RGB Subdivision

Enrico Puppo, *Member*, *IEEE*, and Daniele Panozzo

Abstract—We introduce the *RGB subdivision*, an adaptive subdivision scheme for triangle meshes, which is based on the iterative application of local refinement and coarsening operators, and generates the same limit surface of the Loop subdivision, independent of the order of application of local operators. Our scheme supports dynamic selective refinement, as in Continuous Level Of Detail models, and it generates conforming meshes at all intermediate steps. The RGB subdivision is encoded in a standard topological data structure, extended with few attributes, which can be used directly for further processing. We present an interactive tool that permits to start from a base mesh and use RGB subdivision to dynamically adjust its level of detail.

Index Terms—Triangle meshes, subdivision, level of detail.

1 Introduction

Subdivision surfaces are becoming more and more popular in computer graphics and CAD. During the last decade, they found major applications in the entertainment industry [1] and in simulation [2]. Several solid modelers, both commercial and open source, now support modeling based on subdivision [3], [4], [5], [6]. From the point of view of users, subdivision surfaces come midway between polygonal meshes and NURBS, getting many advantages from both worlds. They allow a designer to model a shape on the basis of a relatively simple control net, which can be handled as freely as a polygonal mesh, while automatically generating either a finer mesh at the desired level of detail (LOD) or a smooth limit surface.

Although the natural approach to modeling based on subdivision is coarse to fine, advances in reverse subdivision suggest that a fine-to-coarse approach can also be undertaken [7], [8], [9], [10], [11]. Direct and reverse subdivision may thus be used together, to take any mesh and automatically generate a whole hierarchy of LODs, coarser as well as more refined than the base mesh. Such a hierarchy, however, will contain just models at uniform resolution, i.e., having the same LOD in all parts of an object. In order to become really competitive with polygonal modeling, subdivision modeling should also support *selective refinement*, i.e., the possibility to vary LOD smoothly across a mesh and dynamically through time. To this aim, it is necessary to combine different levels of subdivision in the context of a mesh, without losing consistency with an underlying subdivision scheme (see Fig. 1). This is the central issue investigated in this paper.

1.1 Motivation

Most often, subdivision is applied up to a certain level and the resulting mesh is used for further processing [12]. Even when users are interested in rendering the limit surface, subdivided meshes can be useful in intermediate computations. For instance, physical engines for animation, as well as system solvers for the finite-element methods, work on polygonal meshes with a limited budget of cells.

It is often desirable to refine different parts of the mesh at different LODs. For instance, the design of characters for videogames is constrained by a certain budget of polygons. More polygons will be used in detailed areas and in the proximity of joints, while rigid parts will be modeled with fewer polygons. Similar arguments apply to domain discretization for the finite-element methods. Manually adjusting the LOD of the different parts of a mesh may be a tedious task, unless sophisticated tools to control LOD are made available. This sort of mechanism is customary in Continuous LOD (CLOD) applied to free-form mesh modeling [13]. A mesh at intermediate LOD can be modified online through selective refinement in either way, by refining some parts of it while other parts may be coarsened. To this aim, refinement and coarsening operations must be local and easily reversible.

For subdivided meshes, adaptivity implies that cells at different levels of subdivision are combined in the context of a single mesh. However, classical subdivision schemes are based on the application of recursive patterns that act uniformly over the whole surface. For instance, the popular Loop [14] and butterfly [15] schemes for triangle meshes are based on recursive one-to-four triangle split (see Fig. 2), which gives nonconforming meshes when applied adaptively at different levels of subdivision (see Fig. 3a).

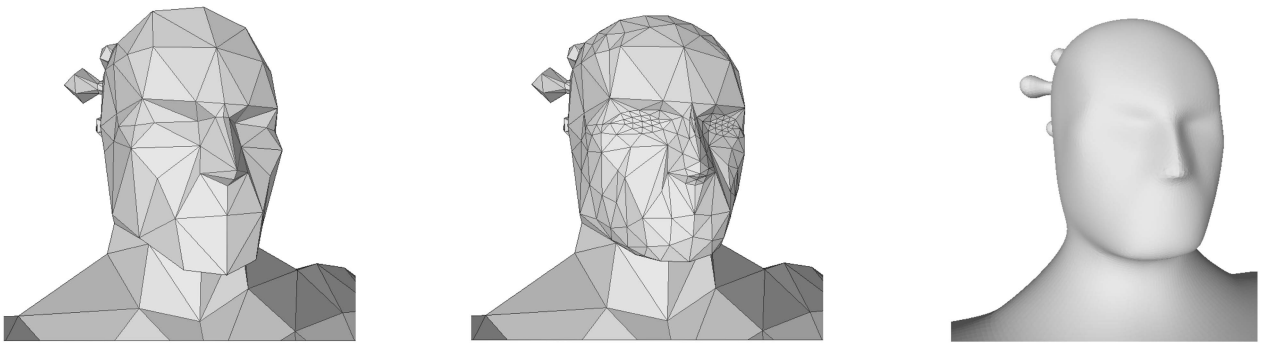
1.2 Contribution

We introduce *RGB subdivision*, an adaptive subdivision scheme for triangle meshes, which is based on the iterative application of local refinement and coarsening operators. Our scheme generates the same limit surface of the Loop subdivision, independently on the order of application of local operators, and it supports dynamic selective refinement and generates conforming meshes at all intermediate steps. The main contributions of this paper are the following:

1. We define local operators for both refining and coarsening a subdivision mesh of triangles by inserting/deleting one vertex at a time.

• *The authors are with the Department of Computer and Information Sciences, University of Genova, Via Dodecaneso 35, 16146 Genova, Italy. E-mail: puppo@disi.unige.it, daniele.panozzo@gmail.com.*

Manuscript received 21 Dec. 2007; revised 9 May 2008; accepted 17 June 2008; published online 25 June 2008.
Recommended for acceptance by J. Stam.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2007-12-0186.
Digital Object Identifier no. 10.1109/TVCG.2008.87.



(a) (b) (c)

Fig. 1. (a) A polygonal model is selectively refined through RGB subdivision, (b) by increasing LOD in the parts representing eyes, nose, mouth, and the top of the head. The limit surface of the RGB subdivision is coincident with that of the (c) Loop subdivision. Model courtesy of Silent Bay Studios (http://www.silentbaystudios.com).

2. On the basis of such operators, we define the family of *RGB triangulations* and we study them as purely combinatorial structures. We show that they are highly adaptive, since they span all possible meshes obtained by combining triangles in a given set.
3. We give a dynamic selective refinement algorithm for RGB triangulations, with the same features of the algorithms developed in the context of CLOD models.
4. Next, we develop *RGB subdivision* by endowing RGB triangulations with the rules of Loop subdivision. We derive a multipass formula to compute control points of vertices correctly and we develop a mechanism to keep an RGB subdivision consistent throughout all steps of selective refinement.
5. We describe a data structure for RGB subdivision, which does not need to store any hierarchy and extends a standard topological data structure with a moderate overhead.
6. We present an application built upon RGB subdivision and the selective refinement algorithm, which allows a user to adjust LOD locally, by interacting with the mesh through a brush tool.

The rest of this paper is organized as follows: In Section 2, we discuss related work. In Section 3, we give the necessary background. In Section 4, we introduce RGB triangulations, and in Section 5, we describe the selective refinement algorithm working on them. In Section 6, we introduce RGB subdivision by describing how to set the position of control points during selective refinement, according to the Loop subdivision scheme. In Section 7, we describe the data structure, and in Section 8, we present our interactive tool to manage LOD on RGB subdivision. Finally, in Section 9, we make some concluding remarks.

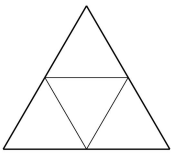


Fig. 2. The one-to-four triangle split pattern.

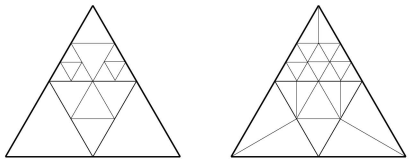
2 Related Work

2.1 Adaptive Subdivision

The literature on subdivision surfaces is quite extended. The interested reader can refer to [16] for a textbook, [12] for a tutorial, and [8] for a survey. Here, we will review only those works related to adaptive subdivision on triangle meshes.

Red-green triangulations were introduced in the context of finite-element methods [17] and have become popular in the common practice, as an empirical way to obtain conforming adaptive meshes from hierarchies of triangle meshes generated from one-to-four triangle split. They are built through a two-step procedure: first, apply one-to-four triangle split adaptively, and then subdivide some triangles further, to fix nonconforming situations (see Fig. 3). Depending on the underlying subdivision scheme, the geometry of vertices (control points), which lie on the transition between different levels of subdivision, may not correspond to that of the same vertices in a uniformly subdivided mesh. This fact may prevent the correctness of further subdivision or coarsening of a red-green triangulation, unless the subdivision process is repeated from scratch. This latter option is unwieldy, and it prevents incremental editing of LOD and may be not sustainable for online processing.

A variant of red-green triangulations was used in [11] to support multiresolution editing of meshes based on the Loop subdivision scheme. Adaptive meshes are computed by reverse subdivision, starting at the finest level and pruning overrefined triangles. Also, in this case, a restricted



(a) (b)

Fig. 3. Red-green triangulation: (a) a restricted nonconforming mesh obtained from adaptive one-to-four split is made conforming by (b) splitting some triangles further, depending on the level of their neighbors.

nonconforming mesh is computed first, which is fixed next by further bisection of some triangles. Correct relocation of vertices is treated by using a hierarchical data structure. Recently, another variant, called incremental subdivision, was presented in [18] for both the Loop and the butterfly schemes. In this case, the correct computation of geometry of control points is addressed by using a larger support area for refinement.

Based on seminal work of Forsey and Bartels [19], a method for hierarchical editing of triangular splines is proposed in [20], which is based on adaptive one-to-four subdivision, like in the first step of red-green triangulations. The nonconforming mesh is used just as a control grid and the continuity of the spline surface is guaranteed by satisfying consistency constraints across adjacent patches at different levels of subdivision.

In [21], the one-to-four triangle refinement scheme is decomposed into atomic local operations, called *quarks*, based on the popular *vertex split* operation [22]. In [23], a factorization of one-to-four triangle split into a sequence of edge split and edge swap operations is proposed, which forms a subset of the local operators we use in Section 4. A factorization of the Loop subdivision rule is also proposed, which makes it possible to compute the control points correctly through the sequence of local refinement operations.

The $\sqrt{3}$ subdivision [24] and the 4-8 subdivision [25] schemes are not based on the classical one-to-four triangle split operator. They are naturally adaptive, being both based on local conforming operators. The $\sqrt{3}$ subdivision alternates one-to-three triangle split at one level, with edge swap at the next level. The resulting triangles can be regarded as being of *green* and *blue* types in our terminology (see Section 4). A closed form solution of the subdivision rule permits to compute control points correctly for a vertex at any level, at the cost of some overrefinement. The 4-8 subdivision is based on edge split, as in our case, applied to a special case of triangle meshes, called *tri-quad meshes*. The correct position of control points is addressed and resolved also in this case with a certain amount of overrefinement. Only basic operations are investigated in [25].

2.2 CLOD Models

Also, the literature on CLOD models is very wide. The interested reader may refer to [13] for a book. Here, we review only some concepts and contributions that are relevant to the rest of this paper. Generally speaking, a CLOD model consists of a base mesh at coarse resolution, plus a set of local modifications that can be applied to the base mesh to refine it. Such modifications are usually arranged in a hierarchical structure, which consists of a directed acyclic graph (DAG) in the most general case. Meshes at intermediate LOD correspond to cuts in the DAG, and algorithms for selective refinement work by moving a front through the DAG and doing/undoing modifications that are traversed by this front. This general framework, as shown in [26], encompasses almost all CLOD models proposed in the literature and it can be applied to the hierarchies generated by $\sqrt{3}$ subdivision and 4-8 subdivision as well.

In [27], a CLOD model is introduced, which does not fit the above general framework, and achieves better adaptivity by using local modifications more freely than in previous models. In Section 4, we use the idea of transitive mesh space proposed in [27] to study the expressive power of RGB triangulations.

CLOD models can provide meshes at intermediate LOD, where detail can vary across the mesh and through time, at a virtually continuous scale and with fast procedures. The outer structure of the algorithm we propose for RGB triangulations in Section 5 is based on a popular scheme proposed first in [28].

There exist a few CLOD models based on subdivision patterns. The model proposed in [28] is based on the recursive bisection of right triangles. This rule is also used by several other authors and may be regarded as a subdivision scheme. It can be applied just to meshes obtained from regular grids (typically representing terrains), while its extension to more general triangle meshes is not straightforward. One generalization is given by 4-k meshes [29], which have in fact a strong relation with 4-8 subdivision [25].

3 Background

3.1 Triangle Meshes

A *triangle mesh* is a triple $\Sigma = (V, E, T)$, where V is a set of points in 3D space, called *vertices*; T is a set of triangles having their vertices in V and such that any two triangles of T either are disjoint or share exactly either one vertex or one edge (i.e., the mesh is inherently *conforming*); E is the set of edges of the triangles in T. Standard topological incidence and adjacency relations are defined over the entities of Σ.

We will assume to deal always with *manifold* meshes either with or without boundary, i.e., each edge of E is incident at either one or two triangles of T; and the *star* of a vertex (i.e., the set of entities incident at it) is homeomorphic either to an open disc or to a closed half-plane. Edges that are incident at just one triangle and vertices that have a star homeomorphic to a half-plane form the *boundary* of the mesh, and they are called *boundary edges* and *boundary vertices*, respectively. The remaining edges and vertices are said to be *internal*. A mesh with an empty boundary is said to be *watertight*. A mesh is *regular* if all its vertices have a valence of six. Vertices with a valence different from six are called *extraordinary*.

A *nonconforming mesh* is a structure similar to a mesh, in which triangles may violate the rule of edge sharing: there may exist adjacent triangles t and t' such that an edge of t overlaps just a portion of the corresponding edge of t'.

3.2 Loop Subdivision

The Loop subdivision [14] is an approximating scheme that converges to a C^2 surface if applied to a regular mesh. The subdivision pattern is *one-to-four triangle split*, as depicted in Fig. 2. The position $p^l(v)$ of a new vertex v introduced at level l of subdivision (called an *odd* vertex) is computed as a weighted sum of positions of vertices from the previous level (called *even* vertices), as depicted in Figs. 4a and 4c: If v splits an internal edge, then its position is given by

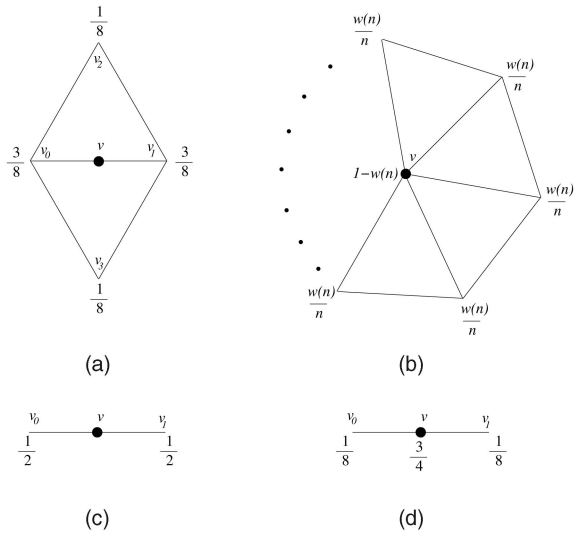


Fig. 4. The stencils used in the Loop subdivision scheme: (a) odd internal vertex, (b) odd boundary vertex, (c) even internal vertex, and (d) even boundary vertex. Numbers are weights assigned to vertices in the linear combination, n is the valence of the even vertex ($n = 6$ in the regular case).

$$p^l(v) = \frac{3}{8}p^{l-1}(v_0) + \frac{3}{8}p^{l-1}(v_1) + \frac{1}{8}p^{l-1}(v_2) + \frac{1}{8}p^{l-1}(v_3). \quad (1)$$

If v splits a boundary edge, then its position is given by

$$p^l(v) = \frac{1}{2}v_0 + \frac{1}{2}v_1. \quad (2)$$

Even vertices are relocated at each level l of subdivision, through a weighted sum of their position and the position of their one-ring neighbors at the previous level: If v is an internal vertex, it is relocated according to the following formula:

$$p^l(v) = (1-\alpha_n)p^{l-1}(v) + \frac{\alpha_n}{n}\sum_{i=0}^{n-1} p^{l-1}(v_i), \quad (3)$$

where

$$\alpha_n = \frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\left[\frac{2\pi}{n}\right]\right)^2, \quad (4)$$

and v_i for $i = 0, \ldots, n-1$ are the n neighbors of v (apart from extraordinary vertices in the base mesh, we always have $n = 6$). If v is a boundary vertex, it is relocated according to the following formula:

$$p^l(v) = \frac{3}{4}p^{l-1}(v) + \frac{1}{8}p^{l-1}(v_0) + \frac{1}{8}p^{l-1}(v_1). \quad (5)$$

Therefore, for each vertex v introduced at level l, there exists an infinite sequence of control points $p^l(v), p^{l+1}(v), \ldots, p^\infty(v)$ that define the positions of v at level l and all successive levels, $p^\infty(v)$ being its position on the limit surface. It is possible to show that any control point $p^k(v)$ for a vertex v introduced at level l, with $0 \le l < k$, can be computed directly just from the positions p^l of v and of all its neighbors at level l. In the Appendix, we derive a multipass closed form for computing directly control points at an arbitrary level.

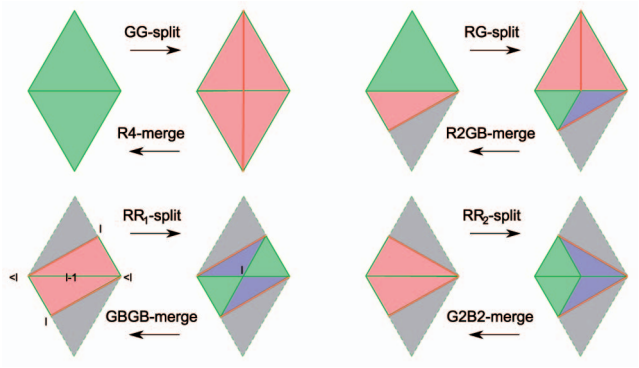


Fig. 5. Edge split and edge merge operators. Labels denote the level of vertices and edges. Gray triangles represent portions of triangulations spanned by parent triangles, which are not affected from the operation, and may have been refined further at arbitrarily many levels.

4 RGB Triangulations

RGB triangulations are defined as all those triangle meshes that can be built through iterative application of the operators for local modification depicted in Figs. 5 and 6, starting at a base mesh Σ_0. Note that the diagrams in such figures cover all possible topological configurations in local mesh subdivision and coarsening.

In this section, we introduce the combinatorial structure of RGB triangulations and the basic rules to manipulate them in a consistent way. In Section 4.1, we define local subdivision operators. The essential idea is that such operators subdivide a mesh by introducing one vertex at a time, they always produce conforming triangulations, and they can be controlled just on the basis of color and level codes. In Section 4.2, we introduce local coarsening operators, which reverse refinement operators, while in Section 4.3, we add one neutral operator. Section 4.4 has a more theoretical flavor: we define and study the transitive space of RGB triangulations, in order to show their expressive power and adaptivity; we also prove some results useful to warrant correctness of the selective refinement algorithm that will be described in Section 5.

All rules defined in this section are purely topological. Just for the sake of clarity, in the figures, we will use meshes composed of equilateral triangles, right triangles, and isosceles triangles to depict the three different types of triangles that may appear in an RGB triangulation. Actually, the shape of triangles is totally irrelevant in the subdivision process, while just level and color codes matter.

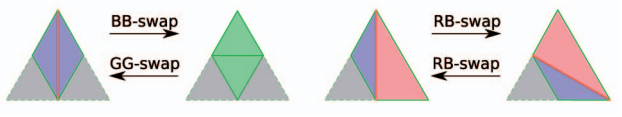


Fig. 6. Edge swap operators. Gray triangles represent portions of triangulations spanned by parent triangles, which are not affected from the operation, and may have been refined further at arbitrarily many levels.

4.1 Local Subdivision Operators

Consider a base mesh Σ_0. We assign level zero to all its vertices and edges and color green to all its edges. As a general rule, the level of a triangle is defined to be the lowest among the levels of its edges; and the color of a triangle is defined to be: green if all its edges are at the same level; red if two of its edges are at the same level l and the third edge is at level $l+1$; and blue if two of its edges are at the same level $l+1$ and the third edge is at level l. It follows that all triangles in the base mesh are green at level zero.

In the following, we define local subdivision operators that, when applied iteratively to Σ_0, will generate a conforming mesh where triangles will be colored green, red, and blue; edges will be colored green and red; and vertices, edges, and triangles will have different levels. Color and level codes allow us to control the application of subdivision operators on a local basis.

We say that an edge e at level $l \geq 0$ is *refinable* (i.e., it can split) if and only if it is green and its two adjacent triangles t_0 and t_1 are both at level l. In case of a boundary edge, only one such triangle exists. We split an edge e at level l, by inserting at its midpoint a new vertex at level $l+1$. The edges generated by the two halves of e are green and at level $l+1$. Note that these definitions of levels for vertices and green edges are fully compliant with those in the standard uniform subdivision based on the one-to-four triangle split pattern.

Splitting an edge e at level l also affects its incident triangles t_0 and t_1: each such triangle is split into two triangles by connecting the new vertex splitting e to the vertex of the triangle opposite to e. If the triangle is green, then the new edge splitting it will be red at level l; if the triangle is red, then the new edge will be green at level $l+1$. Blue triangles do not split (blue triangles at level l have their green edges at level $l+1$, thus they are not refinable by definition). By simple combinatorial analysis, we obtain the following variants of the edge split operator (see Fig. 5):

- **GG-split:** t_0 and t_1 are both green. The bisection of each triangle t_0 and t_1 at the midpoint of e generates two red triangles at level l. Each such triangle will have: one green edge at level l (the one common with old triangle t), one green edge at level $l+1$ (one half of e), and one red edge at level l (the new edge inserted to split t).
- **RG-split:** t_0 is green and t_1 is red. Triangle t_0 is bisected and edge e is split as above. The bisection of t_1 generates one blue triangle at level l and one green triangle at level $l+1$. The green triangle is incident at the green edge at level $l+1$ of old triangle t_1 and also its other two edges are at level $l+1$ (the edge inserted to subdivide t_1, and one half of e). The blue triangle is incident at the red edge of old triangle t_1 and has also two green edges at level $l+1$ (the edge inserted to subdivide t_1, and the other half of e).
- **RR-split:** t_0 and t_1 are both red. Triangles t_0 and t_1 are both bisected as triangle t_1 in the previous case and each of them generates the same configuration made of a blue triangle at level l and a green triangle at level $l+1$. This case may come in two variants: RR_1-split and RR_2-split. Each variant can be recognized by the cycle of colors of edges on the boundary of the diamond formed by t_0 and t_1: this may be either red-green-red-green for RR_1-split, or red-red-green-green for RR_2-split.

Fig. 7. The one-to-four triangle split of a green triangle is factorized into three edge split operations plus an edge swap operation.

Edge split operations applied to boundary edges will affect just one triangle. The resulting configuration depends only on the color of the triangle incident at e.

Edge split operators are not sufficient to factorize the one-to-four triangle split pattern by introducing one vertex at a time. We also need an edge swap operator, as depicted on the left side of Fig. 6, to get rid of blue triangles: **BB-swap** is applied to a pair of blue triangles at level l, which are adjacent along their red edge at level l. Such an edge is eliminated and the other diagonal of the quadrilateral formed by such two triangles is inserted. The new edge, as well as its two incident triangles, will be green at level $l+1$. Note that, by construction, one of the two new green triangles will have all three vertices at level $l+1$. Note also that just green edges can be split, while red edges are only swapped.

The sequence of three edge splits plus one edge swap that factorizes the one-to-four triangle split of a triangle is depicted in Fig. 7 and corresponds to that proposed in [23].

By simple combinatorial analysis, it would be easy to verify that this set of operators is closed with respect to the meshes obtained, i.e., if we start at an "all green" mesh Σ_0 at level 0 and we proceed by applying any legal sequence composed of the five operators above, all refinable edges in the resulting mesh can always be split by one of the four variants of edge split. Rather than proving this claim, in Section 4.4, we prove a more general result that also implies this fact.

4.2 Reverse Subdivision Operators

We define also local operators that invert edge split and edge swap on an RGB subdivision. Edge merge is the reverse operator of edge split and can be applied to triangles incident at vertices of valence four. The same cases depicted in Fig. 5 occur (modifications apply right to left in this case):

- **R4-merge** inverts GG-split;
- **R2GB-merge** inverts RG-split;
- **GBGB-merge** inverts RR_1-split;
- **G2B2-merge** inverts RR_2-split.

A little care must be taken in applying GBGB-merge in order to avoid inconsistencies. Referring to Fig. 5, note that the quadrilateral has two vertices at the same level l and two other vertices at a level lower than l. GBGB-merge must be performed by removing edges incident at the vertices of level l.

Similar rules apply to pairs of triangles along the boundary.

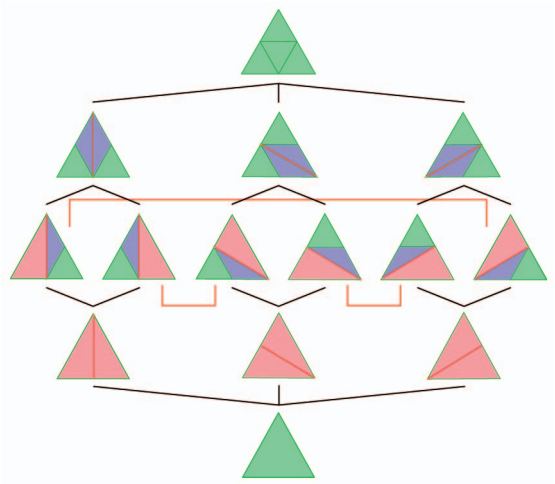


Fig. 8. The first few nodes of the transition space for a mesh formed by a single triangle. Here, we find all patterns that subdivide the triangle between level zero and level one. Arcs in black correspond to transitions through (upward) refinement operators and (downward) coarsening operators; arcs in red correspond to RB-swap operators.

GG-swap, which inverts BB-swap, can be applied to a pair of adjacent green triangles t_0 and t_1 at level $l > 0$, if one of them, say t_0, has all three vertices at level l. This condition is necessary and sufficient to guarantee that t_0 and t_1 have the same parent triangle t in the subdivision and t_0 is the central triangle obtained by subdividing t.

4.3 Neutral Operator

We finally introduce **RB-swap**, a reflexive operator that is neutral with respect to subdivision (i.e., it neither refines nor coarsens). RB-swap takes a pair formed by a red triangle and a blue triangle at the same level l of subdivision, which are adjacent along a red edge, and swaps the diagonal of the trapezoid formed by such a pair, thus obtaining another red-blue pair of triangles at level l (see Fig. 6). This operator may seem redundant. On the contrary, it is very important for both theoretical and practical reasons, as we will discuss in Section 4.4.

For each diagram in Figs. 5 and 6, the colored part is influenced by the corresponding operator, while the gray area refers to a portion of mesh that must exist but is not influenced by the operator and is not necessarily covered by a single triangle. In fact, the gray part may have been refined adaptively at arbitrarily many levels of subdivision.

4.4 The Transition Space of RGB Triangulations

We now have a set of 11 atomic operators: four split operators, four merge operators, and three swap operators. The family $\mathcal{RGB}_{\Sigma_0}$ of RGB triangulations subdividing base mesh Σ_0 is defined inductively as follows:

- Σ_0 is an RGB triangulation;
- If Σ is an RGB triangulation and Σ' is obtained from Σ by applying one of the 11 atomic operators, then Σ' is also an RGB triangulation.

Following the approach in [27], we define the *transition space* of $\mathcal{RGB}_{\Sigma_0}$ as a graph where

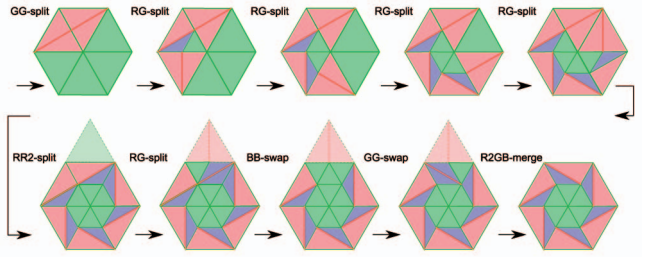


Fig. 9. A *fan* configuration. Without RB-swap: the fan is obtained from an "all green" mesh by a sequence of refinement operators followed by a GG-swap and an R2GB-merge; it cannot be simplified without using refinement operators. With RB-swap: the fan is obtained in a smaller number of steps by applying RB-swap right after the RR2-split and it can be reversed without using refinement operators.

- $\mathcal{RGB}_{\Sigma_0}$ is the set of nodes (where each mesh is taken as an atomic entity).
- There is an arc between two meshes Σ and Σ' if and only if it is possible to transform Σ into Σ' by applying just one atomic operator.

In Fig. 8, we show the initial fragment of transition space for a single triangle, which shows all possible ways to subdivide such triangle at levels zero and one of subdivision and all possible transitions among such configurations.

Note that the transition space is not a strict partial order, because of RB-swap operators. So, one may think that we would better define RGB triangulations without using such operator. In fact, a transition space defined without RB-swap would be a strict partial order, but it would also contain minimal elements different from Σ_0. For instance, if we do not use RB-swap, the "fan" configuration depicted in Fig. 9 becomes a minimal element in the transition space. There are also more practical reasons for using RB-swap. Consider for instance the "strip" configuration shown in Fig. 10. This configuration has been obtained from an "all green" mesh by applying a sequence of edge split operators. The only possible way to coarsen such a mesh without using RB-swap consists in reversing the refinement sequence. In other words, it is not possible to remove any vertex v introduced at an intermediate step, without also removing all vertices following it in the strip. On the contrary, as we will show in the following, any intermediate vertex can be removed by applying a single RB-swap followed by a merge, without affecting the other vertices of the mesh. In summary, RB-swap allows us to obtain monotone and more flexible sequences of refinement and coarsening operators.

For the sake of simplicity, in the rest of this section, we will assume Σ_0 to be watertight. Generalization of the following results to meshes with boundary is straightforward.

Let Δ_{Σ_0} be the set of all triangles that appear in meshes of $\mathcal{RGB}_{\Sigma_0}$, and let $\mathcal{T}_{\Sigma_0}$ be the set of all possible (conforming and watertight) triangle meshes that can be built by combining elements of Δ_{Σ_0}. Note that combination of triangles is arbitrary, provided that they match at common edges. We now show that all elements of $\mathcal{T}_{\Sigma_0}$ are RGB triangulations.

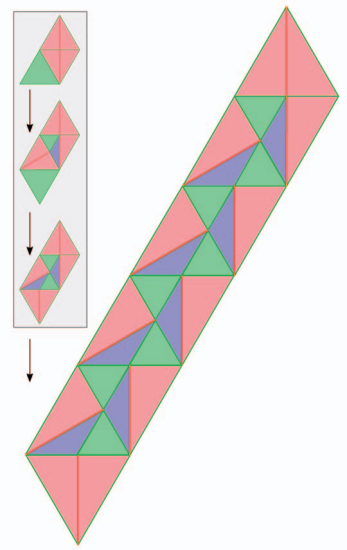


Fig. 10. This strip has been obtained from an "all green" strip by applying a GG-split (at its top end) followed by a sequence of RG-splits (proceeding downward). Without RB-swap, the only possible way to coarsen the strip is by reversing the refinement sequence. With RB-swap, followed by a merge operation, we can remove any intermediate vertex introduced during refinement, without affecting the other vertices.

Lemma 4.1. *The transition space of $\mathcal{RGB}_{\Sigma_0}$ spans $\mathcal{T}_{\Sigma_0}$.*

Proof. Let us first analyze the nature of triangles in Δ_{Σ_0}: since they come from meshes of $\mathcal{RGB}_{\Sigma_0}$, each such triangle t is endowed with a color and a level l. For $l > 0$, t must have been generated from one of the 11 local operators, and it must subdivide a parent triangle t' at level $l-1$. Moreover, along the edge(s) internal to t', triangle t can only be adjacent to other triangles that also subdivide t'. In other words, no matter how we combine triangles to form a mesh Σ of $\mathcal{T}_{\Sigma_0}$, if Σ contains t, then it must contain a group of triangles that subdivide t' exactly. Vertices of triangles in Δ_{Σ_0} also have a level: if a vertex v belongs to Σ_0, then its level is zero; otherwise, v has been generated splitting an edge at level $l-1$, thus its level is l. It is straightforward to see that all triangles incident at that vertex have a level greater than or equal to $l-1$.

Now, let Σ be a mesh of $\mathcal{T}_{\Sigma_0}$. Let us define the level m of Σ to be the maximum level of its vertices. Proof is by induction on m.

If $m = 0$, then Σ can be formed just from green triangles at level zero. Since Σ is watertight and all its triangles come from Σ_0, then we have necessarily $\Sigma \equiv \Sigma_0$, thus Σ is an RGB triangulation.

Now, let us assume all meshes of $\mathcal{T}_{\Sigma_0}$ up to level $m-1$ are RGB triangulations. Given Σ at level m, we know that the level of its green triangles is at most m, while the level of its red and blue triangles is at most $m-1$. We build another mesh Σ' at level $m-1$ as follows: we remove all green triangles at level m and all red and blue triangles at level $m-1$ from Σ; as a consequence, all vertices at level m have also been removed; this means that the holes left after removing such triangles can be filled exactly with green triangles at level $m-1$. Let us call this set of triangles Φ_{m-1}, which in fact contains the parents of triangles that we have removed. Now, since triangles of Φ_{m-1} also belong to Δ_{Σ_0}, then Σ' must belong to $\mathcal{T}_{\Sigma_0}$. Since we have removed all vertices at level m, Σ' is at level $m-1$, thus by inductive hypothesis it is an RGB triangulation.

Now, let us consider all vertices at level m that we have eliminated from Σ to obtain Σ'. By construction, they all lie on green edges at level $m-1$ that are shared by pairs of triangles of Φ_{m-1}. Thus, all such edges are refinable. Let us consider an arbitrary sequence of edge splits that insert such vertices back into Σ', generating another mesh Σ''. Mesh Σ'' has the same set of vertices of Σ and it coincides with Σ at all green triangles of level smaller than m and on all red and blue triangles of level smaller than $m-1$. In fact, the edge splits we have performed affect only the triangles of Φ_{m-1}. Now, each triangle of Φ_{m-1} has been split by one of the patterns depicted in the middle levels of Fig. 8 (where we now assume that the root triangle has level $m-1$). Let t be one triangle of Φ_{m-1}, and let us consider the two patterns decomposing t in Σ and in Σ''. Since we have introduced all and only those vertices that were removed, the two patterns may be different, but they subdivide the edges of t in the same way. By referring to Fig. 8 and comparing the two patterns, we have the following cases:

- If they subdivide just an edge of t, then they must be equal.
- If they subdivide two edges of t and they are different, then it is possible to obtain one from the other by applying an RB-swap.
- If they subdivide all three edges and they are different, then it is possible to obtain one from the other by applying a BB-swap and/or a GG-swap.

All operators listed above only affect triangles that subdivide t, so they can be carried out independently on all subdivisions of triangles of Φ_{m-1}. This means that we can obtain Σ from Σ'' through a sequence of local operators. Concatenating such a sequence with the sequence that transforms Σ' into Σ'', we have a sequence of local operators that transforms Σ' into Σ. Therefore, Σ is also an RGB triangulation. □

It is an open question whether or not the same set of triangulations can be generated by using just combinations of the first 10 operators, without using RB-swap. So, the set of operators we use is sufficient to generate the transition space, but we do not claim it to be minimal.

Concerning comparison with other known schemes, note that a uniform "all green" subdivision at any level belongs to $\mathcal{T}_{\Sigma_0}$; therefore, it is an RGB triangulation. Also, red-green triangulations belong to $\mathcal{T}_{\Sigma_0}$ and they are in fact a proper subset of triangulations. Red-green triangulations that obey the *two-neighbor rule* as defined in [17] will not contain blue triangles. Even in cases where such a rule is not applied, there exist RGB triangulations that cannot be obtained as red-green triangulations. A trivial example is a mesh made of four red triangles obtained from a pair of adjacent green triangles by applying GG-split (as in the upper left case of Fig. 5).

Next, we state some results useful to ensure that the selective refinement algorithm, described in Section 5, does not get stuck in configurations that cannot be either refined or simplified further.

Corollary 4.2. *Any RGB triangulation can be obtained from Σ_0 by applying a sequence of operators composed just of edge split and swap operators. Σ_0 can be obtained from any RGB triangulation by applying a sequence composed just of edge merge and swap operators.*

Proof. The first statement follows from the proof of Lemma 4.1 by considering the operators we have used to obtain Σ from Σ'. The second statement follows from the first one by considering that each split operator is inverted by a merge operator and each swap operator is inverted by a swap operator. □

Sequences used in Corollary 4.2 are not always monotone in the span space. In fact, refinement [coarsening] sequences to obtain meshes that contain configurations in the second upper row of Fig. 8 may require using GG-swap [BB-swap], which is actually a coarsening [refinement] primitive. On the other hand, such configurations are not really interesting: the decomposition of a parent triangle with a configuration that contains two green and two blue triangles is usually better substituted with the standard decomposition made of four green triangles. A mesh containing no configuration made of two blue triangles adjacent along a red edge will be called *stable*; otherwise, it will be called *unstable*. In our implementation of selective refinement, we will use unstable configurations just as transitions. We will perform refinement by using just subdivision operators and coarsening by using just reverse subdivision operators and RB-swap. During refinement, a BB-swap will be forced every time an unstable configuration arises. During coarsening, on the contrary, GG-swap and RB-swap will be used to locally modify the mesh in order to allow a vertex to be removed from a merge operator. The mesh just before applying the merge operator may be unstable, but it will become stable right after it.

We study next the local configurations corresponding to vertices that can be removed during coarsening. Let v be a vertex at level $l > 0$ in an RGB mesh. We say that v is *removable* if and only if all its adjacent vertices are at a level $\leq l$. Since v was introduced by splitting an edge at level $l-1$, by combinatorial analysis we have that the star of triangles surrounding it can have only 28 possible configurations, which are obtained by mirroring from the 18 configurations depicted in Fig. 11. For each such configuration, the graph in the figure provides a sequence of operators to remove v. Notice that BB-swap is necessary only if we start from one of the unstable configurations. Notice also that the local configurations at the end of sequences (i.e., after vertex removal) are all stable.

Corollary 4.3. *If an RGB mesh Σ is stable, then*

1. *Σ can be obtained from Σ_0 by a sequence made just of refinement operators and RB-swaps;*
2. *Σ_0 can be obtained from Σ by a sequence of just coarsening operators and RB-swaps.*

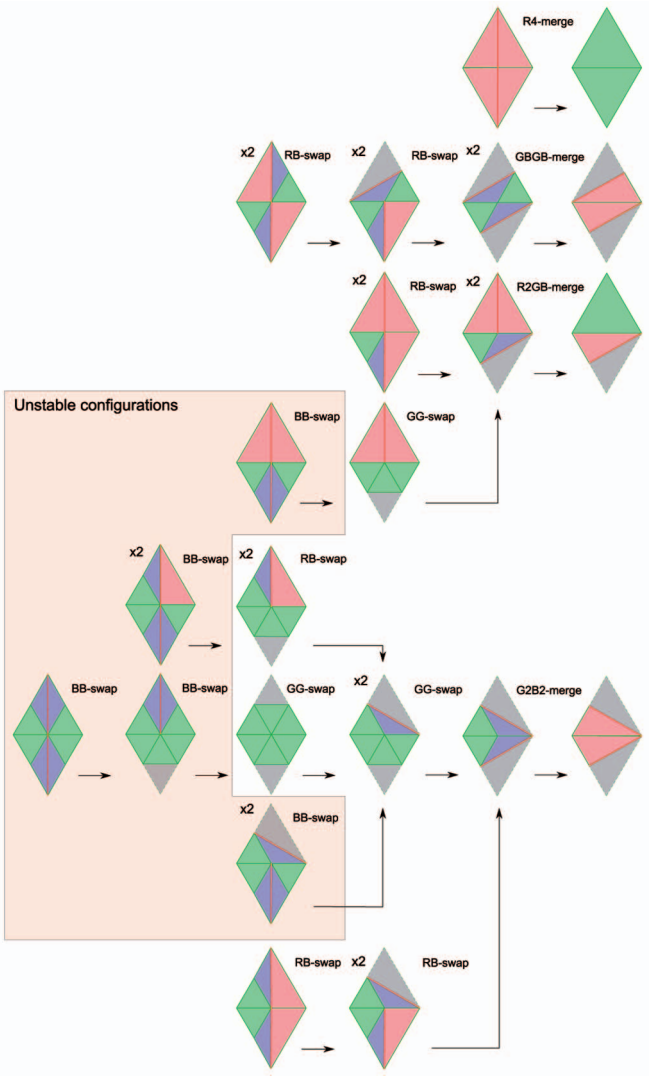


Fig. 11. Sequences of operators to remove a vertex. There exist 28 configurations of triangles incident at a removable vertex, obtained by mirroring from the ones depicted in the figure (except those in the last column, which correspond to the configurations after deleting the vertex). Each configuration labeled with $\times 2$ has a mirror configuration. Triangles in gray correspond to areas of the parent triangles that are not affected by transitions and may be further refined. Except for the unstable configurations, a vertex can be removed without using any refinement operator.

Proof. We prove the second statement first. Let m be the level of Σ. We can obtain Σ_0 from Σ by deleting all vertices of level > 0 level by level, starting at level m. As shown above, if Σ is stable, a vertex can be removed without the need to apply BB-swap, and the resulting mesh will again be stable. Thus, the whole sequence will need just coarsening operators and RB-swap. The first statement follows from the second by considering the inverse operators. □

We now know refinement and coarsening sequences that are monotone in the transition space. Once the star of either a refinable edge or a removable vertex is known, the sequence of operations necessary to perform the corresponding either refinement or coarsening operation, respectively, can be retrieved from a lookup table

and performed on such a star without affecting the rest of the mesh. These sequences will provide the basic ingredients to implement the selective refinement algorithm described next.

5 Selective Refinement of RGB Triangulations

Selective refinement consists of the iterated application of local operators until some user-defined halt condition is verified. Application of local operators is priority-driven. Depending on user needs, priority of an edge to be split may be related to, e.g., its level, its length, the areas of its incident triangles, the distance between its midpoint and the position of the vertex splitting it at next level of subdivision, and so forth. Related criteria set the priority of vertices to be removed.

Following [28], selective refinement is driven by two priority queues: a queue Q_r of underrefined edges to split and a queue Q_c of overrefined vertices to remove. After initialization, the algorithm consists of a loop, which pops elements from the queues and executes the local operations necessary to either split edges or remove vertices. Such operations will involve updating both the current mesh and the queues. Details about the general scheme of the algorithm can be found in [28], while the sequences of refinement and coarsening operators to be performed on an RGB triangulation are detailed in Sections 5.1 and 5.2.

5.1 Coarsening

All removable vertices are added to queue Q_c during selective refinement. However, a vertex v already in Q_c may become nonremovable at a later time because of changes in its star. When a vertex v is popped from Q_c, its star is inspected to check whether or not it is removable. Nonremovable vertices are skipped.

Let now v be a removable vertex at level l that has been popped from Q_c. As we have seen in the previous section, the star of v must be in one of the configurations of Fig. 11. The following operations are performed:

1. Apply the sequence of operators necessary to remove v and update Σ accordingly;
2. For each vertex v' that was adjacent to v, if v' was not removable and it has become removable after removing v, then insert v' into Q_c.

Note that a nonremovable vertex v' at level l becomes removable only if some edge incident at v' at level $> l$ is deleted. This can occur during coarsening only to vertices adjacent to v. Thus, all and only those vertices that are removable are inserted into Q_c.

5.2 Refinement

All green edges are added to queue Q_r during selective refinement. Because of recursive calls, an edge e may be inserted into Q_r more than once. Therefore, it may happen that an edge e popped from Q_c has been already split. When popping an edge, we check whether or not it exists. Nonexisting edges are skipped.

Existing edges in queue Q_r are all green, but they are not necessarily refinable. If an edge e is not refinable, we force recursive split of edges adjacent to e in order to be able to refine it. Splitting a single edge e at level l thus involves the following operations:

1. If e is not refinable, then recursively decompose its incident triangle(s) at level $l-1$ (see details below);
2. Split e at v with the proper split operator and update mesh Σ accordingly;
3. If the mesh has become unstable, then perform a BB-swap;
4. Test each new green edge generated from split and add it to Q_r if it does not fulfill the LOD requirements;
5. Insert v into Q_c.

Recursive decomposition of triangles (step 1) works as follows: Assume we want to split a green edge e having level l and let t be a triangle incident at e. There are only three possible configurations:

- If t is a red triangle at level $l-1$, then its green edge at level $l-1$ is recursively split;
- If t is a blue triangle at level $l-1$, then t must be adjacent to a red triangle t' at level $l-1$ along its red edge; the green edge of t' at level $l-1$ is split recursively (this split will eventually trigger a BB-swap involving t);
- Otherwise, no action is required (t must be at level l and e is refinable).

6 RGB Subdivision

So far, we have been concerned only with topological changes in an RGB triangulation. The RGB subdivision is now derived by studying the geometry of vertices. The basic idea here is to adapt the rules of Loop subdivision to the topology of RGB triangulations, so that the limit surfaces of the two subdivision schemes become coincident.

For odd vertices, we devise a simple mechanism for using the stencil of the Loop subdivision as is. For even vertices, we rather derive a multistep rule for the Loop subdivision that computes the control point of a vertex at any given level on the basis of its insertion position and its limit position. We then factorize computation of the limit position of vertices, which depends on its neighbors, while such neighbors are inserted into the mesh.

Note that in RGB subdivision both refinement and coarsening operations are allowed; therefore, updates to control points must be made for odd vertices during refinement and for even vertices both during refinement and during coarsening.

6.1 A Multistep Rule for the Loop Scheme

Let us consider a vertex v inserted at level l in a Loop subdivision scheme. If $l=0$, then v belongs to the base mesh and its geometry $p^0(v)$ is known; otherwise, its control point $p^l(v)$ is computed on the basis of either (1) or (2), depending on v being an internal or a boundary vertex, respectively (see Fig. 4).

By applying the concept of multistep subdivision rule [24] to the analysis of the Loop scheme developed in [30], the following equations are derived (see the Appendix). The

limit position of an internal vertex v inserted at level $l \geq 0$ on the subdivision surface is given by

$$p^{\infty}(v) = \left(1 - \frac{8\alpha_n}{3+8\alpha_n}\right)p^l(v) + \frac{8\alpha_n}{n(3+8\alpha_n)}\sum_{i=1}^{n} p^l(v_i), \quad (6)$$

where the v_i's and α_n are defined as in (3). For any $k \geq 0$, the control point of v at level $l+k$ is given by

$$p^{l+k}(v) = \gamma_n(k)p^l(v) + (1-\gamma_n(k))p^{\infty}(v), \quad (7)$$

where

$$\gamma_n(k) = \left(\frac{5}{8} - \alpha_n\right)^k.$$

Similarly, if v is a boundary vertex, we have

$$p^{\infty}(v) = \frac{2}{3}p^l(v) + \frac{1}{6}\left(p^l(v_0) + p^l(v_1)\right), \quad (8)$$

and for any $k \geq 0$,

$$p^{l+k}(v) = \left(\frac{1}{4}\right)^k p^l(v) + \left(1 - \left(\frac{1}{4}\right)^k\right)p^{\infty}(v). \quad (9)$$

This means that the base position of a given vertex v introduced at level l plus the positions of its neighbors at level l are necessary and sufficient to compute the position of v at any further level of subdivision.

6.2 Factorized Computation of the Limit Position

In an RGB subdivision, when a vertex v at a level $l > 0$ is inserted into a mesh, some of its neighbors at level l might not belong to such mesh yet, while they will be inserted at a later time. Therefore, it is not always possible to know the limit position of a vertex v right after its insertion into the mesh.

In the data structure encoding an RGB mesh, for each vertex v inserted at level l, we store its control point $p^l(v)$ and we reserve another field to store its limit position $p^{\infty}(v)$. At startup, we fill such fields for all vertices of the base mesh, assigning to $p^0(v)$ the base coordinates and computing $p^{\infty}(v)$ through (6).

For a generic vertex inserted at level $l > 0$, control point $p^l(v)$ is computed and stored when creating v (see Section 6.3), while $p^{\infty}(v)$ is computed incrementally. In $p^{\infty}(v)$, we store at any time a value computed through (6), where we use $p^l(v)$ in place of any missing $p^l(v_i)$. So, initially $p^{\infty}(v)$ will be approximated with $p^l(v)$, and its value will be updated every time the control point at level l for a neighbor v_i becomes available (see Section 6.4). As soon as all six contributions have been obtained, the value of $p^{\infty}(v)$ will be the correct one.

6.3 Control Points for Odd Vertices

Let vertex v be introduced at level $l+1$ of subdivision by splitting an edge e at level l. In order to compute control point $p^{l+1}(v)$, we need to retrieve four vertices in the Loop stencil of v at level l and their control points at level l.

Consider the four possible cases as depicted in Fig. 5. In GG-split, the two splitting triangles form exactly the standard Loop stencil. In the other cases, however, whenever a red triangle is splitting, its vertex opposite to e does not belong to the standard Loop stencil. The correct vertex to be used for the stencil is rather at the apex of the triangular zone depicted in gray in Fig. 5, which may have been refined at arbitrarily many levels, as shown in Fig. 12. Such a vertex in fact belongs to the parent of the red triangle.

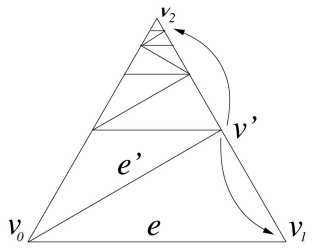


Fig. 12. Upper half of a stencil for an odd vertex. The triangle incident at e at level l might have been split through all successive levels of subdivision. Arrows show references from vertex v' to the endpoints of the edge it splits.

In summary, in order to compute $p^{l+1}(v)$, the two vertices at the endpoints of edge e are always available. For the other two vertices, we do the following: if a triangle incident at e is green, then we use its vertex opposite to e; if the triangle is red, then we use the vertex of its parent triangle opposite to e. We are left with the problem of retrieving such a vertex. It would be possible to retrieve it by navigating the mesh, but we rather prefer to avoid such computational overhead, so we use another method. Referring to Fig. 12, let us consider a red edge e' at level l bisecting a triangle t at level l, and let v_0, v_1, and v_2 be the vertices of t. Without loss of generality, let v_0 be an endpoint of e', and let v' at level $l+1$ be the other endpoint of e'. Note that each of the two halves of t may be refined further at arbitrarily many levels. When e' is generated, we store at v' two references to vertices v_1 and v_2 and we maintain such references until v' has a red incident edge, such that v' is the endpoint with higher insertion level of such edge (there may be at most two such edges incident at v'). In this way, when a red triangle is involved in a stencil, we retrieve the proper vertex to be used for the stencil through the references at its vertex opposite to the splitting edge.

We must take care of keeping our data structure up to date during topological changes of the mesh. This is possible in constant time whenever a red edge is generated. Consider all the possible operators, as depicted in Figs. 5 and 6. New red edges are generated just by operators GG-split, RG-split, GG-swap, and RB-swap. For the split operators, references to vertices v_1 and v_2 are found immediately, since they are the endpoints of the splitting edge. For RB-swap, one of the references is available from the red triangle, while the other is obtained from the vertex that "loses" the red incident edge. For GG-swap, each vertex to be referenced is at the apex of a gray triangular zone, which may have been refined further. Consider one of the two new blue triangles and its adjacent gray zone. Either the gray zone is covered entirely by a green triangle and, therefore, the vertex to be referenced is available from it, or the triangle t' adjacent to the blue triangle inside the gray

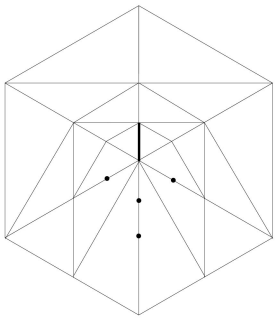


Fig. 13. In order to bisect the edge in bold, vertices marked by bullets must be computed by recursive edge split.

zone is red and the reference is obtained from the proper vertex of the red edge of t'.

Once we know the vertices of the stencil, we must compute their correct control points. If the splitting edge e is at level $l-1$, then, for each vertex v_i in the stencil, we must compute $p^{l-1}(v_i)$. If the insertion level of v_i is $l_i < l-1$, then $p^l(v_i)$ must be computed through (7). To this aim, we need the correct value of $p^\infty(v_i)$, which will not be available unless all neighbors of v_i at level l_i belong to the current mesh. We therefore force the insertion of any missing vertex, by traversing the list of edges incident at v_i and recursively splitting all green edges incident at v_i and having a level lower than l_i. Note that a regular vertex may have green incident edges that differ for at most three levels, thus the number of new vertices to be inserted recursively is usually quite small (see Fig. 13). More in general, an extraordinary vertex of valence n may have green incident edges that differ for at most $\lceil \frac{n}{2} \rceil$ levels.

6.4 Control Points for Even Vertices

An even vertex v inserted at level l will be always represented with a control point at level k, where k is the smallest value between l and the smallest level of edges incident at v. If $k > l$, the value of $p^k(v)$ is computed with (7). As already mentioned, such a value depends on $p^\infty(v)$, which is computed incrementally, so an approximated value will be used, as defined in Section 6.2, until $p^\infty(v)$ is completely specified.

Updates to the limit position $p^\infty(v)$ for a vertex v inserted at level l are done in the following cases:

- When inserting v, we initially approximate all the $p^l(v_i)$'s with $p^l(v)$, so we initially set $p^\infty(v) = p^l(v)$. Then, we collect contributions from adjacent vertices through green edges at level l. If the insertion level of one such vertex v_i is l, then the contribution comes from $p^l(v_i)$; otherwise, we get the contribution only if the correct value of $p^\infty(v_i)$ is available. Such a contribution is added to $p^\infty(v)$ according to the summation of (6), substituting the approximated contribution computed with $p^l(v)$, which is subtracted from $p^\infty(v)$.
- When the contribution of a neighbor v_i of v through a green edge at level l becomes available, then it is computed as in the previous case. This can happen when either the correct limit value of an existing neighbor of v becomes available or a new neighbor of v is generated, and its contribution is readily available.

These operations are also performed in the opposite direction, since each v_i may need the contribution of v to compute its own limit position.

When a vertex v is removed, its neighbors are also checked and the contribution of v is subtracted from its neighbors that received it. For a given neighbor v_i, if the minimum level of incident edges has become lower, then its current control point is also updated accordingly.

7 Data Structure

An RGB triangulation can be maintained in a standard topological data structure for triangle meshes. One possibility is using three dynamic arrays, for vertices, edges, and triangles, respectively, with a garbage collection mechanism to manage reuse of locations freed because of coarsening operators. The following simplified version of the incidence graph [31] can be adopted: for each triangle, links to its three edges are maintained; for each edge, links to its two vertices and its two adjacent triangles are maintained; for each vertex, just a link to one of its incident edges is maintained. This is sufficient to compute topological relations in optimal time.

If the mesh contains n vertices, we can roughly estimate its number of triangles and edges to be about $2n$ and $3n$, respectively. By assuming unit cost to represent a pointer or a number, the total cost for topological information in this base structure is about $19n$, and an additional $3n$ is necessary to maintain the coordinates of the vertices.

This data structure is extended as follows: For each vertex, we maintain its level of insertion and a counter to keep track of the number of neighbors that have given their contribution for computing the limit position (one byte is sufficient for both); two triples of coordinates rather than just one (position at time of insertion and limit position); two references to vertices of its parent triangle, only in case the vertex is incident at a red edge as described in Section 6.3. Since the number of red edges in a mesh is usually small, it may be more efficient to avoid storing such references in the main data structure and use a hash table instead, indexed on vertices.

For each edge and each triangle, we maintain its color and its level. Edges come in just two colors. It is convenient to encode two different types of red triangles and two different types of blue triangles, depending on their orientation: a red triangle will be said to be either Red_{RGG} or Red_{GGR} depending on the colors of its edges, traversed in counterclockwise order starting at the vertex with the highest insertion level; a blue triangle will be said to be either Blue_{RGG} or Blue_{GGR} depending on the colors of its edges, traversed in counterclockwise order starting at the vertex with the lowest insertion level. We thus use five different color codes for triangles: two for red triangles Red_{RGG} and Red_{GGR}, two for blue triangles $Blue_{RGG}$ and $Blue_{GGR}$, and one for green triangles. Since three [one] bits are sufficient for the color of triangles [edges] and levels in subdivision are usually not many, one byte is sufficient to store both color and level.

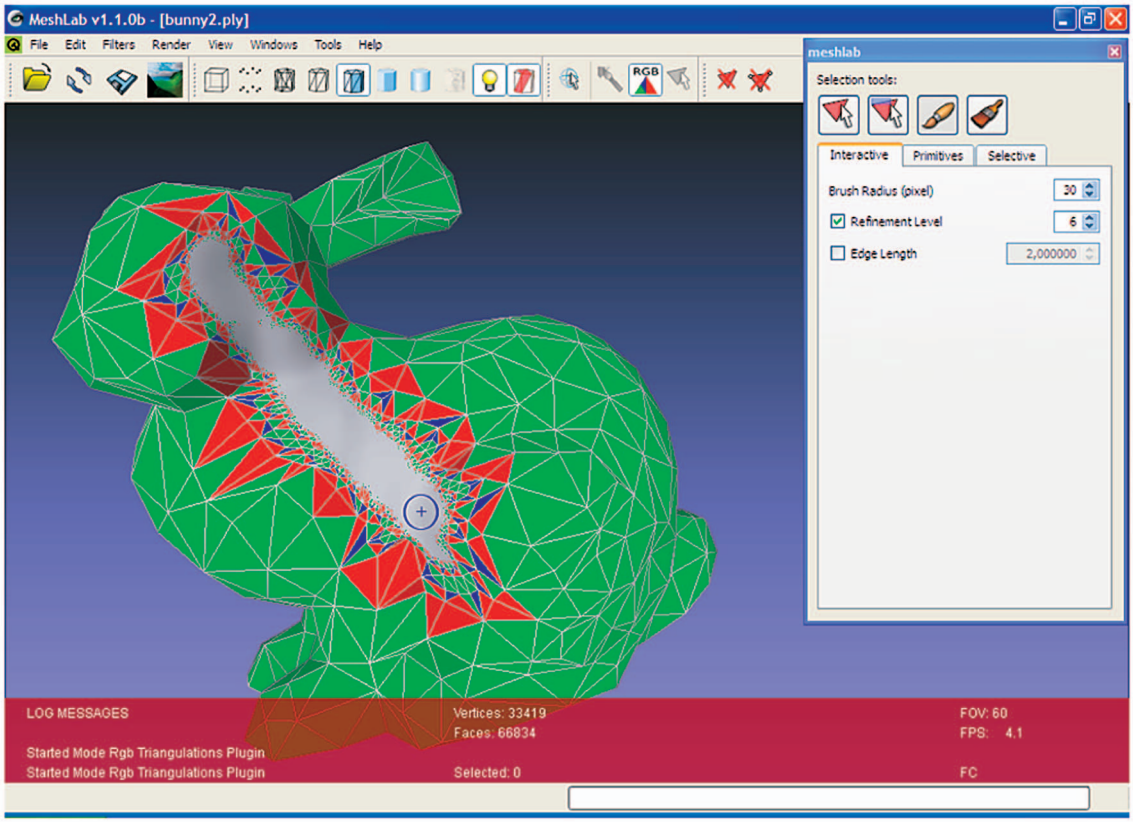


Fig. 14. The front-end user interface of *MeshLab* with the control panel of RGB subdivision on the right. The blue circle on the object visualized is the print of the brush, which is sweeping an LOD refinement from level 0 to level 6 of subdivision. A base mesh consisting of about 1,000 faces has been selectively refined in real time to a mesh with more than 60,000 faces by a single sweep of the brush.

Summing up, by assuming one unit of cost to be equal to four bytes, we have an additional cost between $4.5n$ and $6.5n$, depending on whether or not a hash table is used for the additional references of vertices. This corresponds to an overhead between 20 percent and 30 percent with respect to the base data structure.

8 Modeling with RGB Subdivision

We have implemented an interactive tool that supports fully dynamic selective refinement for LOD editing through graphic widgets. Our tool is a prototype implemented as a plug-in for *MeshLab*, an open source tool for processing, editing, and visualizing 3D triangular meshes [32]. Implementation is based on the *VGC Library*, an open source API for geometry processing [33]. The plug-in will be included in the next release of *MeshLab*. A beta version of the software can be currently downloaded from our website http://ggg.disi.unige.it/rgbtri/.

Fig. 14 shows a snapshot of the front-end user interface of *MeshLab* with the control panel of the RGB plug-in on the right. Our plug-in allows a user to edit LOD in different ways:

- A portion of the mesh can be selected and either refined or coarsened by setting the desired LOD inside and outside selection. LOD may be set in terms of levels of subdivision and maximal length of edges;
- A brush tool can be used to adjust LOD locally: the region swept by the brush is either refined or coarsened by changing the LOD according to the parameters set for the brush;
- Edge split and vertex removal primitives can be applied individually for fine editing.

We have tested our tool on a number of models representing various objects. Most objects were described with base meshes in the order of 10^2-10^4 faces, which have been selectively refined up to sizes of order 10^6. We show results on four models: a very simple model representing a 3D star (base mesh 24 triangles); a character for videogames (base meshes 1,008 triangles); the hand of another character (base mesh 618 triangles); and a mesh representing a hippo, which contains triangles of very different size (base mesh 46,202 triangles).

For the hockey player and the hand, we show how selective refinement can be applied to parts of the skin near joints, e.g., to better follow bending during animation (see Figs. 15 and 16). The star is a very symmetric model, which is useful to show how RGB subdivision acts regularly, in spite of being based on local operators that are applied dynamically (see Fig. 17). We have also removed one spike of the star to show how RGB subdivision acts on boundaries. On the hippo model, we let the selective refinement algorithm refine the whole mesh until a given budget of faces was reached, by giving higher priority to longer edges. In this case, RGB subdivision acts

Fig. 15. Only the face and the right side of the hockey player have been refined: level of subdivision varies from zero to two and it is higher at joints and on the face. This mesh contains 2,826 faces.

as a remeshing method to make the mesh more uniform (see Fig. 18).

In all examples, it is clearly visible that RGB subdivision is truly selective: LOD can increase/decrease as fast as two subdivision levels per ring of triangles around a vertex/edge/triangle. Note that the number of blue triangles appearing in the meshes is quite small. Additional RB-swap operators could be forced to improve the shape of such triangles further (e.g., based on max-min angle criterion) without affecting the RGB structure.

We have evaluated time performance by running the program on a PC with a Pentium IV 2.8-GHz processor and 3 Gbytes of memory. Processing times for editing with the brush tool are always compatible with interactive use. Actually, when the mesh contains many triangles, most time is spent by *MeshLab* in picking and rendering, while the time for selective refinement is almost negligible. In general, time performance is very good with all widgets, even on large meshes, as long as the effects of editing are local. If the tool is used for uniform refinement of the whole mesh through several levels of subdivision, then it becomes slower than standard Loop subdivision. In this modality,

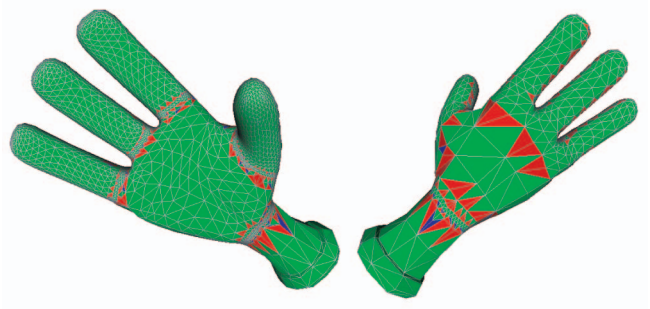


Fig. 16. The two sides of the hand have been refined differently: the palm is at level one, fingers are at level two, and joints of fingers and of the wrist are at level three; the rest of the wrist as well as the back of the hand are at level zero. This mesh contains 4,821 faces.

the performance resulting from our profiles is of about 15,000 refinement operations per second. Therefore, it can take several seconds to get a mesh of size 10^5 through five or six levels of subdivision. This is only in part due to the factorization of subdivision through local operators. In our prototype, we had to comply with the structure of *MeshLab*, so we encoded RGB triangulations in a data structure different from that described in the previous section and less efficient than it. We believe that a more careful implementation could greatly improve performance, hence supporting more real-time tasks, such as view-dependent visualization, even on very large meshes.

9 Conclusions

The RGB subdivision scheme has several advantages over both classical and adaptive subdivision schemes, as well as over CLOD models: it supports fully dynamic selective refinement while remaining compatible with the Loop subdivision scheme; it is better adaptive than previously known schemes based on the one-to-four triangle split pattern; it does not require hierarchical data structures; selective refinement can be implemented efficiently by plugging faces inside the mesh, according to rules encoded in lookup tables, thus avoiding cumbersome procedural updates.

We are currently developing an analogous scheme for the modified butterfly subdivision [34]. Based on a similar approach, we are also developing a hybrid tri-quad adaptive scheme for the selective refinement of quad meshes. The basic ideas of this latter work are sketched in [35].

We believe that this approach to adaptive subdivision may give valid substitutes or complements to standard subdivision for solid modelers and simulation systems. Combined with reverse subdivision techniques, it may also offer a valid alternative to CLOD models for free-form objects in computer graphics.

Our prototype integrated in *MeshLab* can be already used for interactive editing of LOD. However, a more careful implementation of our data structures should provide a much more efficient engine, suitable for tasks such as real-time view-dependent rendering, or integration in a solid modeler.

Concerning rendering, our scheme is already progressive and we did not find many popup effects during selective refinement. However, morphing techniques [22] could be

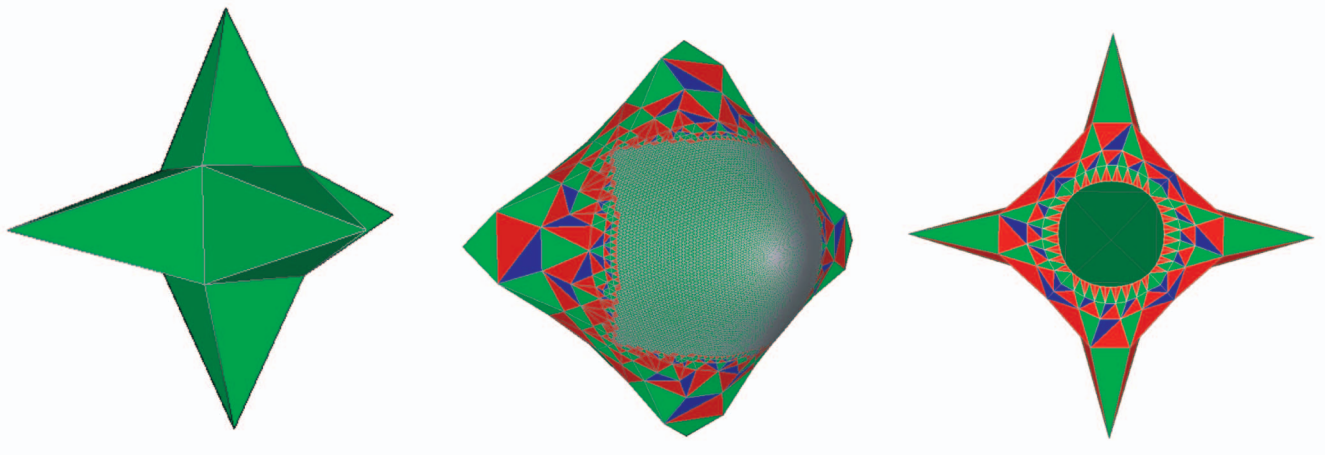


Fig. 17. The base mesh of the star has six spikes; one spike has been selected and refined to six levels of subdivision; one spike has been removed and the square boundary has been refined to four levels.

incorporated easily to make frame-to-frame transitions smooth under any possible changes.

A crucial feature to support modeling is surface editing. Editing at the base level should be easy on the RGB subdivision, by exploiting the mechanisms that we already implemented to propagate contributions for computing the limit position of vertices. Editing at a finer level involves backward propagation to lower levels and may require techniques of reverse subdivision. Either the mechanisms proposed in [11] or those proposed in [20] for hierarchical splines could be probably extended to RGB subdivisions. In the future, we plan to develop these features and to integrate our scheme in Blender [3], an open source solid modeler.

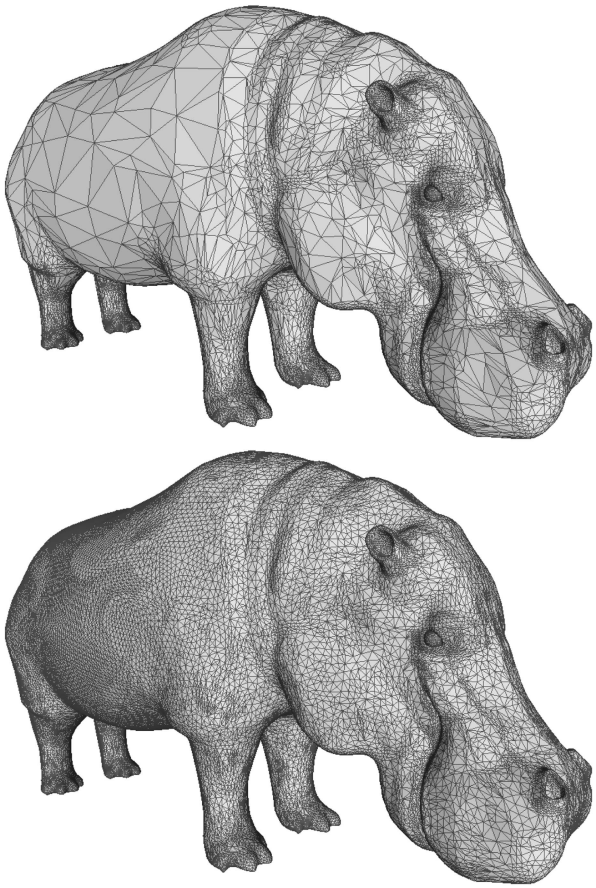


Fig. 18. The original mesh contains triangles of very different size. Selective refinement is run on the whole object until a mesh of 100,000 triangles is obtained, giving higher priority to the refinement of longer edges. The triangles in the resulting mesh are much more uniform.

APPENDIX

MULTIPASS FORMULAS FOR THE LOOP SUBDIVISION

Following Stam [30], the portion of subdivision matrix involving an internal vertex v and its n neighbors $v_0, \ldots, v_{n-1}$ has the following structure:

$$S = \begin{pmatrix} a_n & b_n & b_n & b_n & b_n & \ldots & b_n & b_n & b_n \\ 3/8 & 3/8 & 1/8 & 0 & 0 & \ldots & 0 & 0 & 1/8 \\ 3/8 & 1/8 & 3/8 & 1/8 & 0 & \ldots & 0 & 0 & 0 \\ & & \vdots & & & \ddots & & \vdots & \\ 3/8 & 1/8 & 0 & 0 & 0 & \ldots & 0 & 1/8 & 3/8 \end{pmatrix},$$

where $a_n = 1 - \alpha_n$, $b_n = \alpha_n / n$, and α_n is defined by (4).

If $\mathbf{v} = (v, v_0, \ldots, v_{n-1})^T$, then the product $S\mathbf{v}$ gives the control point of v and all its neighbors at the next level of subdivision. Thus, the control point of v after k levels of subdivision can be computed by multiplying the first row of matrix S^k by $\mathbf{v}$. We can obtain $S^k = U\Lambda^k U^{-1}$ from the decomposition $S = U\Lambda U^{-1}$, where U is the matrix of eigenvectors and Λ is the diagonal matrix of eigenvalues. Actually, we are interested just in the first row of such a matrix. Moreover, by symmetry, we know that all coefficients s^k_{1j} for $j = 2, n+1$ in the first row of S^k must be equal. Thus, it is sufficient to compute just coefficients s^k_{11} and s^k_{12}.

From Stam [30], we have that the first row of matrix U is

$$U_{1,*} = \left(1, -\frac{8}{3}\alpha_n, 0, \ldots, 0\right)$$

and

$$\Lambda = \mathrm{diag}(\lambda_0, \lambda_1, \ldots, \lambda_n),$$

where $\lambda_0 = 1$ and $\lambda_1 = 5/8 - \alpha_n$. For our purposes, we can disregard all the other eigenvalues. Thus, the first row of matrix $U\Lambda^k$ will be

$$(U\Lambda^k)_{1,*} = \left(1, -\frac{3}{8}\alpha_n\left(\frac{5}{8} - \alpha_n\right)^k, 0, \ldots, 0\right).$$

Let u_{11} and u_{21} be the first two coefficients of the first column of matrix U^{-1}, thus we have

$$s_{11} = 1 - \alpha_n = (U\Lambda)_{1,*}U^{-1}_{*,1} = u_{11} - \frac{3}{8}\alpha_n\left(\frac{5}{8} - \alpha_n\right)u_{21}, \quad (10)$$

and since $UU^{-1} = I$, we also have

$$u_{11} - \frac{8}{3}\alpha_n u_{21} = 1. \quad (11)$$

Equations (10) and (11) form a linear system from which we obtain

$$u_{11} = \frac{1}{1 + \frac{8}{3}\alpha_n} \quad u_{21} = \frac{-1}{1 + \frac{8}{3}\alpha_n}.$$

So, now we can compute the first coefficient of S^k as follows:

$$s^k_{11} = (U\Lambda^k)_{1,*}U^{-1}_{*,1} = \cdots = \frac{3 + 8\alpha_n(5/8 - \alpha_n)^k}{3 + 8\alpha_n}.$$

Proceeding in the same way, we obtain the second coefficient of the first row of S^k:

$$s^k_{12} = \frac{8\alpha_n\left(1 - (5/8 - \alpha_n)^k\right)}{n(3 + 8\alpha_n)}.$$

Now, by defining

$$\beta_n(k) = \frac{8\alpha_n\left(1 - (5/8 - \alpha_n)^k\right)}{3 + 8\alpha_n},$$

we have

$$s^k_{11} = 1 - \beta_n(k), \quad s^k_{12} = \beta_n(k)/n.$$

Hence, the control point of vertex v after k levels of subdivisions (following the level at which v was inserted as an odd vertex) is

$$\begin{aligned} p^k(v) &= S^k_{1,*}\mathbf{v} \\ &= s^k_{11}v + s^k_{12}\sum_{i=0}^{n-1} v_i \\ &= (1 - \beta_n(k))v + \frac{\beta_n(k)}{n}\sum_{i=0}^{n-1} v_i. \end{aligned} \quad (12)$$

The limit position of vertex v is thus

$$p^\infty(v) = \lim_{k\to\infty} p^k(v) = \left(1 - \beta^\infty_n\right)v + \frac{\beta^\infty_n}{n}\sum_{i=0}^{n-1} v_i,$$

where

$$\beta^\infty_n = \lim_{k\to\infty} \beta_n(k) = \frac{8\alpha_n}{3 + 8\alpha_n}.$$

By substituting the above value in (12), we obtain

$$p^k(v) = \gamma_n(k)v + (1 - \gamma_n(k))p^\infty(v),$$

where

$$\gamma_n(k) = \left(\frac{5}{8} - \alpha_n\right)^k.$$

For a boundary vertex, the corresponding portion of subdivision matrix is

$$S = \frac{1}{8}\begin{pmatrix} 6 & 1 & 1 \\ 4 & 4 & 0 \\ 4 & 0 & 4 \end{pmatrix}.$$

In this case, we can compute the decomposition explicitly as

$$U = \begin{pmatrix} 1 & 1 & 0 \\ 1 & -2 & 1 \\ 1 & -2 & -1 \end{pmatrix}, \quad U^{-1} = \begin{pmatrix} \frac{2}{3} & \frac{1}{6} & 0 \\ \frac{1}{3} & -\frac{1}{6} & -\frac{1}{6} \\ 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix},$$

and $\Lambda = \mathrm{diag}(1, \frac{1}{4}, \frac{1}{2})$. Therefore, we have

$$s^k_{11} = \frac{1}{3}\left(2 + \left(\frac{1}{4}\right)^k\right) \quad s^k_{12} = s^k_{13} = \frac{1}{6}\left(1 - \left(\frac{1}{4}\right)^k\right).$$

Proceeding as in the previous case, we obtain

$$p^\infty(v) = \frac{2}{3}v + \frac{1}{6}(v_0 + v_1)$$

and

$$p^k(v) = \left(\frac{1}{4}\right)^k v + \left(1 - \left(\frac{1}{4}\right)^k\right)p^\infty(v).$$

ACKNOWLEDGMENTS

This work has been partially supported by Project FIRB-MIUR SHApe modeLing and reasOning: new Methods and tools (SHALOM) funded by the Italian Ministry of Education, University and Research under Contract RBIN04HWR8.

REFERENCES

[1] T. DeRose, M. Kass, and T. Truong, "Subdivision Surfaces in Character Animation," *Proc. ACM SIGGRAPH '98,* pp. 85-94, 1998.

[2] P.-O. Persson, M. Aftosmis, and R. Haimes, "On the Use of Loop Subdivision Surfaces for Surrogate Geometry," *Proc. 15th Int'l Meshing Roundtable (IMR '06),* pp. 375-392, Sept. 2006.

[3] *Blender,* http://www.blender.org/, 2008.
[4] *Autodesk Maya,* http://usa.autodesk.com/, 2008.
[5] *Modo 301,* http://www.luxology.com, 2008.
[6] *Silo 2,* http://www.nevercenter.com/, 2008.
[7] A. Lee, H. Moreton, and H. Hoppe, "Displaced Subdivision Surfaces," *Proc. ACM SIGGRAPH '00,* pp. 85-94, 2000.
[8] M. Sabin, "Recent Progress in Subdivision: A Survey," *Advances in Multiresolution for Geometric Modelling,* N. Dogdson, M. Floater, and M. Sabin, eds., pp. 203-230, Springer-Verlag, 2004.
[9] F. Samavati and R. Bartels, "Multiresolution Curve and Surface Representation by Reversing Subdivision Rules," *Computer Graphics Forum,* vol. 18, no. 2, pp. 97-120, 1999.
[10] F. Samavati, N. Mahdavi-Amiri, and R. Bartels, "Multiresolution Surfaces Having Arbitrary Topologies by a Reverse Doo Subdivision Method," *Computer Graphics Forum,* vol. 21, no. 2, pp. 121-136, 2002.
[11] D. Zorin, P. Schröder, and W. Sweldens, "Interactive Multiresolution Mesh Editing," *Proc. ACM SIGGRAPH '97,* pp. 259-268, http://muggy.gg.caltech.edu/~dzorin/multires/meshed/index.html, 1997.
[12] *Subdivision for Modeling and Animation (SIGGRAPH 2000 Tutorial N.23—Course Notes),* D. Zorin and P. Schröder, eds., ACM Press, 2000.
[13] D. Lübke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Hübner, *Level of Detail for 3D Graphics.* Morgan Kaufmann, 2002.
[14] C. Loop, "Smooth Subdivision Surfaces Based on Triangles," master thesis, Dept. of Math., Univ. of Utah, 1987.
[15] N. Dyn, D. Levin, and J. Gregory, "A Butterfly Subdivision Scheme for Surface Interpolation with Tension Control," *ACM Trans. Graphics,* vol. 9, no. 2, pp. 160-169, Apr. 1990.
[16] J. Warren and H. Weimer, *Subdivision Methods for Geometric Design.* Morgan Kaufmann, 2002.
[17] R. Bank, A. Sherman, and A. Weiser, "Refinement Algorithms and Data Structures for Regular Local Mesh Refinement," *Scientific Computing,* R. Stepleman, ed., pp. 3-17, IMACS/North Holland, 1983.
[18] H. Pakdel and F. Samavati, "Incremental Subdivision for Triangle Meshes," *Int'l J. Computational Science and Eng.,* vol. 3, no. 1, pp. 80-92, 2007.
[19] D. Forsey and R. Bartels, "Hierarchical B-Spline Refinement," *Computer Graphics, Proc. ACM SIGGRAPH '88,* vol. 22, no. 4, pp. 205-212, Aug. 1988.
[20] A. Yvart, S. Hahmann, and G.-P. Bonneau, "Hierarchical Triangular Splines," *ACM Trans. Graphics,* vol. 24, no. 4, pp. 1374-1391, 2005.
[21] S. Seeger, K. Hormann, G. Häusler, and G. Greiner, "A Sub-Atomic Subdivision Approach," *Proc. Vision, Modeling and Visualization (VMV '01),* B. Girod, H. Niemann, and H.-P. Seidel, eds., pp. 77-85, 2001.
[22] H. Hoppe, "Progressive Meshes," *Proc. ACM SIGGRAPH '96,* pp. 99-108, Aug. 1996.
[23] L. Velho, "Stellar Subdivision Grammars," *Proc. First Eurographics Symp. Geometry Processing (SGP),* 2003.
[24] L. Kobbelt, "$\sqrt{3}$ Subdivision," *Proc. ACM SIGGRAPH '00,* pp. 103-112, 2000.
[25] L. Velho and D. Zorin, "4-8 Subdivision," *Computer-Aided Geometric Design,* vol. 18, pp. 397-427, 2001.
[26] E. Puppo, "Variable Resolution Triangulations," *Computational Geometry,* vol. 11, nos. 3-4, pp. 219-238, 1998.
[27] J. Kim and S. Lee, "Transitive Mesh Space of a Progressive Mesh," *IEEE Trans. Visualization and Computer Graphics,* vol. 9, no. 4, pp. 463-480, 2003.
[28] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein, "ROAMing Terrain: Real-Time Optimally Adapting Meshes," *Proc. IEEE Visualization Conf. (VIS '97),* pp. 81-88, Oct. 1997.
[29] L. Velho and J. Gomes, "Variable Resolution 4-K Meshes: Concepts and Applications," *Computer Graphics Forum,* vol. 19, no. 4, pp. 195-214, 2000.
[30] J. Stam, "Evaluation of Loop Subdivision Surfaces," *ACM SIGGRAPH '98 CDROM Proc.,* 1998.
[31] H. Edelsbrunner, *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.
[32] *Meshlab,* http://meshlab.sourceforge.net, 2008.
[33] *VCG Library,* http://vcg.sourceforge.net, 2008.
[34] D. Zorin, P. Schröder, and W. Sweldens, "Interpolating Subdivision for Meshes with Arbitrary Topology," *Proc. ACM SIGGRAPH '96,* pp. 189-192, 1996.
[35] E. Puppo, "Dynamic Adaptive Subdivision Meshes," *Proc. Israel-Italy Bi-National Conf. Shape Modeling and Reasoning for Industrial and Biomedical Application,* pp. 60-64, May 2007.

Enrico Puppo received the Laurea degree in mathematics from the University of Genova, Genoa, Italy, in 1986. Since November 1998, he has been a professor of computer science in the Department of Computer and Information Sciences (DISI), University of Genova, where he coleads the Geometry and Graphics Group. From 1986 to 1998, he has been a research scientist in the Institute for Applied Mathematics, National Research Council of Italy. He has written more than 100 technical publications on the subjects of algorithms and data structures, geometric modeling, computer graphics, and image processing. His current research interests are in geometric algorithms and data structures, with applications to CAGD, GIS, scientific visualization, and shape retrieval. He is a member of the IEEE, the ACM, the Eurographics Association, and the International Association for Pattern Recognition (IAPR).

Daniele Panozzo received the undergraduate degree in computer science from the University of Genova in 2007. He is currently a student in the graduate program in computer science at the University of Genova. Between March and June 2008, he was a visiting student in the Department of Computer Science, University of Maryland, where he worked on his master thesis under the supervision of Prof. H. Samet. This work was the subject of his final project in the undergraduate program. His current research interests are in geometric modeling and computer graphics.