Journal of the Brazilian Computer Society, 202X, XX:1, doi: 10.5753/jbcs.202X.XXXXXX
This work is licensed under a Creative Commons Attribution 4.0 International License.

A concise representation for adaptive hexagonal meshes

Luiz Henrique de Figueiredo [**IMPA** | ***lhf@impa.br***]

IMPA, Rio de Janeiro, Brazil

Received: 15 November 2024 • **Accepted:** 6 May 2025 • **Published:** DD Month YYYY

Abstract Adaptive hexagonal meshes yield high-quality planar quadrilateral meshes, which are desirable in applications. We propose a concise exact representation for adaptive hexagonal meshes that is based solely on the faces. We represent each face by its position, type, orientation, and scale. Our representation is simple to use and requires a small fraction of the memory required by topological data structures. Although no other topological elements or relations are explicitly represented, the mesh and all its topological relations can be reconstructed in linear time.

Keywords: Mesh representation, mesh generation, geometric modeling, data structures

1 Introduction

Polygonal meshes play a key role in modeling, texturing, and physically-based simulations. The accuracy and cost of the numerical computations in these applications depend directly on the quality of those meshes. While triangle meshes are ubiquitous for their simplicity and flexibility, high-quality quadrilateral meshes produce significantly better results.

Generating quadrilateral meshes is considerably more complicated than generating unstructured triangular meshes. The simplest method uses a Coons patch, obtained by bilinear or transfinite interpolation of the boundary curves of a curvilinear rectangle [Frey and George, 2008, Chapter 3]. This technique will give a quadrilateral mesh for a general planar domain once it has been decomposed into curvilinear quadrilaterals, but finding suitable decompositions is already a nontrivial task. Generating unstructured quadrilateral meshes is an even harder task, especially when aiming for high-quality adaptive meshes [Bern and Eppstein, 2000; Eppstein, 2014].

Sußner *et al*. [2005] proposed a subdivision scheme for creating planar adaptive hexagonal meshes (see Figure 1), from which high-quality quadrilateral meshes can be easily extracted. In this paper, we propose a concise and exact representation for those meshes that is based solely on the faces; no other topological elements or relations are explicitly represented. We represent the faces by their centers, which are given exact coordinates as dyadic fractions. We attach to each face two tiny integers that define its type, orientation, and scale. This data takes 8 bits per face and is sufficient to reconstruct the entire mesh and all topological relations in linear time. Our representation is simple to use and requires a small fraction of the memory required by topological data structures. Code supporting our representation is publicly available [de Figueiredo, 2024a].

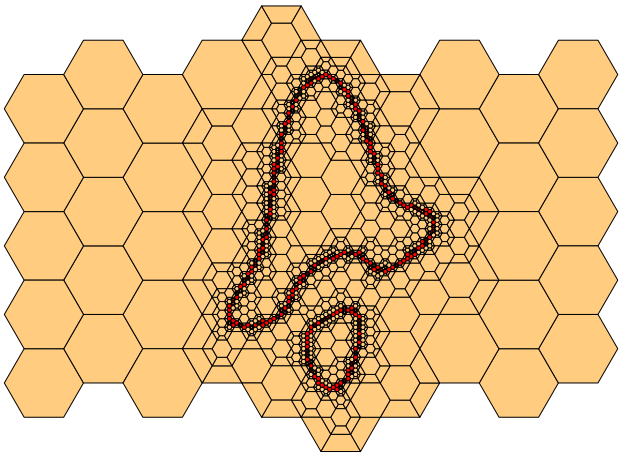


Figure 1. An adaptive hexagonal mesh refined around an implicit curve. The mesh contains regular hexagons and semi-hexagons at different scales; it can be converted to a pure quadrilateral mesh by bisecting hexagons.

2 Related work

Middleton and Sivaswamy [2005] explained the advantages of hierarchical hexagonal meshes in image processing. Sahr *et al*. [2003] discussed multi-resolution hexagonal meshes and their advantages for representing global geography. Sußner *et al*. [2005] proposed a scheme for refining regular hexagonal meshes adaptively and applied it to interactive level-of-detail terrain rendering. Sußner and Greiner [2009] adapted that scheme for triangulating point clouds in the plane using a dual triangular mesh with guaranteed bounds on the shape of triangles. Liang and Zhang [2011] used adaptive hexagonal meshes for generating high-quality quadrilateral meshes for regions bounded by cubic splines.

Virtually all previous work on mesh representations targets general meshes or specializes to triangle meshes [De Floriani and Hui, 2007]. Representations tailored for quadrilateral meshes are rare; the ones we have found target compression [King *et al*., 2000; Diaz *et al*., 2010]. We know of no representations tailored for adaptive hexagonal meshes.

Nevertheless, concise exact representations for planar meshes with rigid geometry and topology have been explored recently. Soto Sánchez *et al*. [2021] described a vertex-centric representation for periodic tilings of the plane by regular polygons, which inspired a vertex-centric representation for adaptive diamond-kite meshes [de Figueiredo, 2024b], a family of high-quality quadrilateral meshes introduced by Eppstein [2014]. The representation we describe here continues this trend.

3 Adaptive hexagonal meshes

We summarize here the main concepts of the adaptive hexagonal meshes proposed by Sußner *et al*. [2005]. While the geometry of those meshes is simple, the refinement process is interesting and deserves a detailed description.

Base mesh. An adaptive hexagonal mesh starts with a *base mesh* of congruent regular hexagons. The base mesh may have multiple components and holes. For concreteness, and without loss of generality, we focus here on the *standard base mesh* shown in Figure 2; it has edges of unit length and horizontal (flat-topped) hexagons. That base mesh was used to produce the refined mesh in Figure 1. Suitable base meshes for applications are typically obtained by applying a similarity transformation to the standard base mesh. Sußner and Greiner [2009] and Liang and Zhang [2011] used a base mesh with a single hexagon containing the input point cloud.

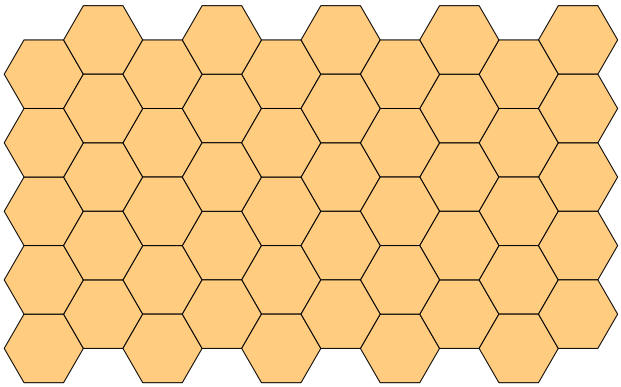


Figure 2. The base mesh is formed by congruent regular hexagons.

Faces. Adaptive hexagonal meshes have only two kinds of faces, at different scales: regular hexagons and semi-hexagons, which are trapezoids obtained by bisecting a regular hexagon along a diameter. The excellent aspect ratio of the faces accounts for the high quality of the meshes. The base mesh has only hexagons; semi-hexagons appear during refinement, as a product of the local subdivision operation described below.

Subdivision. The local subdivision operation replaces a hexagon by a smaller hexagon and six semi-hexagons around it, as illustrated in Figure 3. The six new vertices are the midpoints of the segments joining the center of the hexagon and an old vertex. The mesh is unaffected outside the original hexagon, except perhaps for local merging, which we describe next.

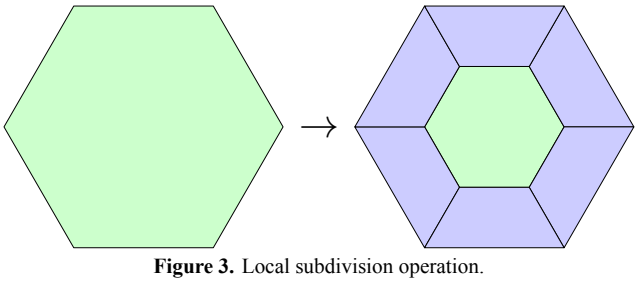


Figure 3. Local subdivision operation.

Merge. A key feature of adaptive hexagonal meshes is that whenever two semi-hexagons are adjacent along their longest edge, they are merged into a single hexagon, as illustrated in Figure 4. This happens as follows: Consider two adjacent hexagons (a). At one point, the bottom hexagon is subdivided (b). Later, the top hexagon is subdivided (c). This gives two semi-hexagons that are adjacent along their longest edge, which then are merged into a full hexagon (d). The mesh is unaffected outside the two original hexagons.

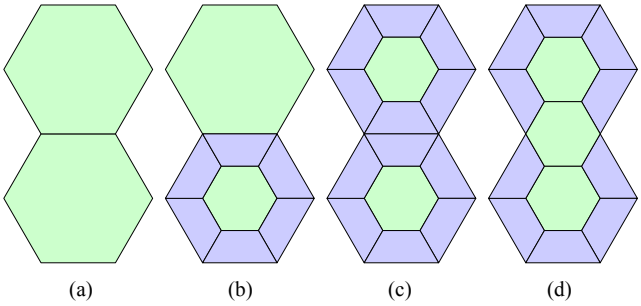


Figure 4. Local merge operation.

Refinement. Adaptive hexagonal meshes are refined by recursively subdividing hexagons selected by criteria defined by the application. To achieve adaptivity, applications need to subdivide semi-hexagons too, but this can only be done indirectly, by subdividing hexagons. Here is a typical scheme, which we have adopted.

Let S be a semi-hexagon that we want to subdivide and let T be the face adjacent to S along the longest edge of S; we call T the *mate* of S. In the simple case, the mate T is a hexagon, as in Figure 5 (a). Then we subdivide T using the local subdivision operation (b). This triggers a local merge that produces a new hexagon H containing S (c). Then we subdivide H, thus finally subdividing S (d).

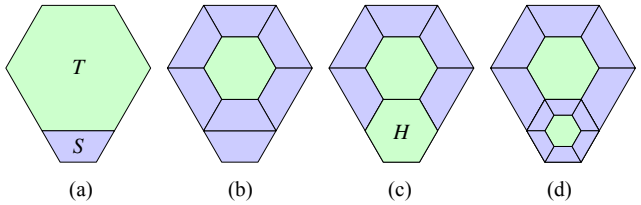


Figure 5. Refining semi-hexagons: the simple case.

In the general case, the mate T may be a semi-hexagon, and so we recursively subdivide T and then subdivide the last hexagon containing S. Figure 6 illustrates this process: S is a semi-hexagon whose mate T is also a semi-hexagon (a). The mate of T is a hexagon, which is subdivided (b), triggering a merge that yields a smaller hexagon which becomes the new mate of S. This hexagon is subdivided (c), triggering another merge that leads to a middle hexagon containing S, which is subdivided, thus finally subdividing S (d).

If during this recursive process we reach a semi-hexagon that has no mate, then that semi-hexagon is a boundary face and its longest edge is on the boundary. We handle this case by subdividing the face as in Figure 7 [Sußner *et al*., 2005] or by creating a full hexagon as its mate, thus extending the mesh, a solution we have adopted in Figure 1 (see also §6).

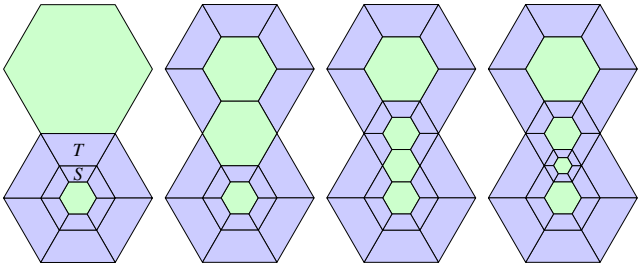


Figure 6. Refining semi-hexagons recursively.

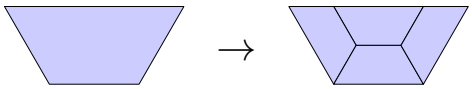


Figure 7. Refining semi-hexagons: the boundary case.

4 Our representation

Our representation for adaptive hexagonal meshes is based on the faces. We represent each face by its position, type, orientation, and scale. We start by discussing why this choice is natural and indeed possible. Then we describe the details of our representation.

Background. General planar meshes are typically represented using a topological data structure which stores the vertices, edges, and faces of the mesh and their adjacency relationships [De Floriani and Hui, 2007]. In complete contrast, planar meshes with rigid geometry and topology should admit concise exact representations that exploit the rigidity. Rigidity here means that there are only a few possibilities for the geometry of the faces and for the stars of the vertices (the star of a vertex is the configuration of vertices around the vertex). Examples of rigid meshes include periodic tilings of the plane by regular polygons and diamond-kite meshes; both admit concise exact vertex-centric representations.

Vertex-centric representations for tilings of the plane by regular polygons are possible because the vertices adjacent to a given vertex are found at unit distance along fixed directions; it only remains to give exact coordinates to the vertices [Soto Sánchez *et al*., 2021]. Vertex-centric representations for adaptive diamond-kite meshes are natural because their local subdivision operates on vertices; they are possible because the vertex stars are restricted to a very small set [de Figueiredo, 2024b]; it only remains to give exact coordinates to the vertices and orientation and scale to their stars.

Adaptive hexagonal meshes are planar meshes with rigid geometry and topology and so should admit concise exact representations. While we could base a representation on vertex stars, our representation is based solely on the faces because the local subdivision operates on faces. Moreover, the faces have rigid geometry and fit together in simple ways. We represent each face by its position, type, orientation, and scale. This data suffices to reconstruct the face (see §5). No other topological elements or relations are explicitly represented; they can be reconstructed easily. Here are the details.

Type and orientation. Except for scale, there are only two types of faces: hexagons, which are always horizontal, and

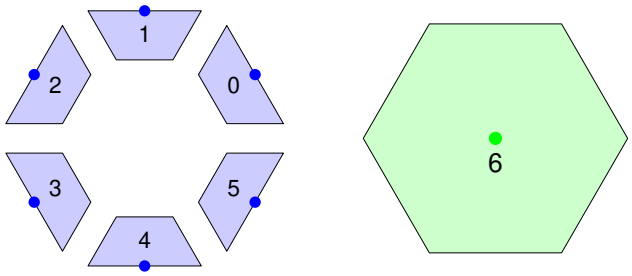


Figure 8. Face types and orientations, and anchor points.

semi-hexagons, which have one of six orientations. Thus, the type and orientation of a face is given by a tiny integer: 0 to 5 for semi-hexagons and 6 for hexagons, as in Figure 8. All faces of the base mesh have type 6. The type of a semi-hexagon changes to 6 after a local merge; the type of hexagons never change during refinement.

Scale. The local subdivision operation creates new edges that are half as long as the original edges (see Figure 3). Therefore, during refinement, edges are repeatedly halved. The scale of a face is an integer that reflects the length of its longest edge: in a face of scale s, the longest edge has length $L_s = 2^{-s}L_0$, where L_0 is the length of the edges in the base mesh. All edges in a face of scale s have length either L_s or $L_{s+1} = 2^{-1}L_s$. All faces of the base mesh have scale 0. The scale of the faces increase as the mesh is refined.

Position. The position of a face is given by a fixed *anchor point* in it, which we choose to be the center of the enclosing hexagon; for semi-hexagons, this center is also the midpoint of the longest edge (see Figure 8). This choice simplifies several tasks, including merging, as we shall see in §5. We give exact coordinates to face anchors as follows.

Lattice coordinates. The Cartesian coordinates of face anchors are not exact in floating-point because they involve $\sqrt{3}$. The key observation to give exact coordinates to all face anchors in an adaptive hexagonal mesh is that the anchors in the base mesh form a regular triangular mesh (see Figure 9) and so can be given *integer coordinates* in terms of *basic vectors* (see Figure 10). We call these coordinates *lattice coordinates* [Soto Sánchez *et al*., 2021; de Figueiredo, 2024b].

Integer coordinate schemes for hexagonal grids are well known in several applications, like game development and image processing; in those contexts, our lattice coordinates are commonly known as offset coordinates [Patel, 2021].

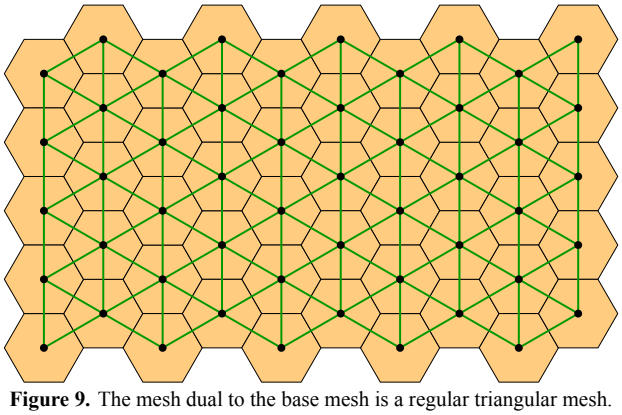


Figure 9. The mesh dual to the base mesh is a regular triangular mesh.

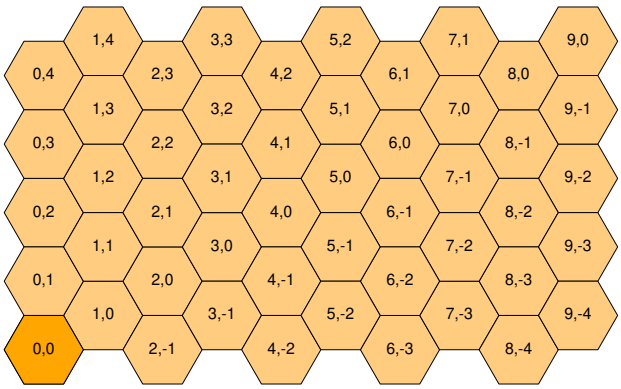


Figure 10. Lattice coordinates for face anchors in the base mesh.

There are many choices for two basic vectors spanning the triangular mesh dual to the base mesh. For concreteness, we choose the natural ones shown in Figure 11 (left): $u_1 = (\frac{3}{2}, \frac{\sqrt{3}}{2})$ and $u_2 = (0, \sqrt{3})$. Then, every face anchor in the base mesh can be written uniquely as $au_1 + bu_2$ and so be given integer lattice coordinates $[a, b]$ (see Figure 10). (As in previous work [Soto Sánchez *et al*., 2021; de Figueiredo, 2024b], we denote lattice coordinates by $[a, b]$ to differentiate them from Cartesian coordinates (x, y).) Figure 11 (right) shows the lattice coordinates of the face anchors around the origin; they will play a central role in §5.

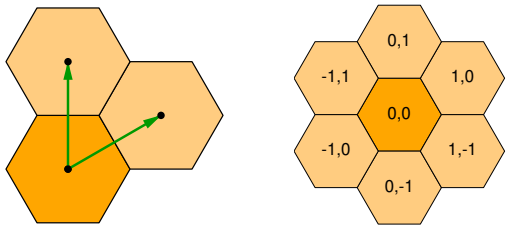


Figure 11. Basic vectors (left) and face anchors around the origin (right).

The semi-hexagons created during a local subdivision operation are anchored at the midpoints between the centers of two adjacent hexagons (see Figure 8). Therefore, after repeated refinement, all face anchors have *rational* lattice coordinates given by dyadic fractions and so can be represented exactly in standard floating-point form. Single precision allows 24 refinement levels, sufficient for most applications.

Summary. Our representation for adaptive hexagonal meshes is a set of records $\langle a, b, t, s\rangle$, one for each face of the mesh, giving their position by the dyadic lattice coordinates $[a, b]$ of their anchors, their type and orientation t, and their scale s. This is an abstract description. We discuss next concrete realizations of our representation.

5 Using the representation

We now describe how our representation supports in practice the key tasks of creating the base mesh, updating the representation after refinement, and reconstructing the mesh and its adjacency relations. The appendix contains code for these operations and some comments on it. Complete code supporting our representation is publicly available [de Figueiredo, 2024a].

Concrete representations. The abstract representation we described in §4 is made concrete and effective by internal and external representations. An *internal representation* is a suitable data structure that supports the abstract representation. As in previous work [Soto Sánchez *et al*., 2021; de Figueiredo, 2024b], we use the *cloud*, a hash table that stores faces indexed by the lattice coordinates of their anchors. An *external representation* is a file that stores the data required to rebuild an internal representation. Our representation is simple enough to be saved in a text file, one face per line; standard CSV files are convenient for this role. External representations record the *complete* data required to rebuild an internal representation: no post-processing is needed because our representation needs no topological relations and can infer them in expected constant time per face (see below). In contrast, standard data exchange formats (such as STL, OBJ, and OFF) require extensive post-processing to rebuild a topological data structure [McMains *et al*., 2001].

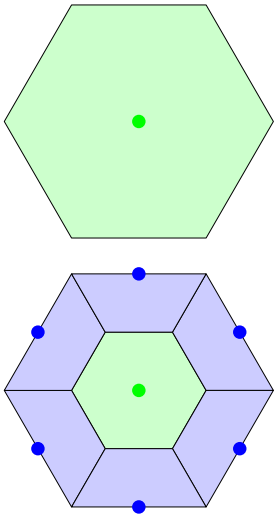
Base mesh. To make a grid of hexagons for the base mesh, just note that if c is the center of a hexagon, then $c + u_1$ and $c + u_2$ are the centers of two adjacent hexagons (see Figure 11). Therefore, if the lattice coordinates of c are $[a, b]$, then the lattice coordinates of those centers are $[a+1, b]$ and $[a, b+1]$. A double loop based on these observations creates a grid of hexagons like the one in Figure 2. All faces in the base mesh have type 6 and scale 0.

A key primitive. A central role is played by the set of face centers around the origin at scale 0. We denote by c_k the center at orientation k. Figure 11 (right) shows their lattice coordinates. A key primitive is $A(c, s, k) = c + 2^{-s}c_k$, which computes an anchor opposite to c at scale s in direction k (mod 6). These operations are performed exactly in dyadic lattice coordinates using standard binary floating-point arithmetic, because multiplying by a power of 2 is exact and fast.

Updating the representation. Consider a local subdivision operation on a hexagon anchored at c and at scale s. To update the representation to reflect the subdivision, first we create six new semi-hexagons, one for each orientation, at scale s, the same as the hexagon. The anchor of the semi-hexagon at orientation k is given by $A(c, s + 1, k)$. Then, we increment the scale of the hexagon to reflect its shrinking (see Figure 12). We follow a similar strategy to refine a semi-hexagon at the boundary, as in Figure 7: we just need to give the proper type to the new semi-hexagons; this takes into account the type of the original semi-hexagon.

Whenever we are about to create a new semi-hexagon, we query the cloud whether a face at the same position already exists. If so, then a local merge operation occurs, and we simply change the type of the existing face to 6, thus merging two semi-hexagons into a hexagon.

The simplicity of the merge operation in our representation is the reason for choosing the anchors of semi-hexagons as the centers of the enclosing hexagons: there is no need to change the center during a merge. Also noteworthy is the simplicity of the update: there are no adjacency relations to be fixed, because none are stored in our representation.



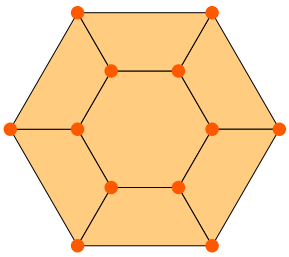
a	b	t	s
0.0	0.0	6	0

a	b	t	s
1	1	0.5	0.5
−1	2	−0.5	1
−2	1	−1	0.5
−1	−1	−0.5	−0.5
1	−2	0.5	−1
2	−1	1	−0.5

Figure 12. Updating the representation after a local subdivision operation. We increment the scale of the hexagon and create six new semi-hexagons, one for each orientation according to Figure 8, at the scale of the original hexagon (0 in the picture).

Reconstructing the mesh. While the vertices of the mesh are implicit in our representation, applications need them to render the mesh, to convert it to other formats, and also to evaluate criteria for adaptive refinement. For the mesh in Figure 1, we tested the signs of a function at the vertices of the mesh to locate the implicit curve (see also §6).

We exploit the rigid geometry of the faces in adaptive hexagonal meshes to recover the vertices of each face from our representation. We use the standard hexagon at the origin and at scale 0 and its subdivision as *templates* for all faces: we scale and translate a template face to match a given face. Figure 13 gives the lattice coordinates of the vertices in the standard face templates. The outer vertices are the vertices of the standard hexagon; they are the barycenters of the centers of their neighboring faces. The inner vertices are the midpoints of the segments joining the origin and an outer vertex. Therefore, all these vertices have lattice coordinates of the form $\frac{1}{3}[a, b]$, where a and b are dyadic fractions.



outer		inner	
$3a$	$3b$	$3a$	$3b$
1	1	0.5	0.5
−1	2	−0.5	1
−2	1	−1	0.5
−1	−1	−0.5	−0.5
1	−2	0.5	−1
2	−1	1	−0.5

Figure 13. Standard face templates and vertex coordinates scaled by 3.

We find the coordinates of any vertex in the mesh by scaling and translating a standard vertex. Consider a hexagonal face anchored at c and at scale s. If v is a vertex of the standard hexagon, then the corresponding vertex in the face is $v' = c + 2^{-s}v$. Consider now a semi-hexagonal face anchored at c and having type t and scale s. If v is a vertex of the standard semi-hexagon of type t, then the corresponding vertex in the face is $v' = c + 2^{-s}v - A(0, s + 1, t)$. (The last term corrects the fact that the anchor of a standard semi-hexagon is not at the origin.) Finally, the Cartesian coordinates (x, y) of a point with lattice coordinates $[a, b]$ are given by $x = \frac{3}{2}a$ and $y = (\frac{1}{2}a + b)\sqrt{3}$. Combining all this information gives the Cartesian coordinates of any vertex in the mesh.

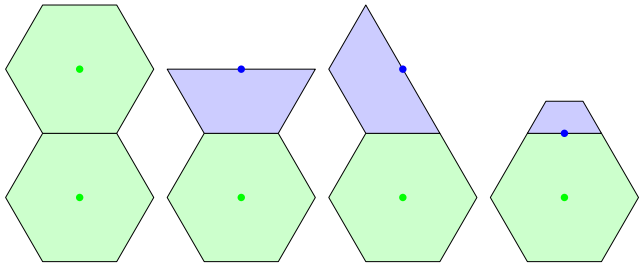


Figure 14. Faces adjacent to a hexagon.

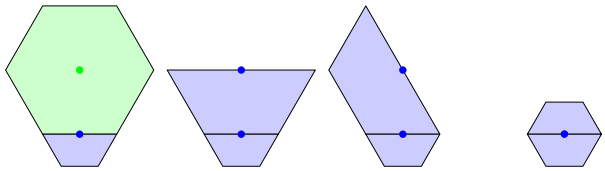


Figure 15. Faces adjacent to a semi-hexagon along its longest edge.

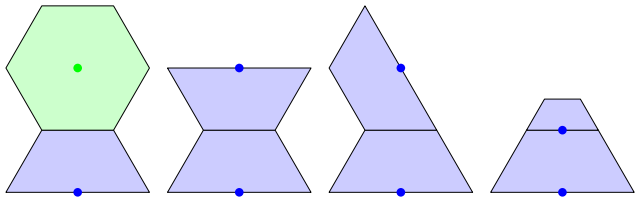


Figure 16. Faces adjacent to a semi-hexagon opposite its longest edge.

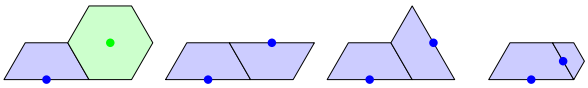


Figure 17. Faces adjacent to a semi-hexagon near its longest edge.

Table 1. Anchors of adjacent faces.

type	edge	candidate 1	candidate 2	figure
hexagon	k	$A(c, s+1, k)$	$A(c, s, k)$	14
semi-hexagon	0	$A(c, s+1, t)$		15
semi-hexagon	1	$A(c, s+2, t+2)$	$A(c, s+1, t+2)$	17
semi-hexagon	2	$A(c, s+2, t+3)$	$A(c, s+1, t+3)$	16
semi-hexagon	3	$A(c, s+2, t+4)$	$A(c, s+1, t+4)$	17

Finding adjacent faces. Finding the faces adjacent to a given face is a basic step in creating a topological data structure. Moreover, finding the face adjacent to a semi-hexagon along its longest edge (the mate) is a key task in the refinement scheme described in §3.

Consider a face anchored at c and having type t and scale s. To find the faces adjacent to that face, we find their anchors. Figures 14–17 show all possible cases and Table 1 gives the anchors of adjacent faces. Except for one case, there are two candidates; we test their presence in the cloud in that order (that is, the nearest one first). If neither anchor is present, then there is no adjacent face and the face is at the boundary. Edge 0 in a semi-hexagon is its longest edge (see Figure 15); the last case does not occur because it triggers a local merge. Figure 18 shows a dual mesh computed with these relations.

Vertex data. Our representation does not represent vertices explicitly. If the application needs vertex data during refinement, it needs to compute this data on the fly. This is exactly

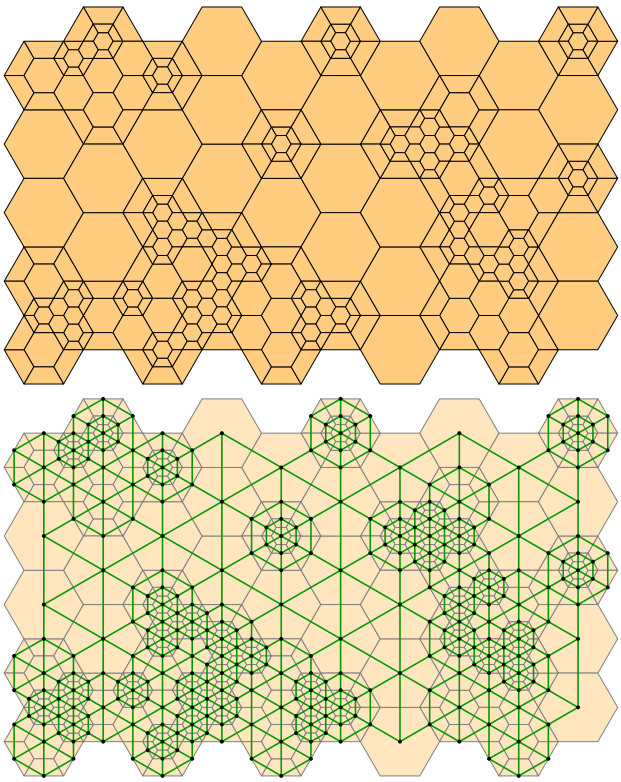


Figure 18. A randomly refined mesh (top) and its dual mesh (bottom).

what is done in the case study discussed in §6: the implicit function is evaluated at each vertex of a face, perhaps repeatedly. If computing vertex data is not possible or acceptable, applications that store data in vertices can still benefit from our representation. To avoid data duplication, create vertices as topological entities by storing them in a separate hash table. As we have seen, every vertex has exact coordinates of the form $\frac{1}{3}[a, b]$, where a and b are dyadic fractions. Then index the vertex hash table by lattice coordinates $[a, b]$. (Remember to divide by 3 when converting those coordinates to Cartesian coordinates.) This scheme provides direct access to vertex data that is shared among faces without explicitly linking vertices and faces.

Hashing. The cloud plays a central role in using our representation. The performance of subdivision, merge, and other primitive mesh operations rely on the performance of queries to the cloud. Therefore, we assume implicitly that the cloud is an efficient hash table that answers queries in expected constant time and manages memory with good occupancy and low bookkeeping. Except for these general requirements, the details of hashing are unimportant. Our proof-of-concept implementation [de Figueiredo, 2024a] uses a standard Python dictionary for the cloud because we can use complex numbers to represent face anchors and to index dictionaries. Naturally, we trust that Python dictionaries are well implemented in the sense described above. We have found no reason to use custom hash tables in our code. Nevertheless, custom hash tables may perform better [McMains *et al*., 2001, §4].

Simplification. Adaptive hexagonal meshes support simplification to coarser meshes because local subdivision and merge are reversible operations. Sußner et al. [2005; 2009] describe a level structure attached to faces that supports simplification by reversing these operations on suitable face configurations. While face scales in our representation can probably replace levels, we cannot reverse face merges because two semi-hexagons sharing their longest edge have the same anchor and so cannot coexist. While this is a limitation of our representation, it is not a crucial one: we could change the anchors of semi-hexagons to their barycenters; it turns out they are also expressed by dyadic lattice coordinates. We will leave the investigation of mesh simplification in our representation as future work.

6 Case study

We present here a brief case study on how our representation of adaptive hexagonal meshes behaves as a function of the refinement level. The task is to approximate the boundary of a region of interest, which for simplicity we model as a plane curve given implicitly by $f(x, y) = 0$. We use the standard criterion and select a face for refinement when the values of f at the vertices of the face have different signs [Suffern, 1990]. The implicit curve is one used by Taubin [1994]. The base mesh is the one in Figure 2. Figure 19 shows the meshes at refinement levels 0 to 3; Figure 1 shows level 4.

The simplest way to represent the topology of a mesh is a list of faces and their vertices, as in standard mesh formats like OBJ and OFF. For an adaptive hexagonal mesh this representation takes $4F_4 + 6F_6$ references, where F_4 is the number of semi-hexagons in the mesh and F_6 is the number of hexagons. (A *reference* is either a pointer or an integer that indexes an array; it typically takes 4 bytes.)

Table 2 reports some statistics for our task, going down to refinement level 6. It shows that the size of the mesh (measured by the number of vertices, edges, and faces) doubles at each refinement level. The last row shows that the simple representation described above takes over 4 references per face in our task. Including face adjacency relations doubles the number of references. Standard topological data structures include more adjacency relations and so use many more references per face [De Floriani and Hui, 2007]. In contrast, our representation uses just 8 bits or 0.25 references per face: 3 bits for the type and 5 bits for the scale because the refinement level is most probably less than 32 (actually, less than 24 if we use single precision for vertex coordinates).

The topological data for a mesh needs to be complemented with geometric data. Standard mesh formats use a list of vertices which are referenced in the list of faces by ids. Then, the geometric data takes 64 bits per vertex in single precision. Our representation uses 64 bits per face for the lattice coordinates of the anchors. In our case study, the number of faces is about 75% the number of vertices. We expect a similar memory reduction holds in other applications.

Table 2. Statistics for implicit curve.

level	0	1	2	3	4	5	6
vertices	130	360	734	1538	3092	6446	13016
edges	179	580	1223	2627	5354	11192	22666
faces	50	221	490	1090	2263	4747	9651
F_4	0	112	276	672	1464	3078	6316
F_6	50	109	214	418	799	1669	3335
refs/face	6.00	4.99	4.87	4.77	4.71	4.70	4.69

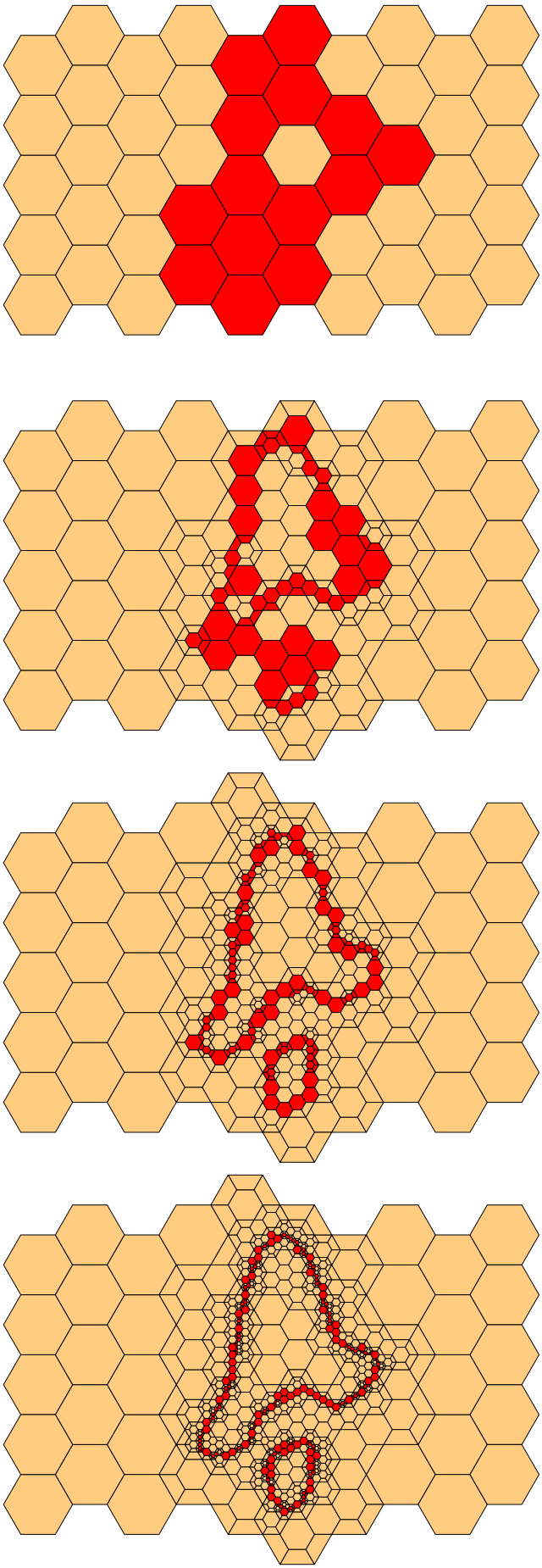


Figure 19. Meshes for implicit curve refined to levels 0 to 3.

The strategy of creating a hexagon as the mate of semi-hexagons at the boundary mentioned in §3 is reminiscent of the continuation method of Dobkin *et al*. [1990], which traverses a regular triangular grid that is never fully built. Figure 20 shows an approximation computed from a base mesh containing a single face (shown in green). That face is subdivided and mates are created for the two semi-hexagons that cross the curve. Then the mesh is extended automatically to cover the connected component that crosses the base face without creating a large base mesh as in Figure 2. Our representation supports all this effortlessly without intervention.

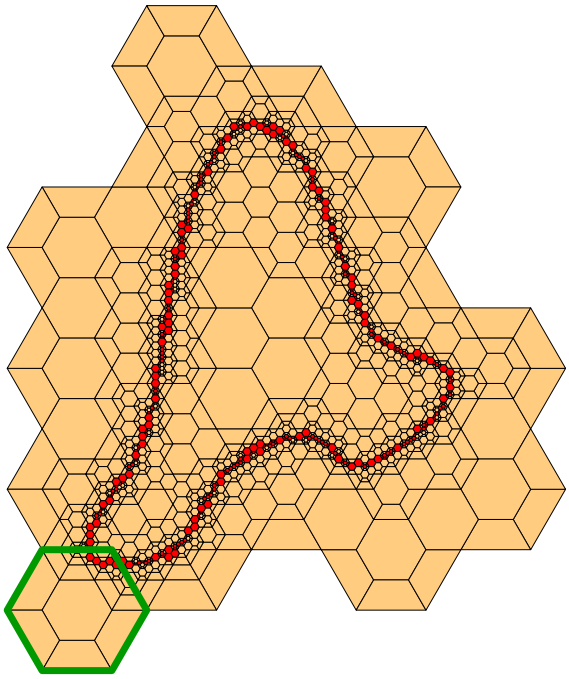


Figure 20. Approximating an implicit curve by continuation. The base mesh contains a single face (shown in green). During refinement, the mesh grows automatically to cover one connected component of the curve.

7 Conclusion

Adaptive hexagonal meshes can generate high-quality quadrilateral meshes for applications like geometric modeling for graphics, finite-element analysis in computational mechanics, and planar meshes for computational geography. Our representation allows the use of adaptive hexagonal meshes with little memory overhead and fast reconstruction algorithms. The representation can be stored in simple text files, such as CSV files. More importantly, it is surprisingly simple to perform both subdivision and merge in our representation: it is just a matter of updating types and scales; no geometric searches or complex geometric computations are necessary, and there are no adjacency relations to be fixed, because none are stored in our representation. The key to this simplicity is that at each moment we know exactly which face anchors to locate and that the cloud can do this in expected constant time. We hope that our simple representation can lead to a wider use of adaptive hexagonal meshes. A good starting point would be public implementations of the previous applications [Sußner *et al*., 2005; Sußner and Greiner, 2009; Liang and Zhang, 2011].

Declarations

Acknowledgements

This research was done in the Visgraf Computer Graphics laboratory at IMPA in Rio de Janeiro, Brazil. Visgraf is supported by the funding agencies FINEP, CNPq, and FAPERJ, and also by gifts from IBM Brasil, Microsoft, and NVIDIA.

Competing interests

The author declares that he has no competing interests.

Availability of data and materials

A proof-of-concept implementation of the representation described in this paper is freely available at GitHub [de Figueiredo, 2024a].

References

Bern, M. and Eppstein, D. (2000). Quadrilateral meshing by circle packing. *International Journal of Computational Geometry & Applications*, 10(4):347–360. DOI: https://doi.org/10.1142/S0218195900000206.

de Figueiredo, L. H. (2024a). A concise representation for adaptive hexagonal meshes. Code at github.com/lhf/hex.

de Figueiredo, L. H. (2024b). A vertex-centric representation for adaptive diamond-kite meshes. *Computers & Graphics*, 119:103910. DOI: https://doi.org/10.1016/j.cag.2024.103910.

De Floriani, L. and Hui, A. (2007). Shape representations based on simplicial and cell complexes. In *Eurographics 2007 State of the Art Reports*, pages 63–87. Eurographics. DOI: https://dx.doi.org/10.2312/egst.20071055.

Diaz, R., Dreux, M., Lopes, H., and Lewiner, T. (2010). A simple compression of tri-quad meshes with handles. In *Full Papers Proceedings of WSCG 2010*, pages 205–212.

Dobkin, D. P., Wilks, A. R., Levy, S. V. F., and Thurston, W. P. (1990). Contour tracing by piecewise linear approximations. *ACM Transactions on Graphics*, 9(4):389–423. DOI: https://doi.org/10.1145/88560.88575.

Eppstein, D. (2014). Diamond-kite adaptive quadrilateral meshing. *Engineering with Computers*, 30(2):223–235. DOI: https://doi.org/10.1007/s00366-013-0327-9.

Frey, P. J. and George, P. (2008). *Mesh Generation: Application to Finite Elements*. Wiley.

King, D., Rossignac, J., and Szymczak, A. (2000). Connectivity compression for irregular quadrilateral meshes. DOI: https://doi.org/10.48550/arXiv.cs/0005005.

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. DOI: https://doi.org/10.1016/j.cma.2011.03.002.

McMains, S., Hellerstein, J. M., and Séquin, C. H. (2001). Out-of-core build of a topological data structure from polygon soup. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, pages 171–182. DOI: https://doi.org/10.1145/376957.376977.

Middleton, L. and Sivaswamy, J. (2005). *Hexagonal image processing: a practical approach*. Springer.

Patel, A. (2021). Hexagonal grids. Web page at https://www.redblobgames.com/grids/hexagons/. Accessed on 6 May 2025.

Sahr, K., White, D., and Kimerling, A. J. (2003). Geodesic discrete global grid systems. *Cartography and Geographic Information Science*, 30(2):121–134. DOI: https://doi.org/10.1559/152304003100011090.

Soto Sánchez, J. E., Weyrich, T., Medeiros e Sá, A., and de Figueiredo, L. H. (2021). An integer representation for periodic tilings of the plane by regular polygons. *Computers & Graphics*, 95:69–80. DOI: https://doi.org/10.1016/j.cag.2021.01.007.

Suffern, K. G. (1990). Quadtree algorithms for contouring functions of two variables. *The Computer Journal*, 33(5):402–407. DOI: https://doi.org/10.1093/comjnl/33.5.402.

Sußner, G., Dachsbacher, C., and Greiner, G. (2005). Hexagonal LOD for interactive terrain rendering. In *Vision Modeling and Visualization*, pages 437–444. DOI: https://www.researchgate.net/publication/228673405.

Sußner, G. and Greiner, G. (2009). Hexagonal Delaunay triangulation. In *Proceedings of the 18th International Meshing Roundtable*, pages 519–538. Springer. DOI: https://doi.org/10.1007/978-3-642-04319-2_30.

Taubin, G. (1994). Distance approximations for rasterizing implicit curves. *ACM Transactions on Graphics*, 13(1):3–42. DOI: https://doi.org/10.1145/174462.174531.

Appendix. Code and comments

To illustrate how to use our representation in practice, we give here code for creation, subdivision, merging, and refinement. The snippets below are part of our Python code [de Figueiredo, 2024a], slightly simplified for readability.

The main global variable is `F` which contains the cloud, implemented as a dictionary indexed by face anchors denoted by `c` for center. We represent a center with lattice coordinates $[a, b]$ by the complex number $a + bj$, for convenience. The value of `F[c]` is `{'t':t,'s':s}`, where `t` is the type of the face and `s` is its scale. Access to these fields uses the dot notation for simplicity. The other global variable is `Q` which contains the refinement queue, implemented as a set. Finally, `A` is the key primitive described in §5.

When we create a face, we test whether there is already a face at the same position. If so, a merge occurs; it is performed simply by changing the type and scale of the existing face. New faces are automatically added to the refinement queue.

```
def addface(c,t,s):
    if c in F:
        F[c].t=6
        F[c].s=F[c].s+1
    else:
        F[c]={'t':t,'s':s}
    Q.add(c)
```

To subdivide a hexagon, create six new semi-hexagons, one for each orientation, at the same scale as the hexagon. Then, increment the scale of the hexagon to reflect its shrinking.

```
def subdivide(c):
    s=F[c].s
    for k in range(6):
        h=A(c,s+1,k)
        addface(h,k,s)
    F[c].s=s+1
```

We follow a similar strategy to refine a semi-hexagon at the boundary. To give the proper type to the new semi-hexagons, we take into account the type (orientation) of the original semi-hexagon.

```
def subdivide4(c):
    t=F[c].t
    s=F[c].s
    for j in range(2,4+1):
        k=(t+j)%6
        h=A(c,s+2,k)
        addface(h,k,s+1)
    F[c].s=s+1
```

We are now ready to describe the refinement process.

The refinement process is standard: we maintain a queue of faces to be examined for refinement and work until the queue is empty.

```
while Q:
    c=next(iter(Q))
    Q.remove(c)
    if needsrefinement(c):
        refine(c)
```

There are two refinement strategies. They coincide for hexagons but they differ in what they do when there is no mate for a semi-hexagon and the refinement has reached the boundary.

The mate of a semi-hexagonal face is the face adjacent to its longest edge. We compute the lattice coordinates of its center but do not query the cloud for it.

```
def mate(c):
    t=F[c].t
    s=F[c].s
    return A(c,s+1,t)
```

The first refinement strategy respects the boundary of the base mesh and subdivides the semi-hexagon.

```
def refine(c):
    if F[c].t==6:
        subdivide(c)
        Q.add(c)
    else:
        m=mate(c)
        if not (m in F):
            subdivide4(c)
        else:
            refine(m)
            refine(c)
```

The second refinement strategy extends the boundary of the base mesh as needed, creating a new hexagon as the absent mate. It is the strategy used in the case study.

```
def refine(c):
    if F[c].t==6:
        subdivide(c)
        Q.add(c)
    else:
        m=mate(c)
        if not (m in F):
            addface(m,6,F[c].s)
        refine(m)
        refine(c)
```