Simplification of Hexagonal Meshes in a Concise Representation

João Kongevold, Diego Brandão, Renato Mauro

[1]School of Informatics and Computer Science
Federal Center for Technological Education Celso Suckow da Fonseca (CEFET/RJ)
Rio de Janeiro – RJ – Brazil

joao.kongevold@eic.cefet-rj.br
diego.brandao@cefet-rj.br
renato.mauro@cefet-rj.br

Abstract. *Adaptive hexagonal meshes avoid hanging nodes and keep consistency without hierarchical structures. A recent concise, exact, face-only representation for these meshes, indexed by dyadic lattice coordinates, encodes each face's type and refinement scale in just 8 bits—unlike the multiple references per face required by classical topological data structures. However, its merge operation, responsible for mesh simplification, was left unimplemented due to a representation limitation. This work closes that gap by implementing merge while preserving the simplicity of join, validating correctness through a load test spanning ten refinement levels and up to 10,000 operations per level, along with a throughput and memory analysis.*

1. Introduction

Computational meshes are fundamental structures for discretizing continuous domains into finite geometric elements, with applications in computer graphics (such as modeling, texturing, and rendering) and numerical methods, such as the Finite Element Method (FEM). Mesh quality and the efficiency of its representation directly affect the accuracy of the results obtained and the computational cost of these applications [Figueiredo 2025]. Formally, a mesh is composed of a set of vertices, a set of edges, and a set of faces, which together describe the geometry and topology of the discretized domain; fully reconstructing the mesh and its adjacency relations requires storing enough information about these three sets and how they relate to each other.

Among mesh types, triangular meshes are widely used for their simplicity, but present limitations in adaptive generation: both 1-to-4 triangular subdivision and quadtree-based approaches produce *hanging nodes* – vertices that arise when the edge of a refined element ends in the interior of an unrefined neighbor's edge, breaking the exact sharing of vertices and edges between adjacent elements. This problem is common in local adaptive refinement and typically requires propagating refinement to neighboring elements or applying additional correction techniques, increasing algorithmic complexity and potentially degrading the quality of the generated elements [Sussner et al. 2005].

Adaptive hexagonal meshes emerge as an advantageous alternative in this context: Sußner et al. [Sussner et al. 2005] propose a hexagonal subdivision scheme that avoids T-nodes and hanging nodes, keeping the mesh consistent at all times without requiring hierarchical structures – an idea later extended to point-cloud triangulation

[Sußner and Greiner 2009] and to generating quadrilateral meshes with angle guarantees [Liang and Zhang 2011], detailed further in Section 2.

Despite these qualities, classical mesh representations, such as half-edge [Botsch et al. 2010] and winged-edge [Baumgart 1972], explicitly store vertices, edges, faces, and their adjacency relations, consuming a significant amount of memory. De Figueiredo [Figueiredo 2025] proposes a concise, exact representation for adaptive hexagonal meshes based solely on faces: each face is described by its position in *lattice* coordinates, exact in floating point, and by its type and scale, the latter two encoded in just 8 bits, against the more than four references per face required by classical topological representations. However, that work leaves the implementation of the *merge* operation, responsible for mesh simplification, as future work, pointing to a limitation of the representation itself.

This paper closes that gap by presenting a merge algorithm that operates directly on the concise face-based representation, without altering its anchor convention. We validate the implementation through round-trip load tests across ten refinement levels, confirming mesh consistency and characterizing merge's throughput and memory footprint relative to split.

2. Related Work

This section presents the works whose ideas are necessary for understanding and motivating the development of this work.

The *quadtree* data structure [Samet 1984] is geared toward adaptive meshes. Starting from a root quadrilateral, it performs recursive subdivisions into four sub-quadrants wherever greater detail is needed; each node represents a face at a given refinement level, and the effective mesh corresponds to the faces stored in the leaf nodes. This hierarchy facilitates refinement and neighbor queries across different scales, but can introduce hanging nodes at the boundaries between regions of unequal refinement, requiring additional handling to preserve mesh consistency.

Red-Green Triangulation, proposed by Bank, Sherman, and Weiser [Bank and Sherman 1999], targets triangular meshes. *Red* refinement performs the regular 1-to-4 subdivision, connecting the midpoints of a triangle's three edges. When applied to only some triangles, irregular vertices (T-nodes) appear at the boundaries between refined and unrefined regions. To eliminate them, *green* refinement is applied, subdividing the affected neighboring triangles by connecting one vertex to the midpoint of the opposite edge, producing two triangles. This step restores mesh consistency, but the resulting triangles may have less regular angles than those from red refinement [Sussner et al. 2005].

Sußner et al. [Sussner et al. 2005] propose a hexagonal subdivision scheme for adaptive, interactive-LOD height-map rendering, defining the *split*, *join*, and *merge* operations. The regular geometry of hexagons avoids T-nodes and hanging nodes, keeps the mesh consistent at all times, and dispenses with hierarchical structures, reducing memory consumption compared to triangular and quadtree-based approaches. Sußner and Greiner [Sußner and Greiner 2009] adapt this scheme to triangulate point clouds in the plane, with guarantees on the shape of the resulting triangles. Liang and Zhang

[Liang and Zhang 2011] use it to generate quadrilateral meshes with angles bounded to $[60° - \varepsilon,\ 120° + \varepsilon]$ and free of hanging nodes, advantageous for FEM applications. Finally, de Figueiredo [Figueiredo 2025] presents a face-based concise representation for Sußner et al.'s hexagonal meshes, further reducing the memory required, but without implementing the *merge* operation.

3. Hexagonal Meshes and a Concise Representation

A mesh of regular hexagons offers advantages for covering a domain: it leaves no gaps, since all hexagons tile perfectly, and it can be refined locally without needing to rebuild the entire mesh. An adaptive hexagonal mesh was proposed by [Sussner et al. 2005], with the operations of subdivision (*split*), union (*join*), and simplification (*merge*), keeping the mesh always consistent and free of hanging nodes. To represent this mesh efficiently, [Figueiredo 2025] proposes a concise, exact representation based solely on faces, dispensing with the explicit storage of vertices, edges, or adjacency relations: each face is described by its position in lattice coordinates, together with its type and scale, these last two encoded in just 8 bits – against the more than four references per face required by classical topological representations.

3.1. Face-Based Representation

An adaptive hexagonal mesh has only two types of face, at different scales: full hexagons and semi-hexagons, trapezoids obtained by bisecting a hexagon along a diameter. The base mesh contains only hexagons; semi-hexagons arise as a product of subdivision. Each face is represented by three attributes [Figueiredo 2025]: the **type**, an integer from 0 to 6 indicating the semi-hexagon's orientation (0 to 5) or a full hexagon (6); the **anchor**, which gives the face's position and is defined as the center of the full hexagon or the midpoint of the semi-hexagon's longest edge — which coincides with the center of the hexagon that would contain it; and the **scale** s, the number of subdivisions already applied, such that the longest edge of a face at scale s measures $L_s = 2^{-s} L_0$, where L_0 is the edge length in the base mesh. All faces are stored in a hash table F, indexed by the face's anchor in lattice coordinates, dispensing with explicit storage of vertices and adjacency relations.

3.2. Lattice Coordinates and Mesh Reconstruction

The Cartesian coordinates of the anchors involve $\sqrt{3}$ and are not exact in floating point. To work around this, note that the base mesh's anchors form a regular triangular mesh, and so can be written uniquely as $au_1 + bu_2$, with $u_1 = \left(\frac{3}{2},\ \frac{\sqrt{3}}{2}\right)$, $\qquad u_2 = (0, \sqrt{3})$, and represented by integer coordinates $[a, b]$, called *lattice* coordinates (or offset coordinates). Since subdivision always places new anchors at midpoints, after successive refinements the lattice coordinates remain dyadic fractions, exact in floating point.

Navigation across the mesh is done via the operator $A(c, s, k) = c + 2^{-s} c_k$, where c_k is the lattice vector connecting the origin to the center of its neighbor in direction k (mod 6), at scale 0. $A(c, s, k)$ thus returns the anchor of the face adjacent to the hexagon centered at c, at scale s, in direction k.

Each face's vertices are recovered from a standard hexagon, at scale 0 and centered at the origin, used as a template. For a hexagonal face with anchor c and scale s, each vertex v' is given by $v' = c + 2^{-s} v$, where v is the corresponding vertex of the template.

For a semi-hexagon of type t, the formula is $v' = c + 2^{-s}\, v - A(0,\, s+1,\, t)$, where the last term corrects for the fact that the standard semi-hexagon's anchor is not at the origin.

3.3. *Split* and *Join*

The *split* operation shrinks a hexagon to half its size, keeping its center fixed, and fills the freed-up space with six semi-hexagons, one per orientation; the new vertices are computed as the arithmetic mean between the original hexagon's vertices and its center. The *join* operation merges two adjacent semi-hexagons that share their longest edge into a single hexagon, removing that shared edge; the resulting hexagon can be subdivided again.

In the concise representation, splitting a face (c, s) is done by incrementing the scale stored in $F[c]$ and creating six new semi-hexagons, one per orientation $k \in \{0, \ldots, 5\}$, with anchor $A(c, s+1, k)$ and the same scale s as the original hexagon. Join happens automatically: when inserting a semi-hexagon, the algorithm checks whether an entry already exists in the hash table at that position; if so, a semi-hexagon already shares the longest edge with the new face, and it suffices to change the existing entry's type to 6 (full hexagon). This way, no adjacency relation needs to be updated, since none is explicitly stored.

3.4. *Merge*

While join is an implicit, local byproduct of split, involving only the two semi-hexagons that come to share a longest edge, *merge* is an explicit operation, requested on a full hexagon: it is the inverse of split and gathers a hexagon and the six surrounding semi-hexagons into a single larger hexagon. To guarantee mesh consistency, every hexagon carries a split counter, incremented on each split and decremented on each merge; the operation can only be applied to a hexagon whose split counter is greater than zero. Additionally, the six neighbors must be checked: if all of them are semi-hexagons, merge can be performed directly; if a neighbor is a full hexagon with a split counter of zero, it must first be split into two semi-hexagons (undoing a previous join) for merge to become possible.

This operation was not implemented in the concise representation of [Figueiredo 2025], which leaves it as future work due to a limitation of the representation: since a semi-hexagon's anchor is chosen as the center of the hexagon that would contain it, two semi-hexagons sharing the longest edge necessarily have the same anchor and therefore cannot coexist in the hash table — the join operation always merges them immediately into a single full hexagon. Merge, on the other hand, would require undoing a join, recreating two semi-hexagons with the same anchor, which the representation, as defined, cannot distinguish. [Figueiredo 2025] points out that this limitation could be worked around by anchoring semi-hexagons at their barycenters — which can also be expressed as dyadic lattice coordinates — leaving the investigation of mesh simplification in this representation as future work.

4. Merge Implementation

This section presents an implementation of the merge operation, the inverse of split, described in Algorithm 1.

The algorithm is split into two loops. The first only checks, without modifying F, whether the operation can be applied: for each of the six neighboring positions $h_k =$

$A(c, s, k)$, it checks whether h_k corresponds to a semi-hexagon not yet touched by another split (type k and scale $s - 1$) or to a full hexagon resulting from a previous join with that neighbor (type 6 and scale s); any other situation — a missing position, an incompatible type, or an unexpected scale — indicates that the neighborhood was altered by another operation, and the algorithm returns false immediately, with no modification to F. Only if all six positions pass is the second loop executed, removing the semi-hexagons from the first case, reverting the hexagons from the second case back to semi-hexagons, and decrementing the scale of c — exactly undoing the split and join operations.

Algorithm 1 *Merge*(c)

Input: Full hexagon centered at c
Output: True if the merge is performed; false otherwise
$s \leftarrow F[c].s$
if $F[c].t \neq 6$ **or** $s < 1$ **then**
  return False
end if
for $k \leftarrow 0$ **to** 5 **do**
  $h \leftarrow A(c, s, k)$
  if $h \notin F$ **then**
    return False
  end if
  $face \leftarrow F[h]$
  if not $((face.t = k$ **and** $face.s = s - 1)$ **or** $(face.t = 6$ **and** $face.s = s))$ **then**
    return False
  end if
end for
for $k \leftarrow 0$ **to** 5 **do**
  $h \leftarrow A(c, s, k)$
  $face \leftarrow F[h]$
  if $face.t = k$ **then**
    Remove h from F
  else
    $face.t \leftarrow (k + 3) \bmod 6$
    $face.s \leftarrow face.s - 1$
  end if
end for
$F[c].s \leftarrow F[c].s - 1$
return True

5. Experimental Evaluation and Results

This section experimentally evaluates the proposed implementation, verifying the correctness of merge and comparing its performance and memory consumption against split (Section 5.2), and assessing whether execution time remains approximately constant as the mesh grows (Section 5.3). Source code, tests, and the full README with implementation details of the consistency checker and memory measurements are available GitHub Repository.

5.1. Experimental Environment

All experiments were run on a computer with an Intel Core i5-11400H processor (2.70 GHz), 16 GB of RAM, and Windows 11 Home Single Language (64-bit), using Python with NumPy 2.4.3, Matplotlib 3.10.8, and Pympler.

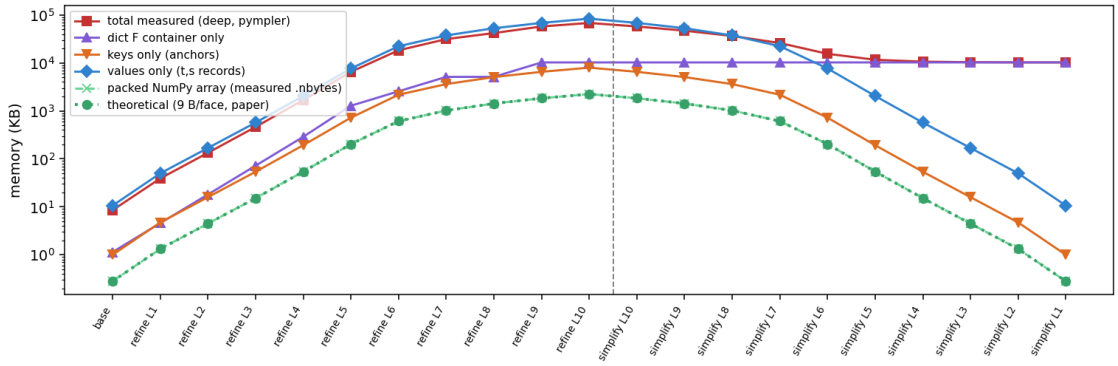


Figure 1. Memory consumption during the round-trip load test: theoretical estimate, compact NumPy representation, and the Python dictionary implementation (total, hash table, keys, and values).

5.2. Round-trip Load Test

Starting from the base mesh, up to 10,000 split calls are performed at each refinement level, recording the subdivided faces; upon reaching the highest level, those same faces are merged in reverse order, progressively reconstructing the initial mesh. At each level, a geometric checker confirms that the mesh remains watertight — free of overlaps, holes, or hanging nodes — and records the representation's theoretical memory footprint (9 bytes per face: 64 bits for the anchor and 8 bits for type and scale, following de Figueiredo), the actual memory used by the Python implementation, and that of an equivalent compact representation in NumPy; at the end, it is also verified whether the reconstructed mesh is identical to the initial one. Implementation details of the checker and the memory measurements are described in the repository accompanying this work.

Unlike our empirical measurements, de Figueiredo's [Figueiredo 2025] evaluation was combinatorial, reporting only a theoretical references-per-face ratio (4.69–6.00); our results confirm empirically that the representation's theoretical footprint (9 bytes/face) holds in practice, aside from Python's own object and hash-table overhead.

The mesh remained consistent throughout every run and every step, and at the end of each run the reconstructed mesh was identical to the base mesh. The average throughput, pooled over 5 independent runs, was 35,973 ops/s for split and 36,956 ops/s for merge. Figure 1 shows the evolution of memory consumption over a full cycle.

5.3. Operation Time as a Function of Mesh Size

To verify whether split time remains approximately constant, validating the expected $O(1)$ complexity of the hash table used to store faces, up to 200 hexagons are refined per level directly with `subdivide()`, without `simplify()`, repeating the experiment 5 times from distinct meshes and pseudo-random seeds. Table 1 shows the mean and standard deviation of the time per operation (μs/op) and the number of subdivisions (*ops*) performed at each level.

6. Conclusion

This work presented an implementation of the merge operation for de Figueiredo's concise representation of adaptive hexagonal meshes, enabling both refinement and simplifi-

Table 1. Operation time as a function of mesh size (mean $\pm$ standard deviation over multiple runs).

Level	Faces before	Ops	Mean μs/op	Std μs/op
1	72	72	7.87	2.54
2	323	200	7.98	2.40
3	1092	200	9.11	1.69
4	2085	200	10.02	4.05
5	3033	200	9.45	3.74
6	3970	200	8.26	0.94
7	4920	200	9.32	1.91
8	5844	200	9.25	1.94
9	6792	200	8.65	1.94

cation of the mesh using the same data structure.

The experiments confirm that merge preserves mesh consistency: across the five runs of the load test, a refinement cycle up to ten levels, with up to 10,000 operations per level, followed by full simplification, no geometric or topological inconsistency was detected, and the reconstructed mesh was always identical to the base mesh. The performance of split and merge proved equivalent (35,973 ops/s versus 36,956 ops/s, a difference under 3%), indicating that the inverse operation does not compromise the structure's efficiency; the average time per operation also remained stable as the mesh grew nearly a hundredfold, corroborating the $O(1)$-cost hypothesis for the hash table used to store faces.

The memory evaluation, in turn, showed that the Python implementation costs significantly more than the theoretical limit of 9 bytes per face, due to object overhead and the hash table, whose internal capacity is not fully released after removals; an equivalent representation using NumPy structured arrays reaches exactly the theoretical 9 bytes, making it a natural candidate for a future reimplementation, provided the hash table is replaced by an equivalent indexing mechanism.

As future work, a complete formal proof of the correctness of the merge operation remains open, covering both the cases in which it should be applied and those in which it must correctly refuse to modify the mesh.

References

Bank, R. and Sherman, A. (1999). Some refinement algorithms and data structures for regular local mesh refinement. *Applications of Mathematics and Computing to the Physical Sciences*.

Baumgart, B. G. (1972). Winged edge polyhedron representation. Technical Report CS-TR-72-320, Stanford University.

Botsch, M., Kobbelt, L., Pauly, M., Alliez, P., and Lévy, B. (2010). *Polygon Mesh Processing*. A K Peters/CRC Press.

Figueiredo, L. (2025). A concise representation for adaptive hexagonal meshes. *Journal of the Brazilian Computer Society*, 31:373–381.

Liang, X. and Zhang, Y. (2011). Hexagon-based all-quadrilateral mesh generation with guaranteed angle bounds. *Computer Methods in Applied Mechanics and Engineering*, 200(23):2005–2020.

Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260.

Sussner, G., Dachsbacher, C., and Greiner, G. (2005). Hexagonal lod for interactive terrain rendering.

Sußner, G. and Greiner, G. (2009). Hexagonal delaunay triangulation. In Clark, B. W., editor, *Proceedings of the 18th International Meshing Roundtable*, pages 519–538, Berlin, Heidelberg. Springer Berlin Heidelberg.