Engineering with Computers (2019) 35:1–20
https://doi.org/10.1007/s00366-018-0579-5

ORIGINAL ARTICLE

# ARTMe: a new array-based algorithm for Adaptive Refinement of Triangle Meshes

**Jéferson Coêlho**[1] · **Marcelo Gattass**[1] · **Hélio Lopes**[1]

Received: 6 February 2017 / Accepted: 5 January 2018 / Published online: 18 January 2018

**Abstract**
This work presents a new efficient array-based algorithm for adaptive mesh refinement capable of interactively generating millions of triangles. The new refinement algorithm satisfies important topological mesh properties, e.g., vertex valence control and a good mesh gradation. Furthermore, all local topological modifications of the triangle mesh are based on Stellar operators implemented on top of the Corner-Table topological data structure. This paper also shows that the proposed implementation provides a good balance in the trade-off between memory and processing time.

**Keywords** Mesh models · Topological data structures · Topological operators · Adaptive refinement · Triangulations · Stellar mesh operators · Algorithms

## 1 Introduction

A triangle mesh is a common object representation adopted in a wide range of applications, e.g., scientific computing [15], animation [17], and geological modeling [22], just to cite a few. In these applications, a triangle mesh usually represents a discrete approximation of an ideal continuous object [8]. The mesh representation is more accurate when it is somehow closer to the object. One solution for improving the accuracy is to refine the mesh, increasing the density of its vertices and triangles in specific regions where its geometry is far from the object.

On the one hand, by performing local refinements on the mesh, we increase the degrees of freedom and, thus, improve the triangular approximation of the continuous surface [8]. On the other hand, if we successively repeat the refinement process, the number of elements in the mesh grows exponentially [27]; thus, we use a considerable amount of computer memory.

Implementations of adaptive refinement algorithms commonly use traditional topological data structures, e.g., the Half-Edge [23]. For large meshes, however, the Mantyla's half-edge data structure consumes too much memory. For the special case of triangular meshes, more memory-efficient data structures, e.g., the Corner-Table [29], have been proposed. Producing good results with simple and cost-effective adaptive meshes using these reduced data structures is still an open issue.

*Contributions* The contributions of this work are twofold. It presents a new algorithm for adaptive mesh refinement and an efficient implementation of this algorithm based on the Corner-Table topological data structure. We assume that the triangle mesh is a combinatorial 2-manifold [19].

This new refinement algorithm was built to provide some desirable topological mesh properties, such as:

1. *Vertex valence control* To avoid generating vertices with high valence (e.g., with many edges) and to preserve their valence in the original mesh as much as possible;
2. *Good mesh gradation* In the resulting mesh the difference in refinement levels of two adjacent faces should not be greater than one [18]. This property is important because, when repetitive refinement operations are

**Electronic supplementary material** The online version of this article (https://doi.org/10.1007/s00366-018-0579-5) contains supplementary material, which is available to authorized users.

✉ Hélio Lopes
lopes@inf.puc-rio.br

Jéferson Coêlho
jcoelho@tecgraf.puc-rio.br

Marcelo Gattass
mgattass@tecgraf.puc-rio.br

1 Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, Brazil

applied to a specific region of the mesh, the relative resolution of such regions becomes greater than the rest of the mesh [27].

Our refinement algorithm can also continue the triangle-subdivision process from the last performed operation. More precisely, once a triangle has been partially refined in a previous step, the algorithm can identify and perform the remaining sequence of steps necessary to complete the triangular refinement. This property classifies the proposed refinement algorithm as an adaptive one. This characteristic is especially important for iterative methods, with which we can refine the mesh step by step. All local topological modifications to the triangle mesh are based on the Stellar operators [19].

Our algorithm is also flexible. With few or no modifications, it can refine meshes similar to traditional algorithms, e.g., simple triangulation [2], Red–Green triangulation [3] and incremental triangulation [27].

Our implementation of this new algorithm finds a good balance between memory and processing time, owing to the Corner-Table topological data structure [29]. It uses six references per triangle to represent the mesh connectivity, which is 2.75 times less than that of the Half-Edge [23], which associates with each half-edge a reference to the next, previous and opposite half-edge, together with a reference to a bounding vertex and incident face. Moreover, this implementation can efficiently undo and redo operations. Although the Corner-Table uses vectors to store the topological information, our implementation was projected so as to eliminate the necessity of garbage collection.

*Paper outline* This paper is organized as follows: Section 2 describes some previous and related work. Section 3 presents the implementation of the Stellar topological operators for locally modifying the triangle mesh using the Corner-Table data structure. Section 4 introduces our new efficient algorithm for the adaptive refinement of triangle meshes, named ARTMe. Section 5 describes some implementation details. Section 6 compares the proposed method with previous work. Section 7 shows the results. Finally, Section 8 discusses the advantages and disadvantages of the proposed algorithm.

## 2 Related work

The proposed algorithm involves three important areas: adaptive refinement strategies, topological operators, and topological data structures. This section reviews each of these areas to relate our proposal with previous work.

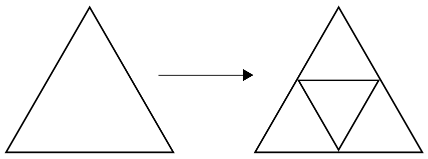


**Fig. 1** One-to-four refinement

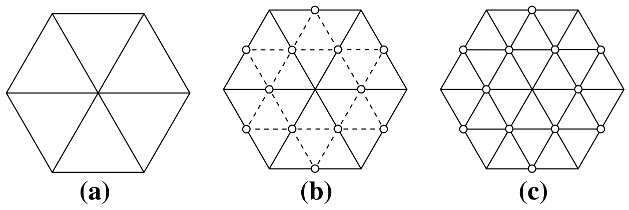


**Fig. 2** Refinement example using the loop, butterfly, and modified butterfly refinement operator. **a** Coarse mesh, **b** edges are refined, **c** refined mesh

### 2.1 Adaptive refinement

Subdivided surfaces are smooth free-form surfaces that are generated using recursive rules [6]. Starting with a control mesh, each step refines its triangles and repositions its vertices to produce a smooth surface at the limit [27]. Many subdivision schemes have been developed since [7] and [9]. Most of them are based on triangular patches, e.g., Loop [20], Butterfly [10], Modified Butterfly [33], $\sqrt{3}$-subdivision [18], and 4–8 subdivision [31].

Triangular subdivision schemes have two main elements: the rules to position the additional vertices and the refinement pattern used to refine the triangles. The former leads to the geometry, and the latter leads to the surface topology. Loop, Butterfly, and Modified Butterfly use the same refinement pattern, the so-called one-to-four refinement, shown in Fig. 1.

This refinement pattern inserts a new vertex for every edge of the given triangle and replaces each triangle by four new triangles. This refinement pattern has the positive effect of preserving the old vertices valence, and all newly inserted vertices inside the subdivided area have valence six [18]. Figure 2 illustrates the refinement of a coarse mesh using one-to-four refinement. Note that all new internal vertices have valence six.

Though the idea of adaptive subdivision seems to be simple, care must be taken to handle topological inconsistencies that arise when a triangle subset is subdivided [27]. Figure 3 shows an example of a one-to-four refinement that generates cracks or T-vertices, which must be fixed. A fast method of solving these inconsistencies is to connect the T-vertices to O-vertices, bisecting the face that has a lower

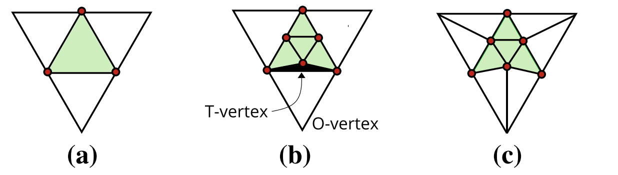


**Fig. 3** Refinement of one triangle and its neighborhood to remove cracks. **a** Selected triangle. **b** Crack generated by refinement process. **c** O-vertex triangulation to the remove the crack; i.e., split the neighboring triangles into new triangles that fit

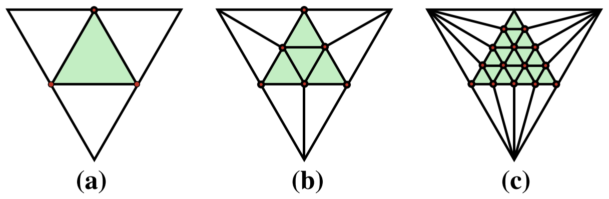


**Fig. 4** Refinement with O-vertex triangulation. **a** Coarse mesh, **b** level 1, **c** level 2

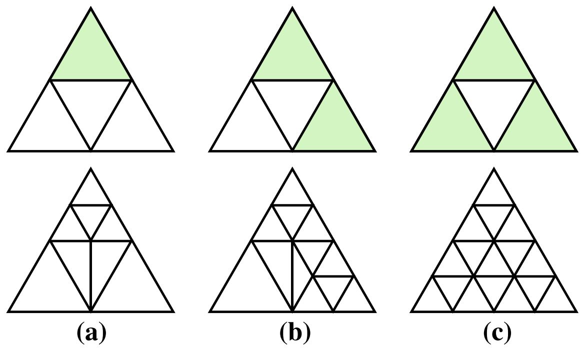


**Fig. 5** Simple triangulation templates when refining the light-green triangles generate cracks on the central triangle. **a** One crack, **b** two cracks, **c** three cracks

refinement level. From now on, we will call this method O-vertex triangulation.

While this triangulation method efficiently removes cracks, repeated refinements using O-vertex triangulation on a selected area produce high-valence O-vertices, which lead to undesirable long triangles. This effect, illustrated in Fig. 4c, occurs because the mesh has not a good mesh gradation. A mesh has a good gradation if no adjacent faces have a refinement-level difference greater than one.

There are other methods to remove cracks, e.g., the simple triangulation proposed by [2]. This method splits the adjacent triangles into two, three, or four new triangles, depending on the number of cracks, as shown in the three templates in Fig. 5. However, repeatedly using a combination of the first two templates of the simple triangulation scheme in the same area also generates high-valence O-vertices.

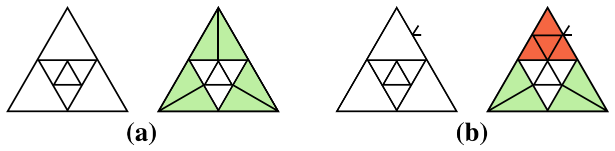


**Fig. 6** Red–Green triangulation rules. **a** Green rule. **b** Red rule for top triangle and green rule for bottom triangles

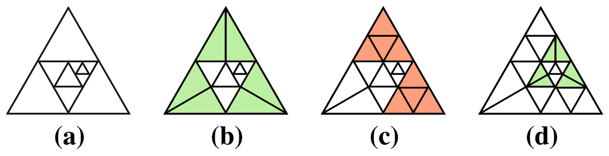


**Fig. 7** Red–Green triangulation example. The Red–Green rules are recursively applied to avoid T-vertices

Another method that removes cracks is the Red–Green triangulation method [3]. This method removes cracks and generates a gadration mesh at the same time by bisecting triangles with one crack (green rule), and splitting triangles with more than one crack per edge into four new triangles (red rule). Therefore, T-vertices, O-vertices, and the edges connecting them are temporary, and high-valence vertices are avoided. The Red–Green rules are shown in Fig. 6, and an example is shown in Fig. 7. However, according to [27], the Red–Green triangulation method is neither efficient nor simple.

An alternative to Red–Green triangulation is the incremental triangulation proposed in [27]. This method expands the selected area with a radius *r* and refines it using simple triangulation without generating high-valence O-vertices. The expansion guarantees that different vertices will be used as O-vertices at different levels of refinement. This property guarantees a good gradation and restricted mesh. Restricted mesh means that the vertices of the selected area and their neighbors have the same refinement level [34]. This property is relevant to subdivision methods like Loop, Butterfly, and Modified Butterfly; all vertices being calculated must be at the same refinement level to correctly compute new vertices on the mesh. According to [27], the incremental triangulation method is faster and simpler than Red–Green triangulation; however, it also generates more triangles. Figure 8 illustrates two refinement levels in the same selected area using incremental triangulation.

Some subdivision algorithms are designed in such a way that their adaptive refinement is a natural extension of the subdivision process. The $\sqrt{3}$-subdivision [18] and 4–8 subdivision [31] are examples of these.

The $\sqrt{3}$-subdivision method inserts a new vertex at the centroid of each triangle. This new vertex is connected to

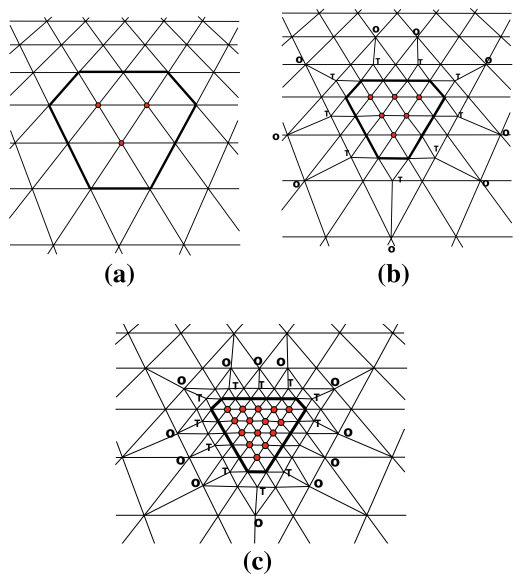


**Fig. 8** Two levels of refinement using incremental triangulation with $r = 1$. **a** Coarse mesh. **b** Level 1 of refinement. **c** Level 2 of refinement

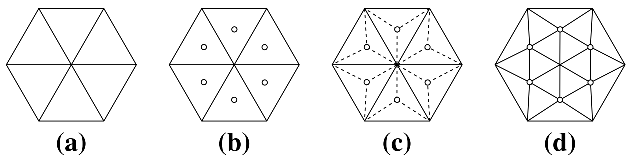


**Fig. 9** $\sqrt{3}$-subdivision refinement. **a** Coarse mesh. **b** New vertices are inserted at the centroid of the triangle. **c** Each new vertex is connected to the vertices of the triangle. **d** By flipping the original edges, the refined mesh is obtained

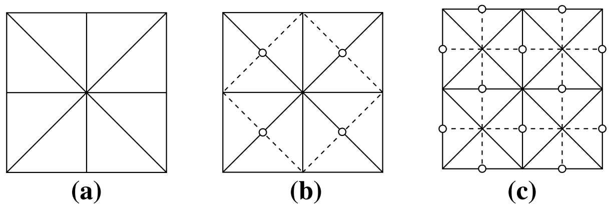


**Fig. 10** 4–8 subdivision refinement. **a** 4–8 mesh. **b** Refinement level 1. **c** Refinement level 2

the triangles existing vertices. In the end, all first edges of the triangle are flipped. This process is illustrated in Fig. 9.

Because the new vertices are inserted on faces, the pattern enables locally adaptive refinement without temporary crack fixes between neighboring triangles from different refinement levels.

Similar to $\sqrt{3}$-subdivision, the 4–8 subdivision method does not require post-processing steps to remove cracks. The 4-8 subdivision method works only on 4–8 meshes; i.e., their vertices have valences equal to 4 or 8 (see Fig. 10a). A basic block is formed by two triangles that form a quadrangle divided along one of its diagonals. The common edge to this pair of triangles is called the interior edge. All other edges are called exterior edges. In 4–8 subdivision, a new vertex is inserted in the interior edge of a basic block. The new vertex is connected to the vertices of the basic block, as illustrated in Fig. 10.

The $\sqrt{3}$-subdivision scheme does not produce a hierarchical triangle refinement because the triangles from one level do not contain sub-triangles of the next level. This lack of hierarchy is an undesired complexity for techniques like Adaptive Finite Element Analysis, where one must transfer field information from one level to another. The 4–8 subdivision method has a fixed structure that cannot deal with the arbitrary meshes that we intend to support. The adaptive refinement algorithm, proposed here, efficiently supports different refinement policies and is capable of reproducing the incremental triangulation or Red–Green triangulation methods, as discussed in Section 6.

## 2.2 Topological operators

In the above-mentioned refinement algorithms, all topological modifications on the mesh are implemented through operators on its representation. Baumgart proposed the Euler operators [4], which are based on the Euler–Poincaré theory [28]. They are composed of a set of low-level operators for modifying the connectivity of the mesh elements. Examples of these operators are MEV (Make an Edge and a Vertex) and MEF (Make an Edge and a Face). These two operators do not modify the Euler characteristic of the mesh; they only alter the local connectivity of the mesh elements. All refinement operations can be considered a composition of these two low-level operators. However, if we implement the refinement operations using Euler operators, we must deal with meshes with non-triangular faces at intermediate steps.

Another set of operators that locally modify the combinatorial structure of the mesh without changing its topology is the so-called Stellar operators, which are based on the Stellar theory [25, 26]. According to [19], the Stellar operators are highly suitable for changing the resolution and the combinatorial structure of a mesh. Examples of these operators include the following:

– The Edge Flip modifies the combinatorial structure of the mesh structure without changing its resolution or topology by swapping an interior edge. Many computational geometry algorithms use this operator.

- The Edge Split bisects an edge and its incident faces. When the edge is on the interior of the mesh, it substitutes the two triangles adjacent to that edge with four new triangles. If the edge is on a boundary, it replaces the unique incident triangle of that edge with two new triangles.
- The Weld operator is the inverse of the Edge Split operator.

Stellar theory states that any Stellar operation can be decomposed into a finite sequence of elementary Stellar operations on edges, which are composed of the three operators cited above [19].

In this paper, we adopt the Stellar operators as the basic low-level topological operators to refine the mesh. Details will be given in Sect. 3.

### 2.3 Topological data structures

As mentioned in Sect. 1, adaptive refinement-algorithm implementations commonly use traditional topological data structures, e.g., the Winged-Edge [4] and the Half-Edge [23]. This is because they facilitate the implementation of the topological operators that are the basis of all topological modifications performed on the mesh. OpenMesh [5] is an example of a library that uses Half-Edge for mesh refinement. The main disadvantage of these traditional topological data structures is their memory consumption. To find a good balance between memory consumption and the time complexity for answering topological queries, several concise topological data structures that specifically represent triangle meshes have been proposed. Examples of these are the Corner-Table [29], the SQuad [11], the LR [12], the ESQ [1], and the Zipper [13]. These data structures use very low memory storage when compared to the Half-Edge. However, they are unsuitable for performing local topological modifications, e.g., refinement and simplification.

In this paper, we adopt the Corner-Table topological data structure. The main reasons for choosing this data structure are as follows:

- It is concise. The Corner-Table uses six references per triangle (rpt) to represent the connectivity, while the Half-Edge uses 16.5 rpt [13].
- It gives time-efficient answers to queries about important adjacency and incidence relations among the elements of the mesh, which facilitates implementing the Stellar operators.

## 3 Stellar operators on the Corner-Table

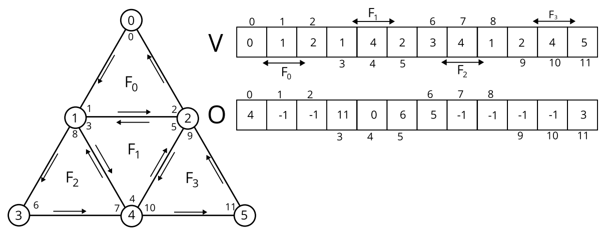
The main objective of this section is to present the implementation of the Stellar operators based on the Corner-Table topological data structure. An efficient implementation of such operators is fundamental for constructing the adaptive refinement algorithm. For simplicity, from now on, we do not consider the cases when the triangle is on the mesh boundary. However, the implementations presented here can be easily adapted to deal with boundary cases by testing O-vector value. If this value is equal to boundary value, no vector need to be updated. In case to be accessing a vertex star, every time a boundary was found the algorithm should stop, go back to the beginning and turn in the other direction. In the supplementary material you can find the full code of the proposed algorithms, including the boundary testing procedures, in the URL: https://web.tecgraf.puc-rio.br/press/publication/Coelho2018/

### 3.1 Corner-Table

The Corner-Table is an array-based topological data structure for triangle-mesh representation. In this data structure, each oriented triangle is represented by three consecutive entries in the array. The index of each entry will be called a corner. Thus, a mesh with $n$ triangles will have $3n$ corners. By definition, the corners $3i$, $3i+1$, and $3i+2$ correspond to the $i$-th triangle of the mesh and vice-versa. We can use the following formula to obtain the index t of the triangle from a given corner c:

$$t(c) = \lfloor \frac{c}{3} \rfloor. \tag{1}$$

In the original work of [29], the topological representation adopted in the Corner-Table consists of two integer vectors, O and V, with dimensions equal to $3n$, where $n$ is the number of triangles. For each corner $c$, V[$c$] is the vertex of the corner, and O[$c$] is the opposite corner in the adjacent triangle. Note that table V represents a set of triangles using an index array. The vertices attributes, along with their geometry, are stored in another table G, whose dimension is given as a function of the number of vertices in the triangulation. To represent the boundary curve of the mesh, the opposite of a corner $c$ that is opposite a border edge is defined with a



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| V | 0 | 1 | 2 | 1 | 4 | 2 | 3 | 4 | 1 | 2 | 4 | 5 |
| O | 4 | -1 | -1 | 11 | 0 | 6 | 5 | -1 | -1 | -1 | -1 | 3 |

Fig. 11 Triangle mesh represented by the Corner-Table data structure

special ID, usually −1; i.e., O[$c$] = −1. Figure 11 illustrates the tables O and V using a simple triangular mesh.

Because the corners of a triangle are consecutively indexed, it is then possible to write functions to obtain the so-called next and previous corners of a corner c, within the same triangle, using only integer operations:

$$n(c) = \text{next}(c) = 3 \cdot t(c) + \{(c+1) \mod 3\} \tag{2}$$

$$p(c) = \text{prev}(c) = 3 \cdot t(c) + \{(c+2) \mod 3\} \tag{3}$$

With these operators, the right and left triangles of a given corner $c$ can be obtained as follows:

$$l(c) = \text{left}\ (c) = O[\text{prev}(c)] \tag{4}$$

$$r(c) = \text{right}(c) = O[\text{next}(c)] \tag{5}$$

Figure 12 illustrates the corners resulting from the above operations for the corner $c$. We can observe that the corners represented in Fig. 11 are not explicitly stored. A corner is, in fact, the index of an entry in the V table, and so it has an implicit representation.

## 3.2 Edge flip

The Edge Flip is a Stellar operator that transforms two adjacent faces into two new adjacent faces through a swap of their common edge.

Let the common edge be $e = \langle uv \rangle$ and $s$ and $t$ are the two opposite vertices of the edge $e$, as shown in Fig. 13. The operation Edge Flip replaces the edge $e$ by $\langle st \rangle$ and the the triangles incident to $e$ by $\langle tsu \rangle$ and $\langle vst \rangle$. Notice that the valence of the vertices $u$ and $v$ decreases, while the valence of the vertices $s$ and $t$ increases.

Algorithm 1 performs the Edge Flip operation on the edge opposite to corner $c_0$, turning the faces involved in the trigonometric orientation.

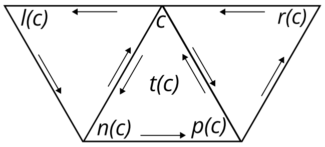


**Fig. 12** Corners $n(c)$, $p(c)$, $l(c)$, and $r(c)$ from operations next, previous, left, and right, respectively. The index of the triangle that contains the corner $c$ is denoted by $t(c)$

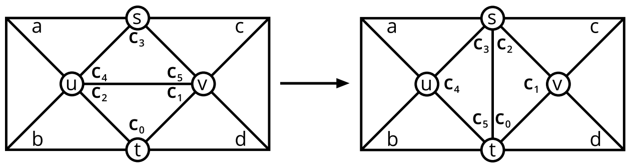


**Fig. 13** Edge Flip operation using the Corner-Table data structure

```
Algorithm 1: EdgeFlip( $c_0$ )
  // Identify incident corners
  $c_1$ = next( $c_0$ ); $c_2$ = prev( $c_0$ ); $c_3$ = O[ $c_0$ ];;
  $c_4$ = next( $c_3$ ); $c_5$ = prev ( $c_3$ ) b = O[ $c_1$ ];
  c = O[ $c_4$ ];
  // Identify incident vertices
  t = V[ $c_0$ ]; u = V[ $c_2$ ];
  s = V[ $c_3$ ]; v = V[ $c_1$ ];
  // Perform edge flip
  V[ $c_2$ ] = s; V[ $c_5$ ] = t;
  // Update opposite corners
  O[ $c_1$ ] = $c_4$; O[ c ] = $c_0$; O[ $c_3$ ] = b;
  O[ $c_4$ ] = $c_1$; O[ $c_0$ ] = c; O[ b ] = $c_3$;
```

This implementation has the important property that only two corners, prev($c_0$) and prev($O[c_0]$), change their corresponding vertices; i.e., the corners $c_0$, $O[c_0]$, next($c_0$), and next($O[c_0]$) remain associated with the same vertices. This property will be of vital importance when implementing the refinement algorithm.

## 3.3 Edge Unflip

To implement an undo/redo mechanism, it is important to have inverse operators. Thus, an Edge Unflip is required to reverse the effect of the Edge Flip operator. The inverse of an Edge Flip operator could be the Edge Flip operator itself. Because it performs a counterclockwise rotation along the two adjacent triangles, Edge Unflip should perform a clockwise rotation instead. Thus, we propose an implementation of the Edge Unflip operator that turns the involved faces in the clockwise direction (Fig. 14).

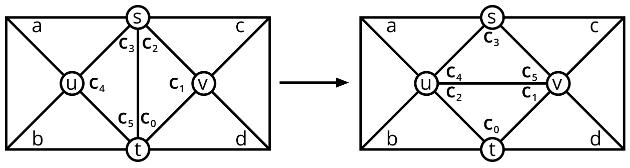


**Fig. 14** Edge Unflip operation using the Corner-Table data structure

Algorithm 2, with small changes in relation to Algorithm 1, implements the Edge Unflip operator. In addition to restoring the Corner-Table configuration to the one prior to applying the Edge Flip operator, the operator also keeps the corners $c_1$, $O[c_1]$, prev($c_1$), and prev($O[c_1]$) associated with the same vertices.

**Algorithm 2:** Unflip( $c_1$ )

```
// Identify incident corners
$c_2$ = next( $c_1$ ); $c_0$ = prev( $c_1$ ); $c_3$ = prev ( $c_4$ );
$c_4$ = O[ $c_1$ ]; $c_5$ = next( $c_4$ );
b = O[ $c_3$ ]; c = O[ $c_0$ ];
// Identify incident vertices
u = V[ $c_4$ ]; v = V[ $c_1$ ];
s = V[ $c_2$ ]; t = V[ $c_0$ ];
// Perform edge unflip
V[ $c_5$ ] = v; V[ $c_2$ ] = u;
// Update opposite corners
O[ $c_0$ ] = $c_3$; O[ b ] = $c_1$; O[ $c_4$ ] = c;
O[ $c_3$ ] = $c_0$; O[ $c_1$ ] = b; O[ c ] = $c_4$;
```

**Algorithm 3:** EdgeSplit( $c_0$ )

```
$F_n$ = number of triangles on current mesh;
$F_{n+1} = F_n + 1$;
// Identify incident corners
$c_1$ = next( $c_0$ ); $c_2$ = prev( $c_0$ ); $c_3$ = O[ $c_0$ ];
$c_4$ = next( $c_3$ ); $c_5$ = prev ( $c_3$ );
d = O[ $c_2$ ]; a = O[ $c_5$ ];
// Identify incident vertices
t = V[ $c_0$ ]; v = V[ $c_1$ ];
s = V[ $c_3$ ]; u = V[ $c_2$ ];
// Update the triangulation adding two triangles
V[ $c_1$ ] = n; V[ $c_4$ ] = n;
V[ $3 \cdot F_n$ ] = t; V[ $3 \cdot F_n + 1$ ] = v;
V[ $3 \cdot F_n + 2$ ] = n; V[ $3 \cdot F_{n+1}$ ] = s;
V[ $3 \cdot F_{n+1}$ ] = u; V[ $3 \cdot F_{n+1} + 2$ ] = n;
// Update opposite corners
O[ $c_0$ ] = $c_9$; O[ $c_5$ ] = $c_{10}$; O[ $c_2$ ] = $c_7$; O[ $c_{11}$ ] = a;
O[ $c_6$ ] = $c_3$; O[ $c_8$ ] = d;
O[ $c_9$ ] = $c_0$; O[ $c_{10}$ ] = $c_5$; O[ $c_7$ ] = $c_2$; O[ a ] = $c_{11}$;
O[ $c_3$ ] = $c_6$; O[ d ] = $c_8$;
```

## 3.4 Edge Split

The Edge Split operation transforms two adjacent triangles into four triangles by inserting a new vertex on the common edge.

Let $e = \langle uv \rangle$ and $s$ and $t$ be two opposite vertices of the edge $e$, as in Fig. 15. The Edge Split operation creates a new vertex $n$ over the edge $e$; and this vertex is then connected to vertices $s$ and $t$ generating new edges $e_1 = \langle nt \rangle$ and $e_2 = \langle ns \rangle$. Moreover, faces $\langle tvu \rangle$ and $\langle suv \rangle$ are replaced by faces $\langle tnu \rangle$, $\langle snv \rangle$, $\langle tvn \rangle$, and $\langle sun \rangle$. The new vertex $n$ will always have valence 4, and vertices $s$ and $t$ will have their valence increased by one.

From now on, in the Corner-Table data structure, each triangle will be implicitly represented by an integer index named $F_i$.There are several ways to reposition triangles $F_1$ and $F_2$ to subdivide the edge $e$. To facilitate the algorithm refinement and undo/redo mechanism, we will be adopting the convention shown in Fig. 16. In our convention, the pre-existing triangles $F_1$ and $F_2$ are positioned in the new configuration at the same place, regardless of whether the operation is carried out in the corner $c$, or $o$, the corner opposite $c$.

Algorithm 3 performs the Edge Split operation on the edge opposite to corner $c_0$, following the adopted convention. By applying the Edge Split operator, faces $F_1$ and $F_2$ remain in the same positions in table V, and only one of each of their vertices changes during the operation. The two new triangles are added to the end of table V.

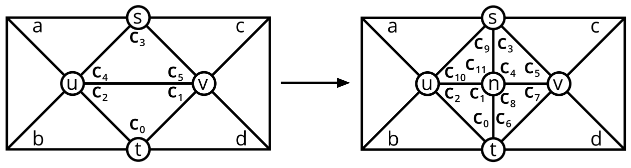


Fig. 15 Edge Split operation using the Corner-Table data structure

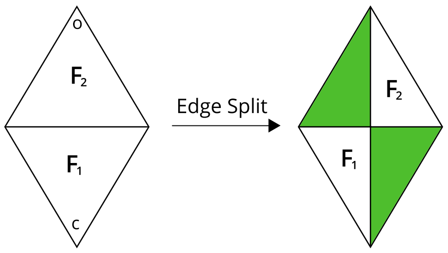


Fig. 16 Adopted convention for arranging triangles in the Edge Split operation

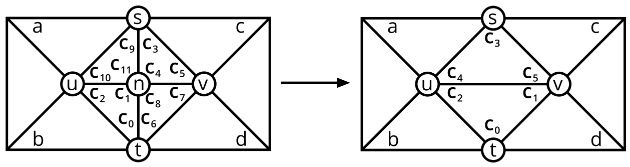


**Fig. 17** Edge Weld operation using the Corner-Table data structure

## 3.5 Edge Weld

The Edge Weld operator is the inverse of the Edge Split operator, and its implementation should be suitable for the undo/redo mechanism. This operator is designed to ensure that the application of an Edge Split followed by an Edge Weld restores the data structure to its original configuration.

Let *n* be a four-valence vertex, adjacent to vertices *s*, *t*, *u*, and *v*, as shown in Fig. 17. The Edge Weld operator applied from corner $c_1$ removes the vertex *n* and the edges $\langle tn \rangle$ and $\langle sn \rangle$ so that the resulting two triangles are $\langle tvu \rangle$ and $\langle suv \rangle$.

```
Algorithm 4: EdgeWeld( $c_1$ ).
    // Identify incident corners
    $c_0$ = prev( $c_1$ ); $c_2$ = next( $c_1$ ); $c_7$ = O[ $c_2$ ];
    $c_3$ = O[ prev( $c_7$ ) ]; $c_4$ = next( $c_3$ );
    $c_5$ = prev ( $c_3$ ); d = O[ next($c_7$) ];
    a = O[ next( O[ $c_5$ ] ) ];
    // Identify incident vertices
    v = V[ $c_7$ ]; u = V[ $c_2$ ];
    s = V[ $c_3$ ]; t = V[ $c_0$ ];
    // Update the triangulation
    V[ $c_4$ ] = u;
    V[ $c_1$ ] = v;
    // Update opposite corners
    O[ $c_5$ ] = a; O[ d ] = $c_2$; O[ $c_0$ ] = $c_3$;
    O[ a ] = $c_5$; O[ $c_2$ ] = d; O[ $c_3$ ] = $c_0$;
```

Because this operator was constructed to reverse the Edge Split operation, the two removed triangles are at the end of vector V, as seen in Sect. 3.4. Thus, to remove these two triangles, we simply decrement the number of triangles. This simple procedure avoids the need for garbage collection operations by keeping the tables always full, without unused spaces.

Vieira et al. [32] proposed an implementation of the Edge Flip and Edge Weld operators to construct a simplification algorithm. This work presents another implementation of these operators to produce a suitable modification for the refinement algorithms.

## 4 ARTMe

This section firstly describes the Stellar operators composition to perform the one-to-four refinement operator and, after that, the adaptive refinement (ARTMe) algorithm for triangular meshes based on the Corner-Table. ARTMe executes an adaptive refinement that can continue the refinement process from the last operations carried out.

### 4.1 One-to-four refinement operator

Unlike general topological data structures, e.g., Half-Edge, the Corner-Table can only represent pure triangular meshes. All refinement algorithms to be presented in this paper use the one-to-four refinement operator that transforms one triangle into four triangles. However, the one-to-four operator's implementation on top of the Corner-Table requires that the mesh contain only triangles in all its intermediate steps. Therefore, this operator is performed by a composition of Stellar operations, as illustrated in 18.

As shown in Sect. 2.1, to generate a conformal mesh, i.e., a mesh without T-vertices, the one-to-four refinement of a triangle must be propagated to adjacent triangles. For each subdivided edge, the opposite triangle should be subdivided into two, by connecting the newly generated vertex with the vertex opposite this edge [27], as shown in Fig. 3. Although this strategy avoids creating T-vertices, the refinement propagates to the adjacent triangles.

The one-to-four refinement operator, with the opposing triangle refinement, can be implemented by a sequence of four basic Stellar operations: three Edge Split operations, followed by an Edge Flip. The Edge Split operations divide the three triangle edges, while the Edge Flip is executed on the first edge created by the first Edge Split carried out on the triangle. The composition for this operation is illustrated in Fig. 18.

Because the one-to-four refinement operator is a composition of Stellar operations, this operator is also a Stellar operator; as described in Sect. 2.2, a Stellar operator does not change the surface topology. Note that all the new vertices generated by a complete one-to-four refinement operator have a valence of six. Moreover, this operator increases the O-vertex valence by one. In addition, we would like to observe that the inverse of this operator is obtained by

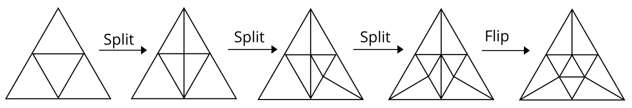


**Fig. 18** One-to-four refinement on the central triangle and its neighborhood

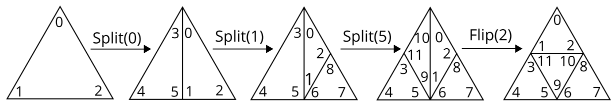


**Fig. 19** A sequence of operators in a triangle refinement step using the Corner-Table data structure: EdgeSplit (0), EdgeSplit (1), EdgeSplit (5), and EdgeFlip (2)

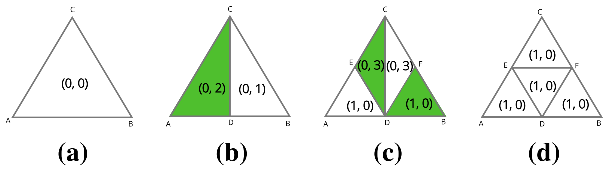


**Fig. 20** Refinement process from a level-0 triangle to four level-1 triangles and the level-updating scheme at intermediary steps, where the pair $(l, s)$ indicates, respectively, the level and sublevel of a triangle

executing the sequence of Stellar operators that define it, in reverse order, in their inverse forms.

Figure 19 illustrates the application of a one-to-four operator in a mesh. This illustrative example shows how the triangles' corners are modified according to the operator conventions defined in Sect. 3. Corner 0 is in the first triangle to be subdivided. Corner 1 is queued so that the faces in the star vertex V[1] (i.e., faces that are incidents to V[1]) are subdivided, and corner 2 is saved for the Edge Flip operator.

The main difficulty in this refinement algorithm occurs when we must apply the one-to-four operator to a triangle in order to refine its neighbors and avoid creating T-vertices. Thus, when the algorithm reaches these neighboring triangles, they have already received the first Edge Split. Consequently, we must manage the states of the mesh triangles during the refining process to ensure that each one is only refined once. We do this using a level and sublevel for each triangle.

## 4.2 Levels and sublevels management

Figure 18 illustrates the four-step one-to-four triangle-refining process. In each of these steps, it is necessary to record the level and sublevel of each triangle. The sublevel information is required to identify the sub-process that the triangle must undergo. Figure 20 shows the process of refining a level-0 triangle to four level-1 triangles. This figure also shows the (level, sublevel) pair for each triangle, as the refinement process progresses.

There are four possible sublevels for a triangle:

0: Indicates that the triangle is in a final state.

1: Indicates that the triangle was created by a refinement of a sublevel-0 triangle; if the subdivision continues in this triangle, it will be subdivided in two. The current triangle will change to become a sublevel-3 triangle (triangle $\langle cfd \rangle$ in Fig. 20c) and a new sublevel-0 triangle (triangle $\langle dfb \rangle$ in Fig. 20c) is created.

2: Similar to sublevel 1, except that when the current triangle is divided, it becomes a sublevel-0 triangle (triangle $\langle aed \rangle$ in Fig.re 20c) and the new created face becomes a sublevel-3 triangle (triangle $\langle ced \rangle$ in Fig. 20c).

3: Indicates that the triangle has already been subdivided and must undergo an Edge Flip to continue to the next level.

Despite the similarities of sublevels 1 and 2, they differ in the position of the sub-triangles in the next refinement step. We could implement this difference as a parity code for all faces with sublevel 1, but for simplicity and to save space, we decided to distinguish them in the sublevel, creating sublevels 1 and 2. That is, sublevel 2 is the same as sublevel 1, but with a different parity.

The vertex also needs to store the refinement level. Given that the vertex is generated only by the Edge Split operations, and that these operations cut only triangles of the same level, the vertex level is given by the triangle level plus one. This level scheme is also illustrated in Fig. 20, where vertices A, B, and C have level 0 and vertices D, E, and F have level 1.

Algorithm 9 updates the level of the triangles when it performs the Edge Split operation through the Algorithm 5. Note that *f*1 is an existing triangle, and *f*2 is a new one. Algorithm 6 updates the triangle level and sublevel after the Edge Flip operation.

**Algorithm 5:** processFaceLevels ( f1, f2 )

```
sublevel = f1.sublevel;
if (subLevel == 0) then
    f2.sublevel = f1.sublevel + 2;
    f1.sublevel++;
end
if (subLevel == 1) then
    f2.level = f1.level + 1;
    f2.sublevel = 0;
    f1.sublevel += 2;
end
if (subLevel == 2) then
    f1.level++;
    f1.sublevel = 0;
    f1.sublevel++;
end
```

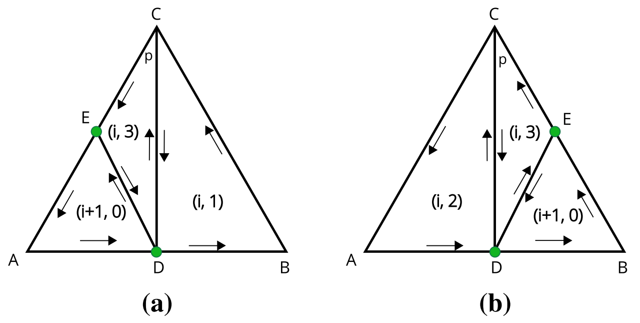


**Fig. 21** **a** Once a corner p is identified, the triangle to be refined is the one incident to the corner right ($p$). **b** Once a corner $p$ is identified, the triangle to be refined is the one incident to the corner left ($p$)

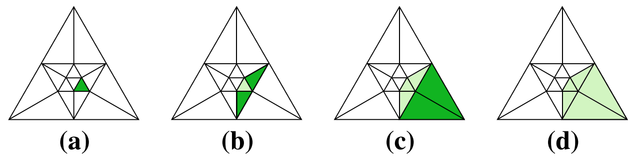


**Fig. 22** **a** Selected triangle to be refined. **b** Adjacent lower-level triangles (dark green) are also selected to be refined. **c** New adjacent triangles with a lower level (dark green) than the last added faces are also selected to be refined. **d** Final triangle list to be refined

## 4.3 Preprocessing

The main difficulty in the adaptive algorithm is handling the transition regions created to avoid T-vertices. In these regions, we may find sublevel-3 triangles sharing an edge with a sublevel-1 or 2 triangle (see Fig. 21). These sublevel-3 triangles must be refined after we split the opposing triangle and flip the common edge to complete the transition of both triangles to the next refinement level. The strategy proposed here is: whenever a sublevel-3 triangle appears in the refinement list, we preprocess this triangle by adding the sublevel-1 or 2 triangle that shares the edge that must be flipped. Subdividing these triangles automatically completes the subdivision process in the original triangle. As this replacement must occur before we process the refinement step, we call it preprocessing.

Figure 21 shows two sublevel-3 triangles that have an edge composed of two vertices (D and E) that are already in the next refinement level. This edge can be used as a guide to finding the correct triangle to add to the refinement procedure. The first step is to identify the corner “p” opposite that edge. Figure 21 also illustrates the two possibilities: in (a), a sublevel-3 triangle was generated from a sublevel-2 triangle; thus, the substitute triangle can be identified as the one that contains the corner right ($p$). In (b), the sublevel-3 triangle was generated from a sublevel-1 triangle, and the substitute triangle is the one that contains the corner left ($p$).

To produce a mesh with a good gradation, it is necessary that the level difference between adjacent triangles does not exceed one [18]. In our algorithm, this is equivalent to stating that the Edge Split operation can only operate on faces with the same level. To ensure this property, we recursively add, in pre-processing, all the triangles surrounding the selected triangles with a lower refinement level to the selected set of triangles to be refined.

This process is illustrated in Fig. 22. An initial triangle is selected to be refined. This triangle has neighbors with lower levels, so they should also be added to the refined list. Moreover, these new triangles may also have neighbors with a lower level; in this case, they are also added to the same list. In the worst case, all triangles in the mesh will be added to the list; then, the complexity of this step is $O(n)$, where $n$ is the number of triangles in the mesh.

In Sect. 6, we show that the level management, with slight modifications to this preprocessing, is enough to generate Red–Green triangulation.

## 4.4 Refinement algorithm

In this section, we will describe the refinement algorithm. We will use a Boolean vector as extra information to determine whether a triangle was processed or not. A triangle is processed when it reaches sublevel 0 or 3. At the beginning, all triangles to be refined will be marked as not processed, and those that remain are marked as already processed.

One difficulty to overcome is that the edge where the Edge Flip operator should be applied should be registered after all the triangle edges have been subdivided. We observe that using our proposed implementation for the topological operators, the edge receiving this operation is always the first edge corresponding to the split triangle.

A sketch for the proposed refinement algorithm is outlined as follows:

1. Pre-process the input list of triangles to be refined.
2. Initialize the processed-triangles vector.
3. For each non-processed triangle

    (a) Apply the Edge Split operator to the edge corresponding to an arbitrary initial corner $c_0$. This operation generates the first vertex / edge pair $(v_0, e_0)$, which must be queued in $Q$. Here, $v_0$ is the newly generated vertex and $e_0$ the edge that subdivided the opposing triangle. Store the corner corresponding to the edge, $e_f$, to be flipped in the triangle that began the refinement process, and update levels and sublevels.

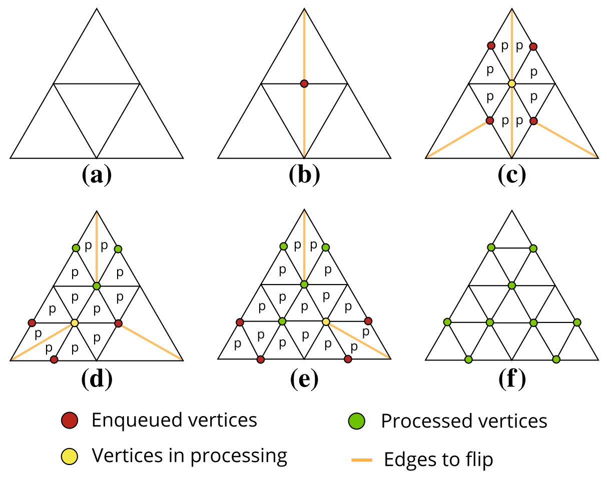


**Fig. 23** **a** Initial mesh to be refined. **b–e** The pairs of corners corresponding to ($v$, $e$) are removed from the queue, and the triangles of the star of $v$ that have not been processed are refined. When processing a vertex, the triangles of its star are marked as processed. After subdividing all triangles in the star, the Edge Flip operator is applied to the opposite edge $e$. **f** Refined mesh

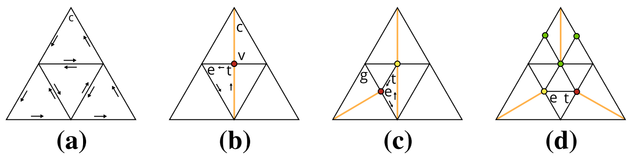


**Fig. 24** **a** Initial mesh. **b** The Edge Split operator is applied to the corner $c$. The pair ($v$, $e$) is inserted into the queue $Q$. **c** When we subdivide the triangle on the star of $V[v]$, the Edge Split operator is applied to the corner $t$, and the pair ($e$, $g$) is inserted into the queue $Q$. **d** After processing the vertex $V[v]$, the Edge Flip operator is applied at the corner $e$. Only the corner $t$ = prev($e$) associated to the already-processed vertex alters its vertex. Thus, the other corners of the triangle that are in the queue remain associated with their same vertices

(b) While the queue $Q$ is not empty, remove the first pair of corners corresponding to vertex $v$ and edge $e$. Refine all triangles in the star of vertex $v$ that have not yet been processed. Each subdivided edge generates a new pair ($v_i$, $e_i$) that is queued in $Q$. Each triangle of star $v$ is marked as processed and all have their level and sublevel updated. Finally, apply the Edge Flip operator to edge $e$.

(c) To conclude, apply the Edge Flip operator to edge $e_f$.

Figure 23 illustrates these steps when all triangles are selected to be refined. The highlighted edges are the ones that must have the Edge Flip operator applied after its star is processed.

Note that when we apply the Edge Split operator, the corner of the next vertex to be inserted in the queue is always the next of the current corner. Furthermore, the edge that receives the Edge Flip operator, when the current vertex finishes processing, is the opposite of the previous current corner. The careful implementation of the Edge Split operator, presented in Sect. 3.4, ensures that the triangles opposing the corner opposite the flipped edge remain opposite the same edge, even after the other two edges of the triangle are subdivided. Figure 19 illustrates this point.

This invariance is necessary because the Edge Flip operator changes the position of some corners and, in theory, could move a corner associated with a vertex that is already in the queue, yielding a wrong reference that could break the algorithm. The vertices entered in the queue, however, continue to be associated with the same vertex index. As described in Sect. 3.2 and illustrated in Fig. 24, when applying the Edge Flip operator at corner $e$, the corner of the triangle that changes its vertex is prev ($e$), which is associated with the vertex that is being processed. Thus, the only lost reference is to vertex V[v], which has already been removed from the queue. The same invariance is valid in the refinement of the opposing triangle.

An iterative refinement process subdivides each triangle in the mesh until a given local condition is satisfied. Here, we are interested in adaptive refinement where:

1. The refinement can be implemented in steps.
2. The result of a step depends only on the state of the current mesh-refinement process and not on the refinement history.
3. Given an initial mesh and a set of local refinement conditions to be met, the resulting refined mesh is independent of the order of the refining steps or which criterion is applied first.

These three requisites yield a general robust adaptive refinement algorithm that is simple to apply to a given problem. It is general because it only depends on a local evaluation that defines whether the refinement should proceed in a given triangle. This evaluation can be passed to the algorithm as a formula or a callback function. It is robust because the refinement order does not affect the result. The level (for triangles and vertices) and sublevel control is sufficient to implement requirement 2.

With the concepts presented above, we can now finally present the adaptive refinement algorithm (ARTMe).

We can observe that the ARTMe algorithm could start from any triangle, except the ones with sublevel 3. However, as explained in Sect. 4.3, the preprocessing step ensures that this case will never occur. In the case where the initial triangle is sublevel 0, the algorithm could start from any of its incident vertices. When the initial triangle is a sublevel-1 or 2 triangle, it is necessary to perform an Edge Split operation from the incident higher-degree

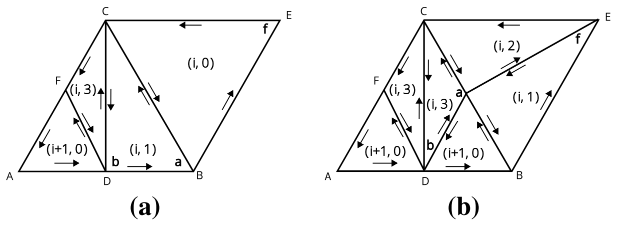


**Fig. 25** Edge Split operation from corner $f$ with an opposing sublevel-1 triangle. Corner $a$ is identified before the Edge Split operation. After that, if the opposing triangle to corner $a$ is at sublevel 3, an Edge Flip operation should be applied to $a$

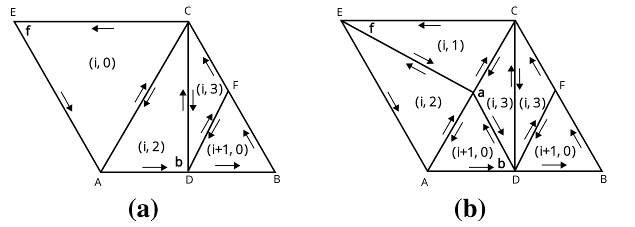


**Fig. 26** Edge Split operation from corner $f$ with an opposing sublevel-2 triangle. Corner $a$ is identified after the Edge Split operation. After that, if the triangle opposing $a$ is at sublevel 3, an Edge Flip operation should be applied to $a$

vertex, i.e., the vertex that was generated by the operation that changed the triangle sublevel from 0 to 1 or 2. For example, note that in Fig. 20(b), to refine one of the two triangles in a correct form, the algorithm should start with vertex $d$.

Another condition that should be observed is that the refinement should be performed in an ordered sequence, with the order of the sequence defined by the triangle level. This guarantees that a triangle with a lower level completes its refinement before an Edge Split can be applied to it, which is caused by the refinement of an adjacent triangle. We can observe that this is equivalent to saying that the Edge Split operation can only work on faces with the same refinement level.

Because an adaptive refinement algorithm should be able to operate on a partially refined triangle, it is possible that when we refine a sublevel-1 or 2 triangle using an Edge Split operation started from another triangle, the only possible operation that could be applied is an Edge Flip. For examples, see Figs. 25 and 26. In this examples, the algorithm should implement an Edge Flip, and the corresponding triangles should be promoted to the next refinement level.

In the case when the new edge is incident to a sublevel-1 triangle (see Fig. 25), corner $a$ should be identified before the Edge Split operation is realized in $f$. After this operation, it should be verified whether the triangle opposing corner $a$ is at sublevel 3; if so, the Edge Flip operation should be applied from corner $a$.

In the case when the new edge is incident to a sublevel-2 triangle (see Fig. 26), corner $a$ should be identified after the Edge Split operation in $f$. Similar to the previous case, after this operation is applied, it can be verified whether the triangle opposing corner $a$ is at sublevel 3; in the affirmative case, the Edge Flip operation should be applied from corner $a$.

```
Algorithm 6: performEdgeFlip( corner )
if (corner == -1) then
    return;
end
triangle = t(corner);
opositeTriangle = t( oposite[ corner ] );
if (triangle.sublevel == 3 and opositeTriangle.sublevel == 3) then
    edgeFlip( corner );
    triangle.sublevel = 0;
    triangle.level++;
    opositeTriangle.sublevel = 0;
    opositeTriangle.level ++;
end
```

Algorithm 9 performs both verifications. When one of these cases occurs, the Edge Flip operation is necessary. Algorithm 6 executes this operation, and updates the level and sublevel of the related triangles. Algorithm 7 initializes the vector that marks the processed faces. Algorithm 8 sets a triangle as processed when it is necessary.

```
Algorithm 7: init ( trianglesToRefine, processedTriangles )
for each triangle f do
    processedTriangles[ f ] = TRUE;
end
for each triangle f in trianglesToRefine do
    processedTriangles[ f ] = FALSE;
end
```

```
Algorithm 8: processTriangles( processedTriangles[], f1, f2 )
if (f1.sublevel == 0 or f1.sublevel == 3) then
    processedTriangles[ f1 ] = TRUE;
end
processedTriangles[ f1 ] = processedTriangles[ f2 ];
```

Algorithm 9 is responsible for applying Edge Split operations to the triangles, updating the levels of the vertices and triangles involved, marking the triangles as processed, and verifying whether or not an edge should be flipped after an Edge Split operation. Such a flip will only be necessary when the algorithm refines a triangle that has been partially subdivided.

**Algorithm 9:** performSplit( corner, processed-Triangles[], Q )

```
f = number of triangles on current mesh;
v = number of vertices on current mesh;
face = t( corner );
oppositeTriangle = t( opposite[ corner ] );
v.level = face.level + 1;
processTriangleLevels( triangle, f );
processTriangles( processedTriangles, triangle, f );
flipEdge = -1;
if (oppositeTriangle.sublevel == 1) then
    flipEdge = next( opposite[ corner ] );
end
EdgeSplit( corner );
if (oppositeTriangle.sublevel == 0 and
processedTriangles[ oppositeTriangle ] == FALSE)
then
    e = prev( opposite );
end
processTriangleLevels( oppositeTriangle, f + 1 );
processTriangles( processedTriangles,
oppositeTriangle, f + 1 );
if (oppositeTriangle.sublevel == 2) then
    flipEdge = prev( opposite[ corner ] );
end
performEdgeFlip( flipEdge );
Q.insert( (next( corner ), e) )
```

Algorithm 10 refines the not-yet-processed triangles. We can observe that Algorithm 10 can verify whether the Edge Flip operation is necessary. This occurs when the algorithm refines a triangle with sublevel 1 or 2.

Suppose that the algorithm starts refining a sublevel-1 triangle, as illustrated in Fig. 25. After the Edge Split operation on $b$, an Edge Flip operation on the corner next ($b$) is necessary. Now, consider the situation when the algorithm starts the refinement on a sublevel-2 triangle, as shown in Fig. 26. Then, after the Edge Split on $b$, an Edge Flip on the corner next (left ($b$)) is necessary. Algorithm 10 tests this after the first application of the Edge Split operation, and executes the Edge Flip operation when necessary.

**Algorithm 10:** refineTriangles( processedTriangles, corner )

```
Queue Q;
f = t( corner );
cornerToFlip = -1;
if (f.sublevel == 0) then
    cornerToFlip = prev( corner );
end
performSplit( processedTriangles, Q, corner );
//Try to flip the edge if it's possible.
if (f.sublevel > 0) then
    if (f.sublevel == 1) then
        //Get the next corner on triangle.
        flipOnTriangle = next( corner );
    else
        //Get the corner that point at the edge that
        can be flipped.
        flipOnTriangle = next( left( corner ) );
    end
end
performEdgeFlip( flipOnTriangle );
while (!Q.empty( )) do
    $(v, e)$ = Q.remove();
    initial = $v$;
    current = initial;
    f = t( current );
    //Process the vertex's star by Edge Split
    operations.
    repeat
        if ( processedTriangles[ f ] == FALSE and
        (f.sublevel == 1 or subLevel == 2) ) then
            performSplit( current,
            processedTriangles, Q );
        end
        aux = right( current );
        current = next( aux );
        f = t( current );
    until (initial == corrente);
    performEdgeFlip( e );
end
performEdgeFlip( cornerToFlip );
```

Algorithm 11 is responsible for sorting the triangles by increasing level in order to realize the preprocessing step and initialize all variables. Next, the algorithm traverses the sorted list of triangles, subdividing all those that have not been processed. When it must refine a sublevel-0 triangle, it initiates the refinement from the first corner of the triangle. Otherwise, it starts the refinement from the corner associated with the highest-level vertex. This is sufficient to guarantee the correctness of the refinement process.

```
Algorithm 11: meshRefine( trianglesToRefine )
  sortFacesByLevel( trianglesToRefine ) ;
  preprocessingTriangles( trianglesToRefine );
  init( trianglesToRefine, processedTriangles );
  for each triangle f in trianglesToRefine do
      if (f.sublevel == 0)) then
          refineTriangles( processedTriangles, $3 \cdot f$ );
      else
          find corner c of the vertex with the level equal
          to f.level + 1;
          refineTriangles( processedTriangles, c );
      end
  end
```

Notice that, the most expensive step of the ARTMe algorithm is the sort step. This implies that the ARTMe algorithm has a theoretical complexity given by O (k log k), where k is the number of triangles to be refined. As k is limited by n, where n is the total of triangles of the mesh, in the worst case the ARTMe algorithm will work on O (n log n). However, as we show in Sect. 7, this algorithm has a low constant, which is very efficient in practice.

## 4.5 Undo and redo

In certain types of applications, like geometric modeling, it is important to be able to undo/redo a set of steps. For this reason, we provide an undo/redo mechanism in the refinement algorithms.

An inefficient solution for this mechanism is to store the entire mesh at every refinement step. However, a better solution is to store in a stack all the topological operations performed by the refinement algorithm.

As mentioned in Sect. 4.1, the adaptive refinements is based on the Edge Split and Edge Flip operations. The proposed implementation of such operators and their inverses, described in Sect. 3, facilitates the construction of this mechanism, owing to the established conventions imposed on their code. Figures 27 and 28 illustrate the application of the inverse operations used in the refinement algorithms.

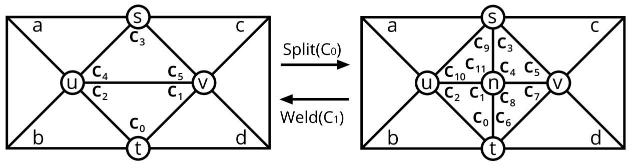


Fig. 27 Inverse of the Edge Split operation by using an Edge Weld operation, and vice-versa

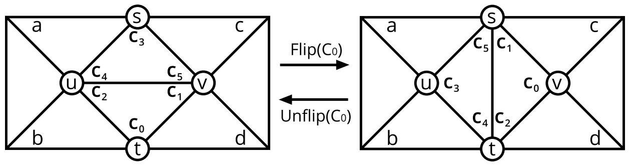


Fig. 28 Inverse of the Edge Flip operation by using an Inverse Edge Flip operation, and vice-versa

The Edge Weld operator removes a vertex and two triangles from the mesh when the removed vertex is not on the mesh boundary. The undo operation should keep the V, O, and G tables contiguous; i.e., it should not remove triangles or vertices from the middle of the tables. This condition is automatically satisfied in our implementation because the Edge Split operator always includes new triangles and new vertices at the end of tables V, O, and G, respectively. Consequently, when we reverse the sequence of steps to perform an undo, the Edge Weld operator will remove the triangles and vertices at the end of the corresponding tables, contiguously maintaining the V, O and G tables. This avoids the extra step of rearranging the table using a garbage collector to apply the undo/redo mechanism.

Table 1 shows the inverse operations that should be stacked to execute the undo/redo mechanism.

# 5 Implementation details

This section describes the implementation details that allow us to maintain the triangle and vertex levels and undo/redo information and, simultaneously, minimize memory consumption.

## 5.1 Triangle and vertex levels

As we mentioned in Sect. 3, the representation of the standard Corner-Table consists of two integer vectors, V and O, with dimensions equal to $3n$, where $n$ is the number of triangles. Thus, the representation of each triangle consists of six integers. Using a 32-bit integer, the Corner-Table consumes 24 bytes per triangle.

Table 1 Inverse Operations

| Operarion | Inverse operation |
|---|---|
| EdgeSplit ($c$) | EdgeWeld(next ($c$)) |
| EdgeWeld ($c$) | EdgeSplit (prev ($c$)) |
| EdgeFlip ($c$) | EdgeUnflip (next ($c$) ) |
| EdgeUnflip ($c$) | EdgeFlip (prev ($c$) ) |

The implementation of the adaptive refinement algorithm (ARTMe) requires storing the level of each vertex, and the level and the sub-level of each triangle in the mesh. The question is how much memory this information requires. Because only four sub-levels are possible, two bits per triangle are sufficient to store the sub-level information.

To store the level information, we must estimate an upper bound for the maximum number of levels a mesh can contain. We assert that 63 is a good estimate for this maximum and, consequently, six bits per triangle can store the level information.

The argument to justify our estimate is that the size of a triangle edge in level 63 is $2^{63}$ smaller than the size of its first ancestor. The presence of both sizes in the same mesh can cause precision problems.

From the above discussion, eight bits or one byte is sufficient to store the level and sub-level information of each triangle. For simplicity, we also use one byte to store the vertex-level information, yielding one byte per triangle plus one byte per vertex. Given that the number of vertices is approximately half the number of triangles [16], the amount of memory needed to perform the adaptive refinement is 25.5 bytes per triangle. This statement is equivalent to saying that 6.375 rpt (references per triangle) are necessary to perform the adaptive refinement on the Corner-Table.

### 5.2 Undo and redo information

To perform the undo and redo operations, it is necessary to store two other pieces of information: which corner should be used, and which operation must be applied using that corner.

A simple method is to create a structure with two integers, one to store the corner and the other to store the operation. However, even when using a 32-bit integer for the first one and an 8-bit integer for the last one, because of memory alignment issues [30], the structure will actually use 64 bits, or eight bytes. This is equivalent to using two 32-bit integers. In doing so, we use a 32-bit integer to store the operation type, which only needs two bits for a complete representation.

In our implementation, we decided to use a single integer to store the corner and the operation information. We use the two least significant bits to store the operation, and the other 30 bits to store the corner index. An advantage of using a single integer is that we only store four additional bytes for each operation instead of eight bytes. However, when the undo/redo is enabled, this approach decreases the maximum number of triangles in the mesh to $\lfloor 2^{30}/3 \rfloor$. If a larger number of triangles is necessary, the integer size can be changed from 32 to 64 bits to represent larger triangle meshes.

## 6 Special triangulations

The method presented in this paper can easily be adapted to produce common triangulations that use one-to-four refinement. In this Sect., we show how to generate the simple [2], Red–Green [3], and incremental [27] triangulations using our method.

### 6.1 Simple triangulation

As presented in Sect. 2.1, Amresh, Farin, and Razdan proposed a simple triangulation method that refines a triangle into two, three, or four new triangles, depending on the number of cracks [2].

The three simple triangulation templates shown in Fig. 5 are automatically generated by our method:

- One Crack: This is a trivial case, as illustrated in Fig. 18.
- Two Cracks: As the refinement algorithm processes triangle by triangle, after refining the first triangle, the mesh will be similar to the mesh on the bottom of Fig. 5a. During the refinement of the second triangle, in order to avoid a T-vertex, the refinement algorithm will use the O-vertex triangulation, connecting the new vertex to the opposite vertex, producing a resulting mesh equal to the mesh on the bottom in Fig. 5b. Depending on the refinement order, a symmetric refinement on middle triangle can be generated. Figure 29 illustrates the two possibilities of resulting meshes.
- Three Cracks: After refining the two triangles, the mesh will be similar to the mesh in Fig. 29a. When refining the last triangle, the refinement algorithm will use O-vertex triangulation to avoid a T-vertex, producing the mesh shown in Fig. 30a. At this point, the refinement algorithm will automatically identify that only an Edge Flip operation is necessary to complete the refinement operation, and then execute it, resulting in the mesh in Fig. 30b.

In fact, the ARTMe algorithm works like simple triangulation when the triangles involved are in the same refinement level.

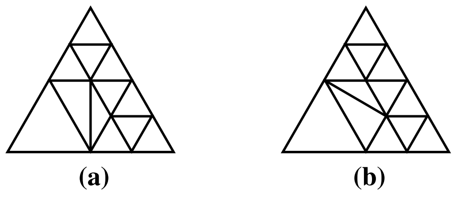


**Fig. 29** Two possible refinements on the middle triangle that can be generated depending on the refinement order

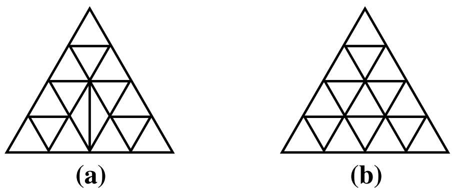


**Fig. 30** **a** Mesh after refining three triangles. **b** Final mesh after the Edge Flip operation

### 6.2 Red–Green triangulation

The Red–Green triangulation method is powerful because it produces a mesh with no T-vertices, using few triangles.

The Red–Green triangulation algorithm proposed in [3] solved the cracks after the refinement process and, during this step, guaranteed a good mesh gradation. When an edge has two or more cracks, the red rule is used, and the triangle is refined into four new triangles. This step is recursively applied. We can observe that more than one crack can be generated in an edge when the difference between the triangle refinement level is greater than one.

The ARTMe algorithm does not generate any cracks during the process because it solves them using the Edge Split operator; this process is equivalent to the green rule. During preprocessing, the ARTMe algorithm guarantees a good gradation mesh by including neighboring triangles with a refinement level lower than the current triangle to be refined. This is necessary because, once this triangle is refined, the level difference between the triangles will be greater than one. We can observe that it is equivalent to applying the red rule.

Nonetheless, the ARTMe algorithm does not generate the Red–Green triangulation if it allows sublevel-3 triangles, as does the simple triangulation method. However, one can avoid this type of triangle if one desires to generate a triangulation equivalent to Red–Green triangulation.

There are two ways to prevent sublevel-3 triangles:

1. Recursively refine all sublevel-3 triangles in a post-processing step.
2. Add triangles to the input in a preprocessing step to guarantee that the triangulation generated by ARTMe does not have any sublevel-3 triangles.

The first one is simple to implement. However, it is inefficient because it is necessary to traverse the mesh looking for that type of triangle in each step. Therefore, the second one is more appropriate.

To implement the second option, we need to refine the triangle that generates sublevel-3 triangles. There are only two ways to produce this type of triangle: refining a sublevel-1 or 2 face without refining the neighboring sublevel-2 or 1 face, respectively, and when two edges of the same triangle are refined, as shown in Fig. 29.

To solve the first option, it is sufficient to always add sublevel-1 and 2 triangles together, whenever one of them is refined. To solve the second one, it is sufficient to include the triangle with two cracks for refinement. With this extra preprocessing step to avoid sublevel-3 triangles, the ARTMe algorithm can generate the Red–Green triangulation.

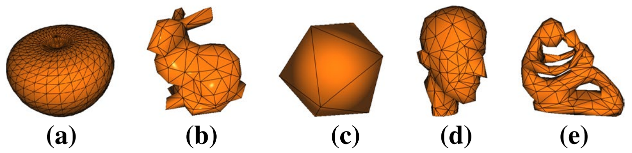


**Fig. 31** Models used in efficiency tests. **a** Apple. **b** Bunny. **c** Icosahedron. **d** Planck. **e** Fertility

### 6.3 Incremental triangulation

As shown in Sect. 2.1, the incremental triangulation proposed by [27] simultaneously produces a good gradation and restricted mesh. As illustrated in Fig. 8, this triangulation method expands the selected area and refines the new area, using simple triangulation to remove cracks. As simple triangulation is automatically induced by the ARTMe algorithm (Sect. 6.1), the ARTMe algorithm automatically generates the incremental triangulation when it is used to refine the expanded area.

## 7 Results

This section presents some examples to validate the proposed method and our implementation. To evaluate the efficiency of the proposed strategy in current computers, we used, for all tests, a simple desktop computer with an Intel(R) Core(TM)2 Quad processor with 2.66-GHz CPUs and 4 GB of 800-MHz DDR2 memory. To avoid memory-cache fluctuations, each reported time includes running the algorithm ten times and averaging the results.

### 7.1 Efficiency results

To illustrate the efficiency of the adaptive refinement algorithm, we used five different models, shown in Fig. 31. The focus here is on the efficiency of the topological operations and, for this reason, the position of the new vertices is always created in the mid-point of the corresponding edge.

**Table 2** The ARTMe algorithm can perform, on average, 1,298,554 refinements per second in the worst case, and 2,904,474 in the best case

| Model | Unconnected components | | Single connected component | |
|---|---|---|---|---|
| | Number of triangles | Refinements per second | Number of triangles | Refine-ments per second |
| Apple | 13,404,570 | 1,280,372 | 14,950,312 | 2,867,170 |
| Bunny | 13,582,600 | 1,271,410 | 14,972,944 | 3,043,840 |
| Icosahedron | 11,990,750 | 1,255,499 | 11,131,566 | 2,866,037 |
| Planck | 10,853,139 | 1,349,468 | 13,281,360 | 2,959,966 |
| Fertility | 10,410,462 | 1,336,022 | 13,218,120 | 2,785,358 |

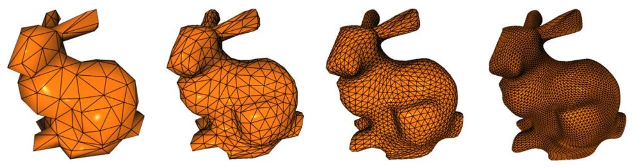


**Fig. 32** Four levels of global refinement. All new vertices are positioned using the Modified Butterfly method

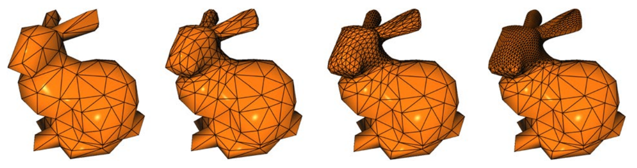


**Fig. 33** Four levels of adaptive refinement using incremental triangulation with $r = 1$. New vertices are positioned using the Modified Butterfly method

The adaptive refinement algorithm test uses 18% of the triangles at each refinement step. These triangles are selected in two ways: (1) randomly, to simulate the worst case when many unconnected components must be refined; and (2) in a single connected component to simulate the best-case scenario.

The refinement rate present in the tables is calculated as:

$$\mathrm{rps} = \frac{\text{total triangles} - \text{initial triangles}}{\text{total time}} \tag{6}$$

where the time is measured in second.

Finally, Table 2 reports the tests for the adaptive refinement algorithm. Multiple function calls to subdivide unconnected regions result in overhead. To prove this point, we reran the test on a single connected component. Nonetheless, the ARTMe algorithm achieves a high performance; once, it generated more than one million triangles per second, even in the worst case. In the best case, the performance increased to 2.9 million refinements per second.

## 7.2 Visual quality results

Figure 32 illustrates four different levels of global refinement of the Bunny model. In this example, all triangles are refined in each step; thus, all triangles are in the same refinement level, and it is not necessary to remove any cracks. Moreover, the new vertices are positioned using the Modified Butterfly method [33].

Figure 33 illustrates the adaptive refinement of the same model. In this example, the triangles on the Bunny's head are refined, and cracks are avoided using incremental triangulation [27]. In this method, the selected area for refinement is expanded, and triangles on the r-ring of the selected area are included in the list of triangles to be refined. The cracks are avoided by O-vertex triangulation. This is sufficient to guarantee a good gradation and restricted triangulation.

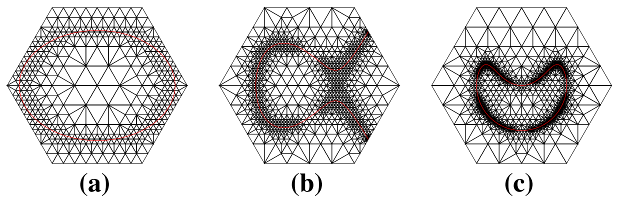


**Fig. 34** Adaptive polygonal approximation of implicit curves using a Red–Green triangulation. In **a** $x^2/3 + y^2/1.5 = 1; F = 2.50, G = 1.00$ are used. In **b** $y^2 - x^3 + x = 0.5; F = 0.50, G = 0.15$ are used **c** $(y - x^2 + 1)^4 + (x^2 + y^2)^4 = 1; F = 0.50, G = 0.15$ are used

Figure 34 illustrates the adaptive polygonal approximation of implicit curves [21]. Inside each triangle, the curve is approximated by a straight line. The mesh is adaptively refined to guarantee a maximum threshold error to the function (F) and gradient (G). Therefore, curved regions require more triangles. Interval arithmetic is used to measure these errors [24].

The next example from [14] Section F.2 presents a simple, steady-state, 2D heat-conduction model for a homogeneous geological layer, subjected to a fixed surface temperature (0 °C), and insulated on the sides. The basal heat flow from the lithosphere is discontinuous at the center of the layer.

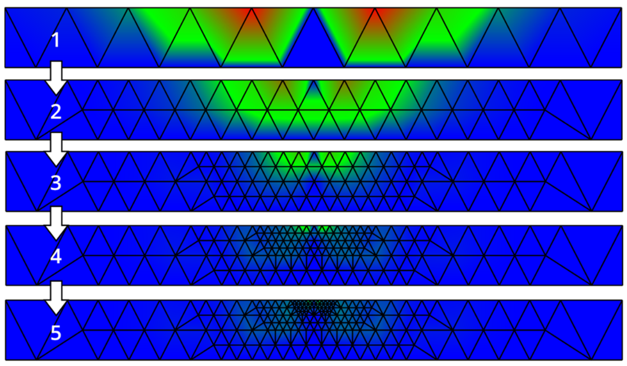


Fig. 35 Results of the error between the simulation and analytic method. The triangle mesh was generated iteratively, to reach a maximum error equal 1 °C

The bottom left part of the layer is subjected to a 50 mW/m$^2$ heat flow and the bottom right to 100 mW/m$^2$.

Figure 35 shows the difference between the temperature values calculated using the finite elements method, and the analytical solution. As expected, the errors are bigger at the vertical discontinuity. To decrease the maximum error, we interactively ran the triangle-refining method with an error greater than 1 °C. The sequence of meshes is shown in Fig. 35, from top to bottom.

## 7.3 Triangulation methods comparison

As discussed in Sect. 6.2, Red–Green triangulation can be obtained from the ARTMe algorithm by eliminating sublevel-3 triangles. This process requires that the sublevel-1 or 2 triangles be refined before the sublevel-3 triangle can be refined; thus, the number of triangles in the mesh increases.

Therefore, the ARTMe algorithm generates a mesh with fewer triangles than the Red–Green triangulation.

Figure 36 illustrates an adaptive polygonal approximation of an implicit curve, using ARTMe triangulation (Fig. 36a), Red–Green triangulation (Fig. 36b), and the incremental triangulation with $r = 1$ (Fig. 36c).

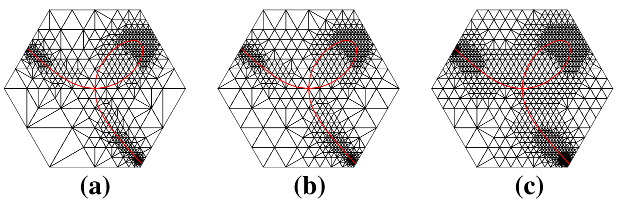


Fig. 36 Adaptive polygonal approximation of the implicit curve $x^3 - 2x^2 + y^3 = 0$ using **a** ARTMe **b** Red–Green, and **c** incremental triangulation with $r = 1$. In all examples, $F = 0.80$ and $G = 0.80$. **a** 1109 triangles. **b** 1215 triangles. **c** 2017 triangles

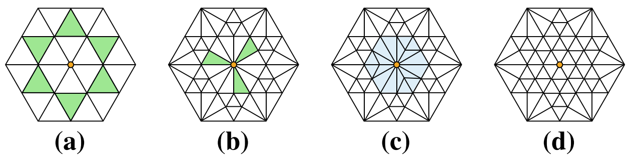


Fig. 37 **a** The green triangles are selected to be refined. **b** The valence of the central vertex is doubled. At this point, no operation can increase the vertex valence. The green triangles are selected to be refined. **c** The triangles selected at the previous step do not increased the vertex valence. **d** After refining the triangles, the original vertex valence is restored

Refining sublevel-3 triangles generates a slight difference in the number of triangles between the ARTMe and Red–Green triangulations (1109 versus 1215). As the number of different refinement levels increases, this difference tends to grow, once sublevel-3 triangles are discovered in the transition regions.

As the incremental triangulation expands the selected area to refine, it presents a considerably higher number of triangles than the ARTMe and Red–Green triangulations, as was expected.

## 7.4 Vertex valence control

As discussed in Sect. 4.1, the one-to-four refinement operator temporarily increases the O-vertex's valence by one. In this section, we will show that the ARTMe algorithm will doubles this valence in the worst case.

In the worst case, every triangle at the vertex's star is split, and the vertex is used as an O-vertex to avoid a T-vertex. As the vertex valence increases by one for each triangle from its star, the greatest possible valence is twice the original.

To illustrate this point, consider the Fig. 37. In Fig. 37a, the green triangles are selected to be refined. Note that the central vertex will be used as a T-vertex by all selected triangles. After the refinement process, the vertex valence changes from six to twelve, as in Fig. 37b.

From the mesh in Fig. 37b, it is impossible to increase the valence. Any refinement operation will maintain or decrease the vertex valence. The refinement of the green triangles in Fig. 37b generates the mesh in Fig. 37c. We can observe that from this point on, it is impossible to refine any triangle in this region without decreasing the vertex valence. Edges that are connected to the central vertex will be flipped, to complete the refinement. These Edge Flip operations decrease the vertex valence.

After refining all triangles at the central vertex star, the original valence is restored, as illustrated in Fig. 37d.

## 8 Conclusions

We presented a new efficient array-based algorithm for the adaptive refinement of triangle meshes. This algorithm, called ARTMe, is versatile enough to generate standard triangulations, e.g., Red–Green triangulation and incremental triangulation, through simple adjustments in its preprocessing step.

We also determined that the ARTMe algorithm could be implemented using Stellar operators in the Corner-Table, a compact array-based topological data structure, to support the undo and redo mechanism without needing garbage collection operations. Moreover, the ARTMe algorithm automatically produced a triangle mesh with good topological properties, as well as vertex-valence control and gradation control. In other words, the ARTMe algorithm preserved the mesh quality in the refinement process.

As the ARTMe algorithm used one-to-four refinement, it had the desirable property of generating new vertices with valence six, while the valences of the old vertices did not change. However, this operator increased the number of triangles by a factor of four, instead of three, as in $\sqrt{3}$-subdivision, or by a factor of two, as in 4–8 subdivision. Despite this constraint, the one-to-four refinement operator can work in any mesh structure, as opposed to the 4–8 subdivision, and it does not require specific and sophisticated boundary rules, as opposed to the $\sqrt{3}$-subdivision refinement operator.

Using the Corner-Table to replace the Half-Edge data structure reduced the memory consumption of the required topological information for mesh refinement by a factor of 2.75.

In concluding, the proposed algorithm is simple and capable of providing an efficient implementation of other standard triangulations with low memory consumption and mesh-quality preservation.

**Acknowledgements** The authors would like to thank CNPQ for partially supporting this research.

## References

1. Aleardi L, Devillers O, Rossignac J (2012) Esq: Editable squad representation for triangle meshes. In: 25th SIBGRAPI conference on graphics, patterns and images (SIBGRAPI), 2012, pp. 110–117. https://doi.org/10.1109/SIBGRAPI.2012.24
2. Amresh A, Farin G, Razdan A (2003) Adaptive subdivision schemes for triangular meshes. In: Farin G, Hamann B, Hagen H (eds) Hierarchical and geometrical methods in scientific visualization, mathematics and visualization. Springer, Berlin, pp 319–327
3. Bank RE, Sherman AH, Weiser A (1983) Some refinement algorithms and data structures for regular local mesh refinement. Sci Comput Appl Math Comput Phys Sci 1:3–17
4. Baumgart BG (1975) A polyhedron representation for computer vision. AFIPS Natl Comput Conf 44:589–596
5. Botsch M, Steinberg S, Bischoff S, Kobbelt L (2002) OpenMesh: a generic and efficient polygon mesh data structure. In: OpenSG Symposium 2002
6. Cashman TJ (2012) Beyond catmull clark? a survey of advances in subdivision surface methods. Comput Graph Forum 31(1):42–61. https://doi.org/10.1111/j.1467-8659.2011.02083.x
7. Catmull E, Clark J (1978) Recursively generated b-spline surfaces on arbitrary topological meshes. Comput Aided Design 10(6):350–355. https://doi.org/10.1016/0010-4485(78)90110-0
8. Caumon G, Collon-Drouaillet P, De Veslud CLC, Viseur S, Sausse J (2009) Surface-based 3d modeling of geological structures. Math Geosci 41(8):927–945
9. Doo D, Sabin M (1978) Behaviour of recursive division surfaces near extraordinary points. Comput Aided Design 10(6):356–360. https://doi.org/10.1016/0010-4485(78)90111-2
10. Dyn N, Levine D, Gregory JA (1990) A butterfly subdivision scheme for surface interpolation with tension control. ACM Trans Graph 9(2):160–169. https://doi.org/10.1145/78956.78958
11. Gurung T, Laney D, Lindstrom P, Rossignac J (2011) Squad: compact representation for triangle meshes. Comput Graph Forum 30(2):355–364. https://doi.org/10.1111/j.1467-8659.2011.01866.x
12. Gurung T, Luffel M, Lindstrom P, Rossignac J (2011) Lr: compact connectivity representation for triangle meshes. ACM Trans Graph 30(4):67:1–67:8. https://doi.org/10.1145/2010324.1964962
13. Gurung T, Luffel M, Lindstrom P, Rossignac J (2013) Zipper: a compact connectivity data structure for triangle meshes. Comput Aided Design 45(2):262–269. https://doi.org/10.1016/j.cad.2012.10.009 (Solid and Physical Modeling 2012)
14. Hantschel T, Kauerauf AI (2009) Fundamentals of basin and petroleum systems modeling. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-72318-9
15. Hatipoglu B, Ozturan C (2015) Parallel triangular mesh refinement by longest edge bisection. SIAM J Sci Comput 37(5):C574–C588
16. Hjelle Ö, Dæhlen M (2006) Triangulations Appl (Math Vis). Springer, New York
17. Kähler K, Haber J, Seidel HP (2001) Geometry-based muscle modeling for facial animation. Graph Interface 2001:37–46
18. Kobbelt L (2000) $\sqrt{3}$-subdivision. In: Proceedings of the 27th annual conference on computer graphics and interactive techniques, SIGGRAPH ’00, pp. 103–112. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. https://doi.org/10.1145/344779.344835
19. Lewiner T, Lopes H, Medeiros E, Tavares G, Velho L (2010) Topological mesh operators. Comput Aided Geom Design 27(1):1–22. https://doi.org/10.1016/j.cagd.2009.08.004
20. Loop CT (1987) Smooth subdivision surfaces based on triangles. Department of Mathematics, University of Utah
21. Lopes H, Oliveira JB, de Figueiredo LH (2002) Robust adaptive polygonal approximation of implicit curves. Comput Graph 26(6):841–852. https://doi.org/10.1016/S0097-8493(02)00173-5
22. Mallet JLL (2002) Geomodelling. Applied Geostatistics. Oxford University Press Inc, Oxford
23. Mantyla M (1988) Introduction to solid modeling. W. H. Freeman & Co., New York
24. Moore RE, Kearfott RB, Cloud MJ (2009) Introduction to interval analysis. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (**ISBN 0898716691, 9780898716696**)
25. Newman MHA (1926) On the foundations of combinatorial analysis situs. In: Proceedings of the royal academy, pp. 610–641
26. Pachner U (1991) P.l. homeomorphic manifolds are equivalent by elementary shellingst. Eur J Comb 12(2):129–145. https://doi.org/10.1016/S0195-6698(13)80080-7

27. Pakdel HR, Samavati FF (2007) Incremental subdivision for triangle meshes. Int J Comput Sci Eng 3(1):80–92. https://doi.org/10.1504/IJCSE.2007.014467
28. Poincaré H (1893) Sur la géneralisation d'un théoréme d'Euler relatif aux poliédres 117(117):437–464
29. Rossignac J (2001) 3d compression made simple: Edgebreaker with zip&wrap on a corner-table. In: Proceedings of the international conference on shape modeling and applications, SMI '01, pp. 278. IEEE computer society, Washington, DC, USA. http://dl.acm.org/citation.cfm?id=882486.884089
30. Tanenbaum AS, Austin T (2005) Structured computer organization, 5th edn. Prentice-Hall, Inc., Upper Saddle River (**ISBN 0131485210**)
31. Velho L, Zorin D (2001) 48 subdivision. Comput Aided Geom Design 18(5):397–427. https://doi.org/10.1016/S0167-8396(01)00039-5 (Subdivision Algorithms)
32. Vieira AW, Lewiner T, Velho L, Lopes H, Tavares G (2004) Stellar mesh simplification using probabilistic optimization. Comput Graph Forum 23(4):825–838. https://doi.org/10.1111/j.1467-8659.2004.00811.x
33. Zorin D, Schröder P, Sweldens W (1996) Interpolating subdivision for meshes with arbitrary topology. In: Proceedings of the 23rd annual conference on computer graphics and interactive techniques, SIGGRAPH '96, pp. 189–192. ACM, New York, NY, USA. https://doi.org/10.1145/237170.237254
34. Zorin D, Schröder P, Sweldens W (1997) Interactive multiresolution mesh editing. In: Proceedings of the 24th annual conference on computer graphics and interactive techniques, SIGGRAPH '97, pp. 259–268. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. https://doi.org/10.1145/258734.258863