle to find the vertex we are looking for. We store the degrees (d) for another convenience function. In this case, vertices/edges/faces adjacent to a vertex can be listed without checking whether we have returned to the starting vertex or not. So, if the data structure is broken for some reason, we will not get an infinite loop during the query.

The storage cost of this data structure can be calculated as follows.

- Vertices (coordinates, face index, vertex index, degree): $n \cdot (3 + 1 + 1 + 1)$;
- Faces (3 vertex index, 3 adjacent triangle index, 3 edge index): $m \cdot (3 + 3 + 3)$.

Total storage cost assuming $\chi \approx 0$:

$$6n + 9m \approx 6n + 9 \cdot 2n = 24n.$$

COMPARATIVE TESTS

In our experiments we used an efficient Half-edge data structure [14], and we implemented our proposed data structure in C# language using Unity engine [6]. We compared the runtime of the Half-edge and SolidMesh implementations for four different tasks. These tasks, which we will discuss in detail in the following section, were as follows:

1. Creation
2. Rendering
3. Laplacian smoothing
4. Loop subdivision

Each algorithm was run approximately 10 times, and the best runtime was recorded. The results are presented in Table 1. and 2. We used 8 different models as inputs for the algorithms, ranging from the minimum vertex count of 8 to the maximum of 15002 vertices. The genus of the models also varies; we examined models with genus 0, 1, and 2. According to this, their Euler characteristics are respectively 2, 0, and -2. These test meshes can be seen in Fig. 4.

Creation

In this case, we measured the time it takes to create the topological data structures. We read the models from Wavefront OBJ files and stored the information about vertices and triangle indices in appropriate `float` and `int` arrays. Then, by reading the vertex and index information from the temporary buffers, we created Half-edge and SolidMesh data structures. The time taken for their creation was recorded in Table 1.

The SolidMesh data structure was created using Algorithm 2 and 1. The algorithm begins with a preprocessing step, where edge information is collected for the first and only time. This information is stored in a dictionary (or hashmap). The algorithm iterates over all the triangles, and the consecutive vertices (edges) spanning the triangles are hashed as ordered pairs (χ_1 and χ_2). If the hashmap H does not yet contain a value at the χ_1 key, the pair $(i, j) \in J \times \tau$ is stored there; otherwise, it is stored at the χ_2 key. Note that since we are dealing with compact 2-manifolds, the completed hashmap H will store, at the χ_1 and χ_2 keys, the indices of triangles adjacent along the same edge and the index of their shared edge.

Formally, in the algorithm, the function h performs the hashing, mapping a pair of numbers from I^2 to a value in the key set K. The hashmap H then maps keys in K to elements in the $J \times \tau$ set. We can formulate the following

$$h : I^2 \to K, \quad H : K \to J \times \tau, \quad H(h(p, q)) = (j, k) \iff \{p, q\} = \{T(j, k), T(j, k \oplus 1)\}.$$

After preprocessing, the main algorithm follows, which constructs the SolidMesh data structure. To do this,

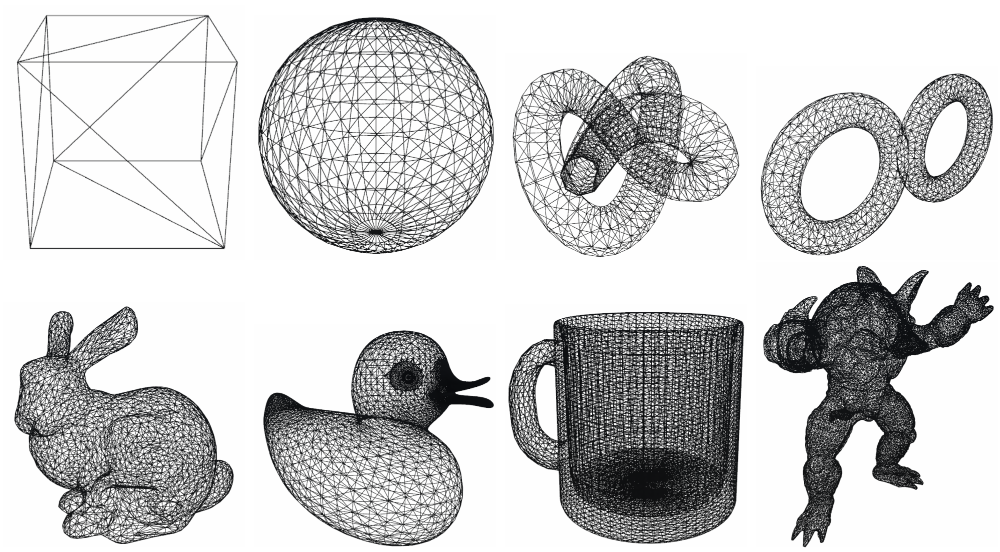

Figure 4: Test meshes with their vertex count (from left to right, top to bottom): Cube (8); Sphere (482); Torusknot (880); 2-tori (1156); Bunny (2503); Ducky (5084); Mug (6390); Armadillo (15002).

Algorithm 1 BuildEdgeDictionary(V, T)

```
Require: $V, T$ : vertex and triangle array
Ensure: $H$ dictionary (hashmap) containing edge-triangle adjacencies
 1: for $j = 0, \dots, n-1$ do
 2:     for $k = 0, 1, 2$ do
 3:         $\chi_1 := h(T(j,k), T(j, k \oplus 1))$
 4:         $\chi_2 := h(T(j, k \oplus 1), T(j,k))$
 5:         if $H(\chi_1) = \emptyset$ then
 6:             $H(\chi_1) := (j,k)$
 7:         else
 8:             $H(\chi_2) := (j,k)$
 9:         end if
10:     end for
11: end for
```

we need to iterate over the triangles again. The consecutive vertices of the triangles are hashed, but by this point, the hashmap H is already filled. Therefore, at the key χ_1, we either store the currently examined triangle or its edge-adjacent triangle. Consequently, we select the key that points to the neighboring triangle between χ_1 and χ_2, and use this to populate the values of the arrays A and α Then, we update the degree of each vertex, and the branch on line 13 is required solely to avoid double-counting the contribution of each edge to the vertex degree. Finally, in the arrays B and β, we store a reference for each vertex to a triangle containing it.

Algorithm 2 CreateSolidMesh(V,T)

```
Require: $V,T$ : vertex and triangle array
Ensure: $(V,T,A,\alpha,B,\beta,d)$ : SolidMesh representation of $(V,T)$ mesh
 1: $H$ :=BuildEdgeDictionary$(V,T)$
 2: for $j = 0,\dots,n-1$ do
 3:   for $k = 0,\dots,2$ do
 4:     $\chi_1 := h(T(j,k), T(j,k\oplus 1))$
 5:     $(j_1,k_1) := H(\chi_1)$
 6:     $\chi_2 := h(T(j,k\oplus 1), T(j,k))$
 7:     $(j_2,k_2) := H(\chi_2)$
 8:     if $j_1 = j$ then
 9:       $A(j,k),\ \alpha(j,k) := j_2,\ k_2$
10:     else
11:       $A(j,k),\ \alpha(j,k) := j_1,\ k_1$
12:     end if
13:     if $T(j,k) < T(j,k\oplus 1)$ then
14:       $d(T(j,k)) = d(T(j,k)) + 1$
15:       $d(T(j,k\oplus 1)) = d(T(j,k\oplus 1)) + 1$
16:     end if
17:     $B(T(j,k)) := j$
18:     $\beta(T(j,k)) := k$
19:   end for
20: end for
```

Rendering

In this test, we measured the time it takes for Unity to pass the information necessary for rendering a `Mesh` object [17], namely the positions of vertices and triangle indices using the `SetVertices` and the `SetIndices` functions. To be entirely precise, in this test, we did not measure the actual rendering time, but rather the time it takes to pass the data to Unity in a form that enables it to perform rendering.

Model name	Number of vertices	Creation time [ms]		Rendering time [ms]	
		Half-edge	SolidMesh	Half-edge	SolidMesh
Cube	8	0.04	0.04	0.06	**0.04**
Sphere	482	1.17	**1.02**	0.72	**0.09**
Torusknot	880	2.57	**2.02**	1.21	**0.10**
2-tori	1156	2.85	**2.72**	1.02	**0.11**
Bunny	2503	7.86	**6.32**	3.59	**0.27**
Ducky	5084	15.67	**12.95**	4.66	**0.30**
Mug	6390	18.39	**17.48**	5.43	**0.34**
Armadillo	15002	**52.97**	236.97	14.20	**0.84**

Table 1: Time costs of creation and render.

Laplacian smoothing

Mesh smoothing or denoising is a task aimed at reducing or eliminating noise present in vertex positions. Typically, this task is solved by applying some low-pass filter that suppresses high-frequency variations on the surface. The most commonly used procedures are iterative, but there are also single-step smoothing techniques, as described, for instance, in [7]. Further information on mesh smoothing algorithms and their comparison can be found in [3] and [1]. One of the most commonly used and simplest mesh smoothing algorithms is Laplacian smoothing. In this iterative process, the following steps are performed for each vertex.

1. Calculate the centroid of the vertex's neighboring vertices.
2. Replace the vertex position with the position of the centroid, as illustrated in Fig. 5.

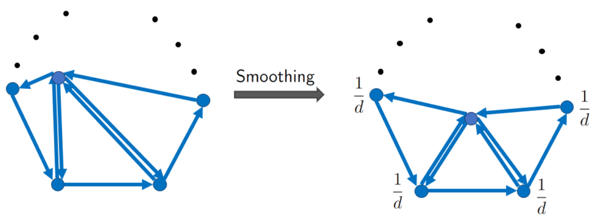

Figure 5: Laplacian smoothing: the position of the vertices is overwritten with the centroid of the neighboring vertices.

It is important to note that position updates are performed simultaneously for all vertices, and the triangulation of the surface remains unchanged throughout the operation. Therefore, during mesh smoothing procedures, only vertex positions are updated. However, since this must be done simultaneously, a temporary buffer must be created to store the new vertex positions. At the end of the smoothing step, the data stored in the buffer must be copied into the original topological data structures. The above step is repeated until the desired smoothness of the surface is achieved, emphasizing the importance of fast computation. Example iterations of the Laplacian smoothing procedure are shown in Fig. 6. In the comparison, we measured the runtime of a

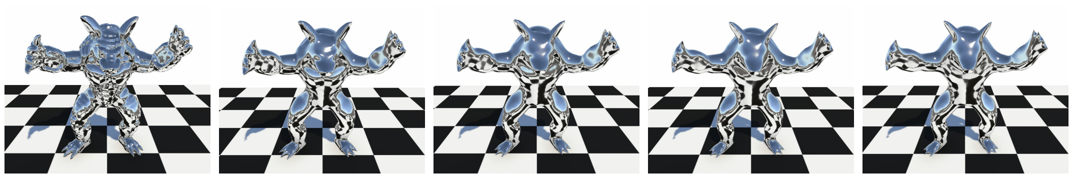

Figure 6: Results of Laplacian smoothing algorithm on the Armadillo model after $0, 4, 8, 12, 16$ iterations, respectively.

single step of Laplacian smoothing for models with different numbers of vertices. The critical topological query is the determination of neighboring vertices for each vertex. After that, the calculations can be performed quite simply and quickly.

Loop subdivision

A general subdivision algorithm takes a polyhedron as input. During subdivision, we divide the faces of the polyhedron into multiple faces and then reposition both the newly created and existing vertices according to a so-called subdivision scheme. The subdivision schemes are designed in such a way that repeated application converges the polyhedron to a (once or multiple times) differentiable surface. A brief summary of popular subdivision algorithms can be found in [8], and detailed studies on the topic are available in [15]. Loop subdivision is a global operation defined for a polyhedron defined by triangular faces [12]. In each step of the subdivision algorithm, the following operations are performed.

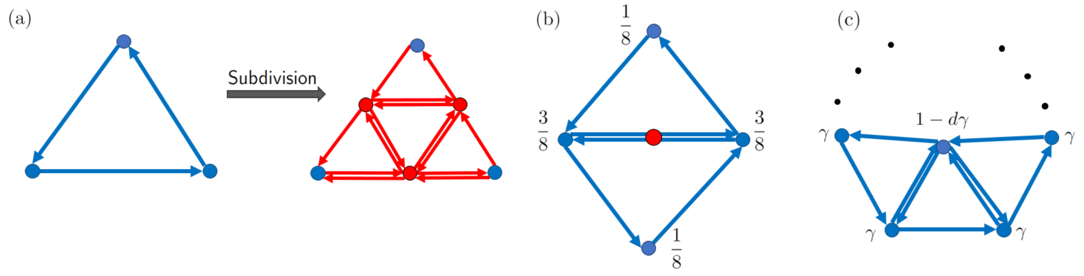

Figure 7: Loop-subdivision

1. Create a new vertex on each edge.
2. Connect the new vertices to split each triangle into 4 smaller triangles (see Fig. 7 (a)).
3. Calculate the positions of the new vertices as the barycentric combination of the two vertices spanning the edge and the third vertices of the triangles that are sharing the edge (see Fig. 7 (b)).
4. Recalculate the positions of the old vertices as the barycentric combination of the adjacent old vertices (see Fig. 7 (c)).

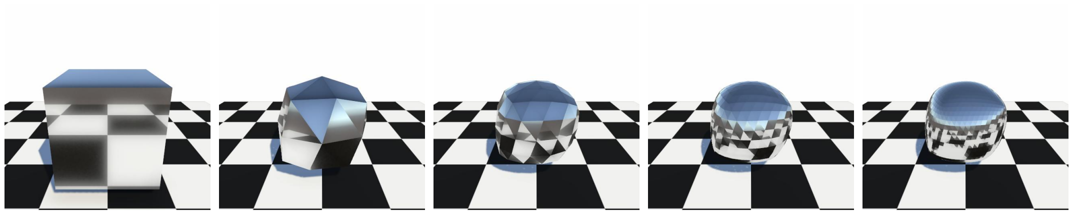

Figure 8: The first few steps of Loop-subdivision algorithm on a cube model.

An example of the results of the first few iterations of the subdivision scheme is shown in Fig. 8. As we can see, the subdivision operation is not trivial; we need to break down every face of the polyhedron and compute the coordinates of every vertex (new and old). By calculation of vertex positions the edge-adjacent triangle pairs play an important role, therefore the Half-edge data structure is often chosen for implementing this subdivision scheme. Unlike Laplacian smoothing, in this case, not only the positions of vertices but also the triangulation changes. We create 4 new triangles for each original triangle. Therefore, after subdivision, the entire data structure of both implementations needs to be replaced, utilizing the connectivity information from before the subdivision.

Model name	Number of vertices	Smoothing time [ms] Half-edge	Smoothing time [ms] SolidMesh	Subdivision time [ms] Half-edge	Subdivision time [ms] SolidMesh
Cube	8	0.02	**0.01**	0.10	**0.02**
Sphere	482	0.74	**0.16**	5.63	**0.79**
Torusknot	880	0.92	**0.26**	11.36	**1.46**
2-tori	1156	1.33	**0.35**	14.37	**2.05**
Bunny	2503	3.70	**0.72**	34.78	**4.81**
Ducky	5084	8.48	**1.49**	60.87	**8.83**
Mug	6390	9.05	**1.93**	71.99	**11.12**
Armadillo	15002	26.50	**4.50**	370.69	**31.74**

Table 2: Time costs of Laplacian smoothing and Loop subdivision.

CONCLUSIONS

Our results for creation and rendering are shown in Table 1., the smoothing and subdividing time costs can be seen in Table 2. In this section, let us summarize our findings regarding the comparative tests:

1. **Creation:** both implementations performed similarly in our tests, with SolidMesh proving slightly faster in all cases except for the Armadillo model. In this case, the creation of the SolidMesh data structure took significantly more time than constructing the Half-edge structure. We have not yet been able to explain this substantial efficiency drop, which deviates from the observed trend.

2. **Rendering:** this test is not entirely fair, as the SolidMesh data structure is specifically designed to explicitly contain vertex and index arrays for efficient rendering. However, it is an interesting question to see how much better the optimized data structure performs compared to the industry standard. In the case of the Half-edge data structure, both extracting vertex positions and determining the index triples of faces are more costly than in the SolidMesh representation. Therefore, we reasonably assumed that our own data structure would outperform in this test. Our assumption was confirmed, sending the data to the GPU required for the rendering is 10-15 times faster in SolidMesh than in Half-edge.

3. **Laplacian smoothing:** the SolidMesh implementation ran 3-5 times faster than the algorithm implemented in the Half-edge structure, see the left side of Fig. 9. Upon analyzing the results, we concluded that the speed increase does not arise from the difference in the number of operations but rather from the fact that in the SolidMesh data structure, there is no need to allocate temporary memory for storing neighboring vertices for each vertex. This is because all information is available in place due to not applying encapsulation during the storing of local connectivity relations.

4. **Loop subdivision:** the SolidMesh implementation ran approximately 7 times faster than the Half-edge implementation, see right side of Fig. 9. The increase in efficiency in this test may have a similar underlying reason as in the case of Laplacian smoothing. In the Half-edge implementation, eventually, a completely new data structure needs to be constructed, whereas, in the SolidMesh implementation, we can heavily rely on the data structure from before the subdivision. Reindexing of the individual faces and vertices with complete maintenance of neighborhood relations can be quickly performed by examining the subdivision scheme.

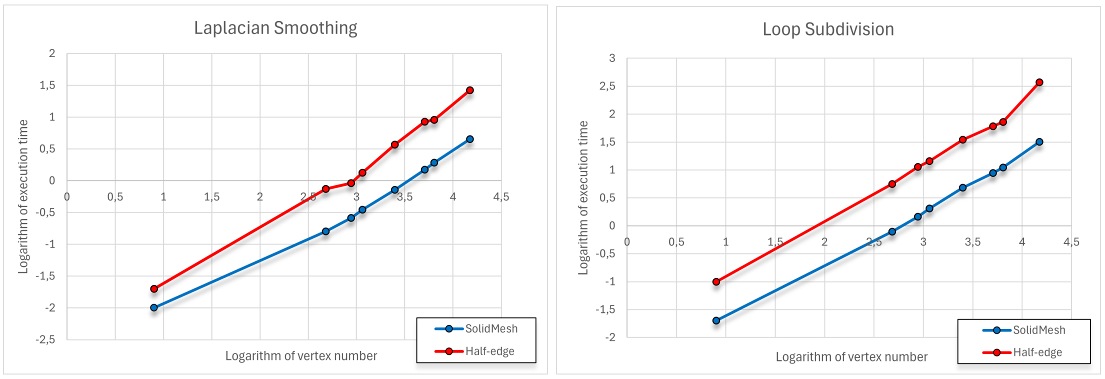

Figure 9: Results of Laplacian smoothing (left) and Loop subdivision (right) on a log-log graph.

In conclusion, the SolidMesh implementation generally showed better performance across various tasks, except for one single test. In the future, we plan the efficient implementation of attaching and detaching algorithms in the SolidMesh data structure. We hope that similar speed-ups can be observed in these cases as well. If our prediction turns out to be correct, our surface representation could play an important role in use cases where geometric objects can be cut, fractured, destructed, or glued together in real time.

ACKNOWLEDGEMENTS

Supported by the ÚNKP-23-4 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

Új Nemzeti Kiválóság Program MINISTRY OF CULTURE AND INNOVATION

Gábor Fábián, https://orcid.org/0000-0003-0255-5379

REFERENCES

[1] Bade, R.; Haase, J.; Preim, B.: Comparison of fundamental mesh smoothing algorithms for medical surface models. vol. 1, 289–304, 2006.

[2] Baumgart, B.G.: A polyhedron representation for computer vision. In National Computer Conference and Exposition (AFIPS '75), 589–596, 1975. http://doi.org/10.1145/1499949.1500071.

[3] Belyaev, A.; Ohtake, Y.: A comparison of mesh smoothing methods. Israel-Korea Bi-National Conference on Geometric Modeling and Computer Graphics, Tel Aviv University, 83-87 (2003), 2003.

[4] Botsch, M.; Steinberg, S.; Bischoff, S.; Kobbelt, L.: Openmesh - a generic and efficient polygon mesh data structure. https://www.graphics.rwth-aachen.de/media/papers/openmesh1.pdf, 2002.

[5] Campagna, S.; Kobbelt, L.; Seidel, H.P.: Directed edges – a scalable representation for triangle meshes. Journal of Graphics Tools, 3(4), 1–11, 1998. http://doi.org/10.1080/10867651.1998.10487494.

[6] Fabian, G.: SolidMesh. https://github.com/robagnaibaf/SolidMesh.

[7] Fábián, G.: Generalized Savitzky–Golay filter for smoothing triangular meshes. Computer Aided Geometric Design, 100, 102167, 2023. ISSN 0167-8396. http://doi.org/https://doi.org/10.1016/j.cagd.2022.102167.

[8] Farin, G.: Curves and Surfaces for CAGD: A Practical Guide. Computer graphics and geometric modeling. Elsevier Science, 2002. ISBN 9781558607378. https://books.google.hu/books?id=D0qGMAwSUkEC.

[9] Guibas, L.; Stolfi, J.: Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. ACM Transactions on Graphics, 4(2), 74–123, 1985. http://doi.org/10.1145/282918.282923.

[10] Kettner, L.: Halfedge data structures. https://doc.cgal.org/latest/HalfedgeDS/index.html, 2024.

[11] Lee, J.M.: Introduction to Topological Manifolds. Springer, 2000. http://doi.org/10.1137/1.9780898717761.

[12] Loop, C.T.: Smooth Subdivision Surfaces based on Triangles. Ph.D. thesis, University of Utah, 1987.

[13] Mantyla, M.: An Introduction to Solid Modeling. Computer Science Press, 1988. http://doi.org/10.1137/1.9780898717761.

[14] Müller, D.E.; Preparata, F.P.: Finding the intersection of two convex polyhedra. Theoretical Computer Science, 7(2), 217–236, 1978. http://doi.org/10.1016/0304-3975(78)90051-8.

[15] SIGGRAPH.: Course Notes: Subdivision for modeling and animation. Course Notes: SIGGRAPH 2000: 27th International Conference on Computer Graphics and Interactive Techniques, Conference 23-28 July, 2000, Exhibition 25-27 July, 2000, New Orleans. Association for Computing Machinery, 2000. https://books.google.hu/books?id=mxw-zgEACAAJ.

[16] Tobler, R.F.; Maierhofer, S.: A mesh data structure for rendering and subdivision. In Proceedings of the January 30 - February 3, 2006, Winter School of Computer Graphics, WSCG '2006. Plzen, Czech Republic, 2006.

[17] Unity Technologies: Mesh class. Unity Scripting API. https://docs.unity3d.com/ScriptReference/Mesh.html. Retrieved April 1, 2024.

[18] Weiler, K.: Edge-based data structures for solid modeling in curved-surface environments. IEEE Computer Graphics and Applications, 5(1), 21–40, 1985. http://doi.org/10.1109/MCG.1985.276271.