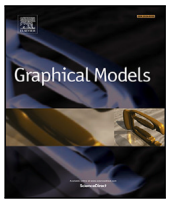
Graphical Models 144 (2026) 101322

Contents lists available at ScienceDirect

Graphical Models

journal homepage: www.elsevier.com/locate/gmod



Simplification of locally refined gradient meshes

E. Kato [b], T. Ophelders [b,c], A. Telea [b], J. Kosinka [a,*]

[a] University of Groningen, Groningen, Netherlands
[b] Utrecht University, Netherlands
[c] TU Eindhoven, Netherlands

ARTICLE INFO

Dataset link: https://github.com/evakato/grad mesh-simplification

Keywords:
Vector graphics
Gradient meshes

ABSTRACT

Gradient meshes are powerful vector graphic primitives known for producing smooth and detailed color transitions. However, their fixed rectangular topology complicates editing, as adding detail in one region introduces control points across the entire mesh. To improve editability and better support artist workflows, we propose a simplification method for gradient meshes based on local refinement. Our method transforms a traditional, globally-refined mesh into a locally-refined one by iteratively merging adjacent faces and eliminating redundant data while preserving visual quality. We achieve this by rasterizing the mesh and applying established visual quality metrics to ensure consistency with the original. Additionally, we offer artists control over the simplification process by introducing an error threshold, allowing them to balance the level of simplification with visual fidelity. Finally, we conduct a thorough comparison of various mesh simplification strategies to analyze the trade-offs between simplification quality and speed so as to inform users to the optimal one that they can use to obtain the desired trade-off.

1. Introduction

Vector graphics [1] represent images using geometric primitives, and provide a compact and resolution-independent alternative to raster graphics. A gradient mesh is an advanced primitive in vector graphics that allows for intricate color variations across quadrilateral meshes, and supports highly detailed and photorealistic illustrations. Introduced by Adobe Illustrator [2] as the Mesh object, later adopted by CorelDRAW [3] and Inkscape [4], traditional gradient meshes use a fixed rectangular topology with colors assigned at grid vertices and smooth color transitions filling the grid cells. Yet, when artists need more detail in specific areas, traditional refinement techniques introduce new patches *globally*, complicating the design and slowing down workflows [5]. This problem arises not only for gradient meshes, but also for tensor-product surfaces at large [6].

To address these issues, local refinement and meshes with arbitrary topologies have emerged. While such innovations have enhanced the expressiveness and versatility of traditional gradient meshes, they are primarily used for mesh generation from scratch or for image vectorization [7,8]. Yet, artists may often work with pre-existing, traditional globally-refined, gradient meshes. When editing such meshes, there is no simple way to transition to more advanced representations. We address this challenge by an automatic method that reduces the number of patches in a gradient mesh using local coarsening, without sacrificing visual quality. Our contributions are as follows:

- A robust framework for iteratively merging patches within a mesh;
- A merge error function based on visual quality metrics to guide the simplification process;
- Several algorithms that automate the simplification, each offering a different trade-off between computational complexity and simplification quality.

A strong advantage of our method is that it is fully compatible with the SVG specification as it relies solely on bicubic patches.

The remainder of this paper is structured as follows. Section 2 provides the mathematical foundation of the gradient mesh primitive, including equations for splitting and merging for local refinement (full derivations in Appendix A.1). Section 3 reviews and compares related work on gradient meshes. Section 4 details our framework for iteratively merging patches. Section 5 outlines our merge error function. Section 6 explores several edge selection algorithms. Section 7 evaluates our approach by many examples (full results in Appendix D). Section 8 summarizes our findings and proposes directions for future work. Appendix B details our tool's user interface. Appendix C explains our selection of input meshes.

* Corresponding author.
E-mail address: j.kosinka@rug.nl (J. Kosinka).

https://doi.org/10.1016/j.gmod.2026.101322
Received 4 July 2025; Received in revised form 20 January 2026; Accepted 27 February 2026
Available online 13 March 2026

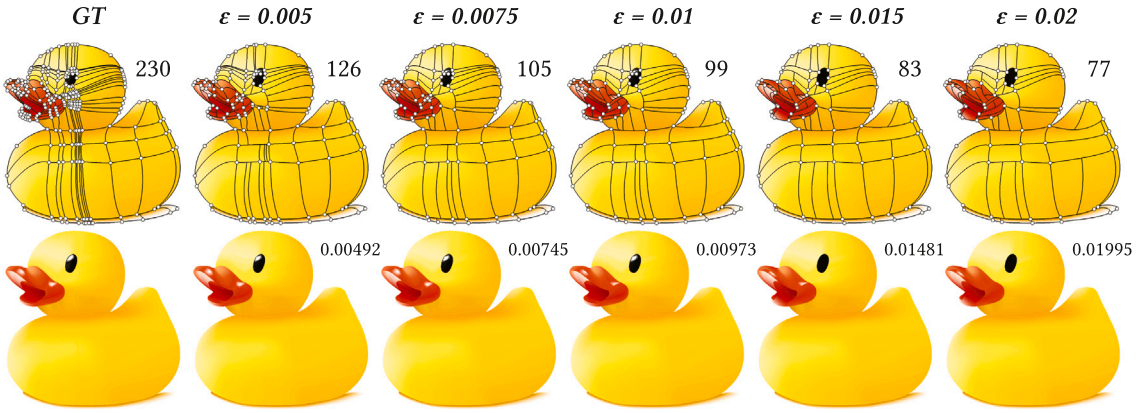


Fig. 1. Duck gradient mesh, 230 patches, simplified using our greedy-quad error method, ꟻLIP error metric, 700 × 700 raster resolution, varying error threshold ϵ values (see also Fig. 15). Full meshes (top) show remaining patches in the simplified mesh. Rendered patches (bottom) are labeled with their error value. $\epsilon = 0.005$ and $\epsilon = 0.0075$ yield simplifications with no visible artifacts; $\epsilon = 0.01$ has a subtle difference in the gradient of the beak; $\epsilon = 0.015$, $\epsilon = 0.02$ lose visible detail in the eye and beak.

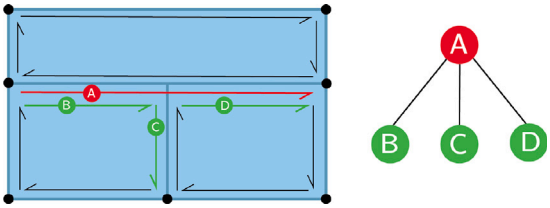


Fig. 2. A T-junction represented in a doubly-connected edge list (left) and the corresponding half-edge tree structure (right). Node A is the parent shown in red, with bar half-edges (Nodes B and D) and a stem half-edge (Node C) as its children in green.

2. Background

2.1. Gradient meshes

A traditional gradient mesh consists of a set of bicubic patches arranged in a fixed rectangular topology. Each patch defines geometry and color information at its corners, which are smoothly over the patch. We use for this the Hermite representation $H : [0,1]^2 \to \mathbb{R}^5$ where

$$H(u,v) = \begin{pmatrix}1 & u & u^2 & u^3\end{pmatrix} \mathbf{B}\mathbf{C}\mathbf{B}^\top \begin{pmatrix}1 & v & v^2 & v^3\end{pmatrix}^\top. \tag{1}$$

The basis matrix $\mathbf{B}$ and control matrix $\mathbf{C}$ are defined as

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -3 & -2 & -1 & 3 \\ 2 & 1 & 1 & -2 \end{pmatrix}, \tag{2}$$

$$\mathbf{C} = \begin{pmatrix} \mathbf{m}^0 & \mathbf{m}^0_v & \mathbf{m}^2_v & \mathbf{m}^2 \\ \mathbf{m}^0_u & \mathbf{m}^0_{uv} & \mathbf{m}^2_{uv} & \mathbf{m}^2_u \\ \mathbf{m}^1_u & \mathbf{m}^1_{uv} & \mathbf{m}^3_{uv} & \mathbf{m}^3_u \\ \mathbf{m}^1 & \mathbf{m}^1_v & \mathbf{m}^3_v & \mathbf{m}^3 \end{pmatrix}. \tag{3}$$

$\mathbf{C}$ specifies the geometry and color data of the patch. Its elements $\mathbf{m}$ are vectors in $\mathbb{R}^5$ – two coordinates x and y and three color components r, g, and b. The tangents with respect to the parameter directions u and v are denoted by $\mathbf{m}_u$ and $\mathbf{m}_v$ respectively. These tangents, also called *handles*, help define the patch's boundary curves; their color is set to zero. Elements $\mathbf{m}_{uv}$ are mixed partial derivatives at the patch corners, also called *twist vectors*.

Bicubic patches make up the gradient mesh. Positions, colors, and tangent handles of corner points are shared across adjacent edges to guarantee global C^1 continuity.

2.2. Local refinement

Gradient meshes are locally refined [5,9,10] by *splitting* a patch along $u = t$ or $v = t$; complete formulations are provided in Appendix A.1. Splitting a patch twice in alternating directions creates a junction in the mesh called a *T-junction*; see Fig. 2 (left). In contrast with *X-junctions*, where four patches meet symmetrically at a shared interior point, *T-junctions* involve only three incident faces at a junction point due to asymmetric, directional splitting. Following [5], we do not make T-junctions editable by the user to maintain smoothness. A T-junction consists of a parent curve, created by the first split, linked to at least three child curves. Unlike all other curves, parent curves span more than one face in the mesh. *Bar* curves are children that are parallel to the parent curve, while *stem* curves are orthogonal to it. The tail control points of all child curves are derived from the parent curve. Tangent handles for bar curves are similarly inherited, whereas those for the stem must be explicitly stored [5].

We use a doubly-connected edge list (DCEL) data structure to model a gradient mesh with local refinement. The DCEL consists of four types of records: points, handles, faces, and half-edges. To accommodate T-junctions, the DCEL is augmented to include a tree structure within the half-edges, acting as a parent or a child. For child nodes, an interval is stored specifying the range (from 0 to 1) of the parent it spans.

2.3. Merging patches

Van der Vis [11] recently proposed an approximate inverse of the splitting operation – combining two adjacent patches into a single patch – a process we call *merging*, a new approach developed specifically for local mesh coarsening.

To merge two patches, Van der Vis first finds the parameter r at which the patch was originally split. The precise calculation assumes that the patches are C^∞, and the merged patch is approximated otherwise. We use linear approximations for r derived from C^1 continuity

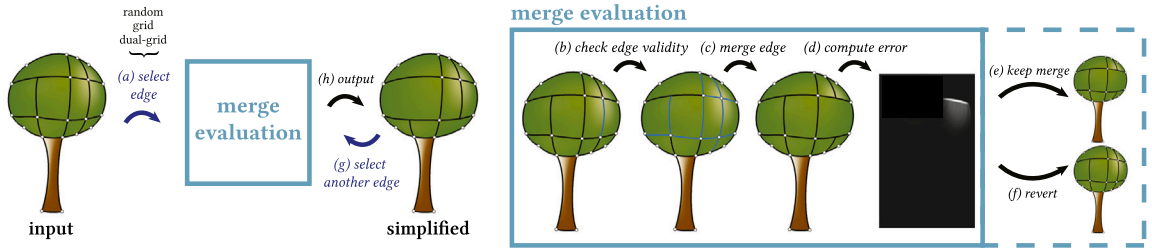


Fig. 3. Our gradient mesh simplification pipeline: (a) select an edge (Section 6), (b) check for merge edge validity (Section 4.2), (c) merge the edge (Sections 2.3 and 4.3), and (d) evaluate the merge quality (Section 5). If the simplified mesh is of sufficient quality, we (e) keep the merge; otherwise we (f) return to a previous state. We repeat the process (g, h) until all edges have been checked.

conditions (see Appendix A.2). Given two adjacent Hermite control matrices $\mathbf{L} = (\mathbf{m})_{ij}$ and $\mathbf{R} = (\mathbf{n})_{ij}$ (left and right), we estimate r as

$$r = \frac{1}{4}\sum_{i=1}^{4} r_i, \text{ where } \begin{cases} r_1 = \dfrac{\left|\mathbf{m}_v^2\right|}{\left|\mathbf{m}_v^2+\mathbf{n}_v^0\right|}, \\ r_2 = \dfrac{\left|\mathbf{m}_v^2+\frac{1}{3}\mathbf{m}_{uv}^2\right|}{\left|\mathbf{n}_v^0+\mathbf{m}_v^2+\frac{1}{3}\left(\mathbf{n}_{uv}^0+\mathbf{m}_{uv}^2\right)\right|}, \\ r_3 = \dfrac{\left|\mathbf{m}_v^3-\frac{1}{3}\mathbf{m}_{uv}^3\right|}{\left|\mathbf{n}_v^1+\mathbf{m}_v^3-\frac{1}{3}\left(\mathbf{n}_{uv}^1+\mathbf{m}_{uv}^3\right)\right|}, \\ r_4 = \dfrac{\left|\mathbf{m}_v^3\right|}{\left|\mathbf{m}_v^3+\mathbf{n}_v^1\right|}. \end{cases} \tag{4}$$

Merging in the v direction, we approximate two adjacent patches as a single Hermite patch $\mathbf{C}_v$ as

$$\mathbf{C}_v = \begin{pmatrix} \mathbf{m}^0 & \left(\frac{1}{r}\right)\mathbf{m}_v^0 & \left(\frac{1}{1-r}\right)\mathbf{n}_v^2 & \mathbf{n}^2 \\ \mathbf{m}_u^0 & \left(\frac{1}{r}\right)\mathbf{m}_{uv}^0 & \left(\frac{1}{1-r}\right)\mathbf{n}_{uv}^2 & \mathbf{n}_u^2 \\ \mathbf{m}_u^1 & \left(\frac{1}{r}\right)\mathbf{m}_{uv}^1 & \left(\frac{1}{1-r}\right)\mathbf{n}_{uv}^3 & \mathbf{n}_u^3 \\ \mathbf{m}^1 & \left(\frac{1}{r}\right)\mathbf{m}_v^1 & \left(\frac{1}{1-r}\right)\mathbf{n}_v^3 & \mathbf{n}^3 \end{pmatrix}. \tag{5}$$

The first and last columns of $\mathbf{C}_v$ are $\mathbf{L}$'s first and $\mathbf{R}$'s last columns respectively, ensuring C^0 continuity. The second and third columns contain $\mathbf{L}$'s tangents scaled by $\frac{1}{r}$ and $\mathbf{R}$'s tangents by $\frac{1}{1-r}$, respectively. This guarantees C^1 continuity in the v direction. The similar equation for merging in the u direction is provided in Appendix A.2.

3. Related work

Representations: The gradient mesh (Section 2.1) is a commonly used vector graphics primitive that allows users to directly control the gradient. Defined by a regular grid of patches [12], this representation is effective but has limited ability to adapt locally to variations in image feature density. Alternative representations have been developed to address this issue. Diffusion curves [13] use curve-based primitives to define gradients; yet, they do not directly control the gradient itself. Xia et al. [14] and He et al. [15] use curved Bézier triangles and [16] polygonal patches for gradient representation, achieving greater flexibility; yet, higher-order continuity can be an issue.

Several gradient meshes using approximating or interpolating *subdivision schemes* have been proposed. The authors of [17,18] use triangle decomposition and Loop's approximating subdivision scheme [19] for vectorization and editing, allowing for flexible mesh topologies. These approaches, however, require the triangles to be quite small for colors to be approximated well, potentially complicating the editing process. Lieng et al. [20] alleviate this issue by using Catmull–Clark's subdivision to support topologically unrestricted meshes. Yet, in all such approaches, local refinement may change the valence of faces, thereby changing the limit surface and disrupting the overall shape and continuity of the model.

Barendrecht et al. [5] introduced mathematically exact local refinement to gradient meshes, allowing deviations from the traditional gradient mesh topology. They allow splitting the mesh locally without affecting geometry or color propagation. Refice [9] extended this method to support local refinement at arbitrary parameter values and enable greater flexibility in local splitting (see Section 2.2). The authors of [5] also introduced *branching*, which allows for non-rectangular topologies, and *sharp color transitions*, which supports a fast change in color across patches without increasing mesh complexity.

Simplification: We coarsen traditional gradient meshes by converting them into a representation that supports local refinement, following the method of Barendrecht et al. [5]. We opt for this approach due to the inherent limitations of rendering subdivision surfaces and their incompatibility with standard gradient meshes.

Li et al. [21] introduced a method to simplify a traditional gradient mesh by merging adjacent patches and thereby transforming it into an alternative, sparser, representation. They leverage the cubic mean value interpolant which allows an arbitrary number of patches to converge at a vertex. However, some parameter values may become negative in non-convex regions, leading to only C^0 color surfaces. Additionally, after the simplification process, their cubic mean interpolant representation does not seem suitable for intuitive editing.

Using the local refinement approach from [5], Van der Vis [11] proposed an approximation to unify two patches into a single patch, ensuring at least C^0 continuity (see Section 2.3). Visual quality of the simplification process was measured via patch continuity. However, this approach yielded unsatisfactory visual results, as continuity does not always align with human visual perception, and is susceptible to floating-point errors. To overcome this limitation, [22] introduced pixel-based metrics to the simplification process. They rasterize the mesh both before and after merging and next compare the resulting images using established image similarity metrics.

Gradient mesh simplification with local refinement has proven effective for simple meshes (<10 patches). Extending this approach to more complex meshes remains an open challenge. One issue is that iterative merging of patches can create degenerate cases in the mesh structure, for which robust solutions have yet to be developed. We address this problem to achieve high patch reduction while maintaining visual quality.

4. Mesh simplification

4.1. Overview

Fig. 3 shows our pipeline for simplifying gradient meshes using local refinement. The input and output gradient meshes are composed of bicubic Hermite patches (Section 2.1), leveraging the locally refinable representation in the process (Section 2.2). We also accept input meshes with existing T-junctions and meshes with branching and sharp color

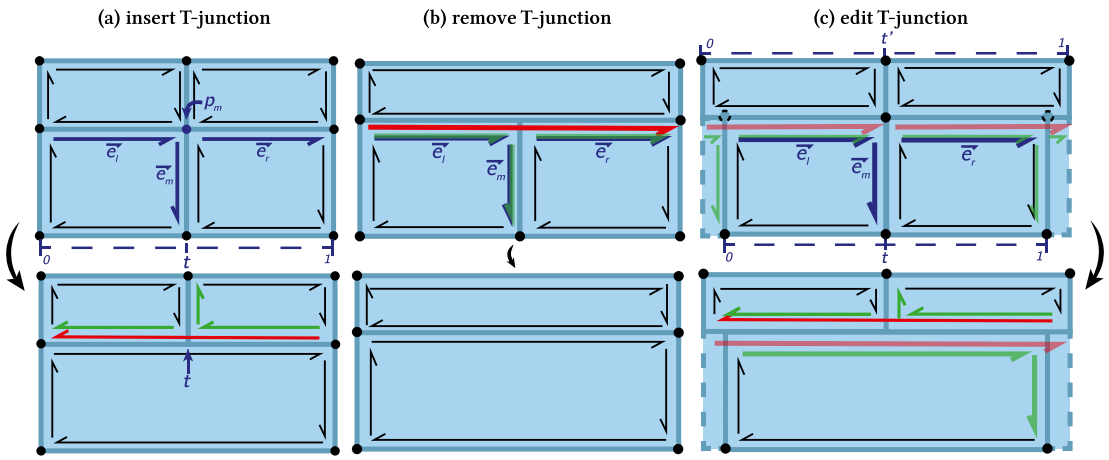


Fig. 4. (a) Merging two faces at half-edge $\vec{e}_m$ using the notation described in Section 4.3 (top) and subsequently inserting a T-junction (bottom). Here, t is the splitting factor of the two patches incident to $\vec{e}_m$. (b) Merging at a stem $\vec{e}_m$ removes the T-junction $\vec{e}_m$ belonged to. $\vec{e}_m$, $\vec{e}_l$, and $\vec{e}_r$ are children edges (marked green). (c) Merging an edge $\vec{e}_m$ where editing existing T-junction(s) is required. $\vec{e}_l$ and $\vec{e}_r$ belong to different T-junctions (top) and after merging, their parent edges are combined (bottom). t' is the reparameterized t value computed by Eq. (6).

transitions. To simplify a gradient mesh while preserving its visual quality within an acceptable margin, we impose a threshold ϵ for the error of the simplified mesh.

Simplification starts with computing an image $\mathcal{R}_o$ of the original mesh at a user-defined resolution (Section 5). We next select a half-edge for evaluation, either manually or automatically (Fig. 3a, Section 6). The edge is checked to find if it is a valid merge edge (Fig. 3b, Section 4.2). If valid, the edge is merged as described in Section 2.3, along with the necessary data structure operations to handle T-junctions (Fig. 3c, Section 4.3). The mesh with the merged edge is then rasterized and compared against $\mathcal{R}_o$ (Fig. 3d, Section 5). If the error is below ϵ, we accept the merge (Fig. 3e); else, we revert to the previous state (Fig. 3f). Another edge is then selected for evaluation (Fig. 3g). The process ends when all edges have been checked (Fig. 3h).

4.2. Defining valid merge edges

Not all half-edges in a mesh can be merged, so we must check the selected half-edge candidate before attempting to merge it. For such an edge to be mergeable, it has to comply with the following conditions:

1. It points to a valid twin;
2. Its twin is not a parent;
3. It is not a bar child of a T-junction;
4. It does not create a cycle upon merging (Section 4.4);
5. It is not part of a U/L branch case (Section 4.4).

In detail: (1) excludes boundary half-edges; (2) and (3) exclude twin pairs that differ in length due to T-junctions. (4) and (5) prevent degenerate cases from occurring in the mesh.

4.3. Handling hierarchical half-edges

Merging faces creates or removes T-junctions in the mesh, thereby introducing *hierarchical half-edges* in the DCEL, acting as parent or child nodes in a tree. While merging normal half-edges is straightforward, merging hierarchical half-edges requires more careful handling. We outline the cases of creating, deleting, and editing T-junctions below, using the following notation (see Fig. 4a): $\vec{e}_m$ is the merge edge with origin p_m; $\vec{e}_l = Prev(\vec{e}_m)$ is the edge pointing toward p_m lying immediately clockwise of $\vec{e}_m$; and $\vec{e}_r = Next(Twin(\vec{e}_m))$ points outward from p_m and lies immediately counterclockwise of $\vec{e}_m$.

Inserting T-junctions: Merging patches requires adding T-junctions unless the tail of $\vec{e}_m$ lies on a boundary or $\vec{e}_m$ is a stem (see Fig. 4a). In the basic case, the T-junction is added at parameter t, where the newly merged patches were found to be split. The two half-edges forming the bars of the new T-junction are $Twin(\vec{e}_r)$ and $Twin(\vec{e}_l)$. The new stem is defined by $Next(Twin(\vec{e}_r))$. A new parent must be added, representing the entire span of the new bars. All new children point to this parent and its intervals are updated according to t.

Removing T-junctions: If $\vec{e}_m$ is a stem, $\vec{e}_l$ and $\vec{e}_r$ must share the same parent (along with $\vec{e}_m$) and can be merged without computing a split factor t (see Fig. 4b). If $\vec{e}_l$ and $\vec{e}_r$ are the only two bar children of their parent, they can be directly combined by using the parent edge. If the parent points to other bar children, then the combined half-edge is given by the combined interval of $\vec{e}_l$ and $\vec{e}_r$. This effectively removes the T-junction $\vec{e}_m$ was part of. Moreover, because there is no control point at the tail of the half-edge, a T-junction is not added.

Combining parent half-edges: It is possible that $\vec{e}_l$ or $\vec{e}_r$ are bar half-edges belonging to parent(s) unrelated to $\vec{e}_m$ (see Fig. 4c). In such cases, t must be reparameterized to scale the control handles of the parent edges of $\vec{e}_l$ and/or $\vec{e}_r$, as well as to add a T-junction. Let s be the fraction of the parent curve spanned by the subinterval of $\vec{e}_l$; let q be the fraction of the parent curve spanned by the subinterval of $\vec{e}_r$. The reparameterized t is given by

$$t' = \frac{1}{1 + r_1 + r_2} \text{ with } r_1 = \frac{1-t}{t}s,\ r_2 = \frac{1-q}{q}r_1. \tag{6}$$

4.4. Degenerate cases

Merging introduces T-junctions into the mesh, leading to dependencies between half-edges that can accumulate over several merges. This can result in a degenerate case known as a *cycle*, where it becomes impossible to define the geometry of one or more faces. Two types of cycles exist: *next* cycles and *origin* cycles (see Fig. 5).

A next cycle occurs when four parent edges form a loop where an edge's next pointer eventually points back to itself. An origin cycle arises when the parent–child dependency structure forms a loop rather than a tree. While the augmented DCEL can model cycles, they cannot be rendered properly. Rendering a curve (and thus a patch) requires both its origin and endpoint to be explicitly defined or interpolated from its parent curve. In a next cycle, rendering fails because the endpoint of a half-edge depends on the origin of its next half-edge, which is part of the loop. Similarly, an origin cycle fails to render because generating its origin point also forms a loop. Since cycles cannot be rendered properly, any edge merge that induces a cycle must be excluded from edge selection (Section 4.2, Point 4). Moreover,

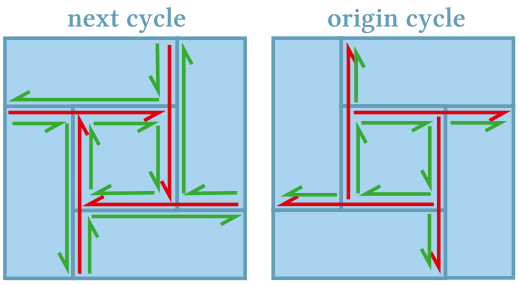


Fig. 5. *Next* cycle, where the parent's "next" pointer loops back to itself, creating a closed cycle; and *origin* cycle, where interpolating the origin point of the parent edge also loops back to itself. Parent edges are red; child edges are green. In the origin cycle, parent edges simultaneously act as stem children.

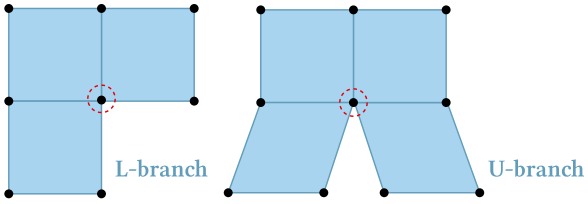


Fig. 6. *L-branch* and *U-branch* cases appear from meshes with branching, where the circled control point – a boundary point incident to more than two faces – must be preserved during simplification.

while cycles are avoided during merges, they present challenges for full simplification, leaving the affected regions of the data structure unsimplifiable (Section 6.1).

Meshes that feature branching may have boundary control points that are incident to more than two faces. Removing such control points creates a degenerate case, as the point is needed for rendering the third and/or fourth incident face. An *L-branch* appears when three faces are incident to a boundary point; an *U-branch* appears when four faces are incident to a boundary point (see Fig. 6). Any half-edges incident to such control points must be preserved — see condition 5 in Section 4.2.

5. Merge error function

As already explained, an error function is needed to evaluate the impact of merging two patches on the overall mesh quality. Earlier approaches use for this the geometric and color continuity of the two patches being merged [11]. However, this ignores errors introduced in the gradient mesh due to discontinuities with neighboring patches. Moreover, continuity differences do not always translate to perceptible visual differences, as continuity does not directly correspond to human perception. To address these limitations, we propose a novel error function that builds on the rasterization-based approach of [22], as follows. We compute a rasterized image of the entire mesh before simplification begins (Section 4.1). A second image is computed from the simplified mesh. Next, we compare the two images using established image distance metrics that range from 0 (perfectly similar) to 1 (perfectly dissimilar). We perform the merge if this distance, or error, is below a given threshold ϵ. The parameters of the entire process are detailed next.

Image distance metric: We use the SSIM (Structural Similarity Index Measure, [23]) and ꟻLIP [24] metrics. SSIM focuses on structural and perceptual fidelity; ꟻLIP emphasizes human visual system sensitivity. Both metrics produce per-pixel distance maps, which we aggregate by computing the mean. Note that we use $1-$SSIM as the distance measure.

Error threshold: ϵ gives the maximum allowable error ranging from 0 to 1. A smaller ϵ imposes stricter similarity but limits the number of merges; a larger ϵ acts conversely. Suitable ranges for ϵ, found empirically, are $[0.005, 0.01]$ for SSIM and $[0.01, 0.02]$ for ꟻLIP – see Table 4 and Fig. 14.

Table 1
Comparison of algorithms based on preprocessing and total complexity for a mesh of E edges.

Algorithm	Preprocessing	Total
Random	None	$O(E)$
Grid	None	$O(E)$
Dual-Grid	None	$O(E)$
Motorcycle	$O(E)$	$O(E)$
GQE	$O(E^2)$	$O(E^2)$
GQE 1-Step	$O(E^2)$	$O(E^4)$

Raster resolution: We rasterize images to encompass the mesh's bounding box, with a few pixels added for padding, at a user-specified resolution. Higher resolutions improve accuracy at the cost of increased computation time. Lower resolutions work conversely. In practice, we found that resolutions within 700×700 to 1000×1000 pixels balance well accuracy *vs* efficiency.

6. Edge selection

Edge selection plays a crucial role in mesh simplification. The simplest approach involves the user manually choosing the edge to merge (see Appendix B.2). We also examine several automated methods with different balances of simplification quality *vs* computational complexity (see Table 1). First, we explore several naive approaches where the edge merging order is determined in constant time using simple heuristics (Section 6.1). We next explore methods that preprocess the whole mesh to help finding the merge order: In Section 6.2, we propose a method that first computes error metrics for all edges in linear time, then reaches a simplification of desired quality. In Section 6.3, we propose a method that preprocesses the mesh in quadratic time to support different edge-selection heuristics next.

6.1. Naive edge selection

Naive edge-selection methods are simple, fast, but potentially leading to limited simplification. We describe three naive strategies below: *random*, *grid*, and *dual-grid* edge selection.

Random: Edges are chosen randomly from the mesh until all edges have been visited. This method involves minimal computation and serves as a baseline approach. While fast, its lack of structure when checking edges can accumulate T-junctions, which restricts certain edges from being merged in the next iterations due to topology changes. The randomness can also reduce the number of edge candidates since some edges become invalid because they introduce cycles upon merging.

To counter the above, we propose two edge orderings with more structure: grid and dual-grid (see Fig. 7). For both, we start by finding a corner of the mesh, i.e., a face where two consecutive edges are boundary edges. The two non-boundary edges from the corner are labeled as $\vec{e_A}$ and $\vec{e_B}$, where $\vec{e_A}$ acts as the first merge edge and $\vec{e_B}$ is used in the second iteration. Our method has two passes: the first pass considers all edges $\mathcal{E}_1$ lying in the direction of $\vec{e_A}$; the second pass considers all edges $\mathcal{E}_2$ in the orthogonal direction, aligned with $\vec{e_B}$.

Grid (Fig. 7a): $\vec{e_A}$ is the first edge processed in its row of faces. We then proceed through each successive edge in the row until reaching the boundary. Next edges in the row are given by $Next(Next(Twin(\vec{e})))$ where $\vec{e}$ is the current edge. The next row of faces is given by the edge directly across the X-junction from $\vec{e_A}$. We repeat this process until all rows are checked. In the second pass, we check edges in the orthogonal direction, starting from $\vec{e_B}$. Note that all orthogonally-facing edges can

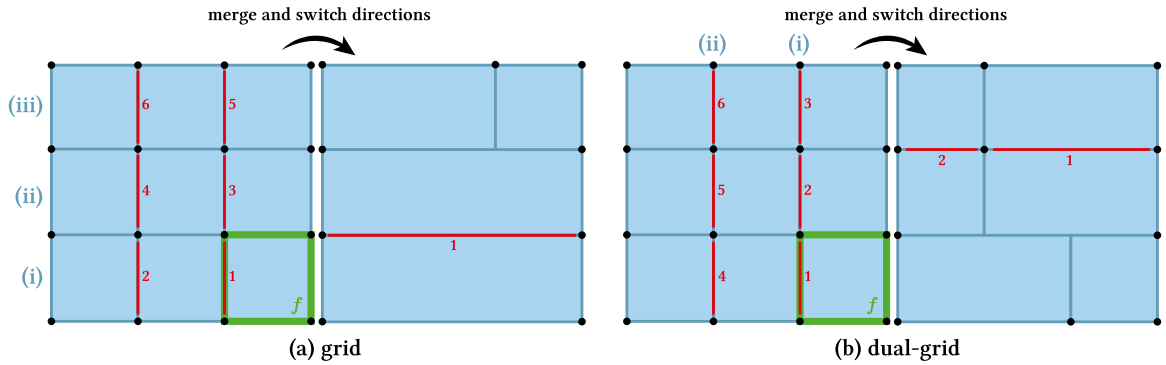


Fig. 7. Examples of grid (a) and dual-grid (b) edge orderings on a 3 × 3 face mesh. The starting corner of the mesh is represented by f. The rows of faces are labeled with Roman numerals, corresponding to the real mesh for the grid method and the dual mesh for the dual-grid method. Red numbers indicate the order in which edges are processed within a single iteration. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

be collected during the first pass by storing the set $\{\text{Next}(\vec{e}) \mid \vec{e} \in \mathcal{E}_1\}$ prior to merging.

Dual-grid (Fig. 7b): The dual-grid algorithm operates similarly but considers the rows of faces of the *dual* mesh, where faces correspond to vertices and vice versa. As such, $\vec{e_A}$ is the first merge edge and proceeds in its row with the edge directly across its X-junction. The next dual row is given by the next edge in the row of nondual faces of $\vec{e_A}$. Once all dual rows are checked, edges in the orthogonal direction are considered, starting from $\vec{e_B}$.

6.2. Mesh partitioning edge selection

Preprocessing the mesh allows comparing edges prior to merging and enables more complex heuristics that yield stronger simplifications. Naive methods may permit an edge with a higher error (though still below ϵ) to be merged early, potentially limiting subsequent merges. The creation of T-junctions during merging can also reduce the edges that can potentially be merged due to topological restrictions, even in structured orderings like grid and dual-grid. To alleviate these issues, we propose methods that partition the mesh into tensor product region quads before merging. For a detailed overview, we refer to Appendix B.3.

Our first method involves preprocessing the mesh by iterating over every edge e_i and collecting the error E_i that merging e_i individually would introduce into the mesh. We then mark edges whose error exceeds $\kappa \cdot \epsilon$ or any edges that are unmergeable (Section 4.2). We partition the mesh into tensor product regions such that all marked edges lie on the border of these regions. We execute this partitioning by constructing a *motorcycle graph* as described in detail below. Once the partition is computed, we sort regions in ascending order based on $\sum_{e \in \text{region}} E_i$, and merge edges in each partition in this order using any of the naive methods listed in Section 6.1. Results shown further in this paper use the grid method and $\kappa = 0.05$.

Our partitioning approach (Fig. 8) utilizes a variation of the motorcycle graph, a structure also used for canonical partitioning of quad meshes in 3D [25]. A typical motorcycle graph is formed by tracing straight-line paths of particles moving at a certain velocity, stopping upon intersecting another particle's path. Instead of tracing all particle paths at once, we trace complete paths over each particle iteratively: After marking all edges as described above, we iterate over all mesh vertices, identifying two types of vertices where particle paths need to be extended based on vertex valence, i.e., the number of incident marked edges. For vertices with a valence of 1, the path can be extended in the direction directly opposite to the marked edge or by tracing out the two paths orthogonal to the marked edge. Paths are extended in a straight line within the mesh structure until they collide with another marked edge; we choose the shorter path of the two

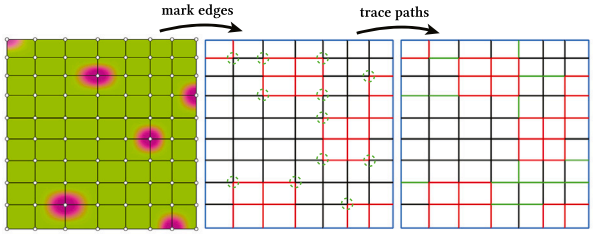


Fig. 8. Constructing a motorcycle graph from an example input mesh. Edges are first marked: red edges are single merges with error > ϵ; blue edges are unmergeable. Special valence vertices are identified (circled in green), and new paths (green edges) are traced from them. Black edges are unmarked and represent internal edges of the final partition. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

and next mark the edges along this path. For vertices with a valence of 2 where the marked edges are adjacent (forming an L-shape), two single paths can be extended in either direction; we similarly choose the shorter path. For higher-valence vertices, no path needs to be grown, since their local configuration already defines a valid quad partition. This process produces a pseudoforest, where each connected component contains at most one cycle.

6.3. Using tensor product regions

We present an algorithm that first processes and merges every tensor product region (TPR) in the mesh, aiming to find an optimal mesh partitioning that maximizes the size of merged regions while satisfying error constraints. In a quad mesh without branching, a TPR can be defined by any two edges of differing faces. For each TPR t_i, we store the error E_{t_i} it introduces to the mesh and the number of patches S_{t_i} it spans. We then model the set of all TPRs as a conflict graph, where edges between nodes indicate overlapping regions (see Fig. 9).

Given such a conflict graph, our simplification selects and merges an independent set of graph nodes I such that $\sum_{t_i \in I} E_{t_i} < \epsilon$. To maximize the number of merged faces while satisfying this error constraint, we propose two heuristics. Both heuristics rely on a heuristic

$$h(S, E) = \omega S - (1 - \omega)E, \tag{7}$$

that considers region size (S) and error (E) of a TPR, both normalized using min–max normalization. The weight ω determines how regions are prioritized, balancing between minimizing error and maximizing size. Good presets for ω are in the range $[0.7, 1)$; see Section 7. Our first heuristic, called *greedy quad-error* (GQE), operates as follows: I starts

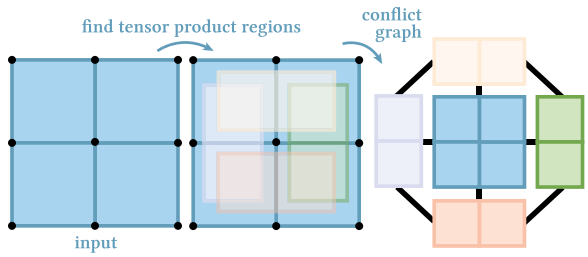


Fig. 9. Constructing the conflict graph of tensor product regions constructed over an example 2×2 face mesh.

as an empty set. We then iterate over all TPRs, sorted in descending order on $h(S, E)$. A TPR is added to I if it does not share an edge with any TPR already in I.

The second heuristic, *one-step greedy quad-error*, uses lookahead to better find the independent set that maximizes the size of merged regions. Instead of being scored in isolation, a TPR t_i is evaluated based on its combined score with the non-adjacent TPR that yields the highest score among all possible pairs involving t_i. This selection is done using a heuristic function

$$h(S_1, S_2, E_1, E_2) = \omega(S_1 + S_2 - 1) - (1 - \omega)(E_1 + E_2). \tag{8}$$

Here, $S_1 + S_2 - 1$ accounts for the fact that simplifying a pair of TPRs adds an extra face compared to a single TPR. If, for a given TPR t_i, $h(S, E)$ yields a better score than any of the scores $h(S_1, S_2, E_1, E_2)$ t_i is a part of, we use further $h(S, E)$. Like for GQE, we sort all TPRs by $h(S_1, S_2, E_1, E_2)$ and greedily construct I. Note that the preprocessing step in 1-step GQE is identical to that of GQE; however, evaluating $h(S_1, S_2, E_1, E_2)$ incurs a quadratic growth relative to GQE (see Table 1). Despite this, for meshes of roughly under 200, the total runtime is mainly given by preprocessing TPRs. This is because applying image distance metrics in the per-edge error function is significantly more expensive—thousands of operations per pixel—than the basic operations involved in independent set construction.

The sum $\sum_{t_i \in I} E_{t_i}$ is only an approximation of the actual error e in the mesh. Hence, the simplified mesh obtained by merging our selection of non-overlapping TPRs may yield higher error than $\sum_{t_i \in I} E_{t_i}$. We therefore run the GQE algorithm $N = 10$ times using a binary search to find the highest value γ that satisfies both $\sum_{t_i \in I} E_{t_i} \leq \gamma$ and $e \leq \epsilon$, where the lower bound is 0 and the upper bound is 2ϵ.

7. Results

We implemented our methods in C++20 with OpenGL 4.0. We tested on both synthetic and real-world meshes (see Appendix C for the full list and how we chose these meshes). For all experiments, we use a raster resolution of 700×700.

We begin by comparing the six edge-selection algorithms (Section 6) on various meshes. Experiments in Table 2 use the SSIM error metric with $\epsilon = 0.005$; those in Table 3 use ꟻLIP with $\epsilon = 0.02$. Since the random, grid, and dual-grid methods are nondeterministic, we average their results over 20 iterations. For the GQE and 1-step GQE methods, we set $\omega = 0.75$. After applying each simplification method, we measure the final number of patches in the simplified mesh, the final error (for nondeterministic methods: the error of the most simplified iteration), and the total processing time.

Fig. 10 tests our algorithms on a "perfectly mergeable" gradient mesh which can be fully merged into a single patch with minimal error. For both error metrics, all methods, except the random approach, consistently converged to a single patch. The random method occasionally reached the optimum but often became stuck in partially-merged states with no valid edges left to merge. This is because the arbitrary nature of random selection leads to states where the only topologically valid merge edge(s) would form a cycle upon merge (Section 4.4).

Fig. 11 next tests the simplification of a perturbed version of the mesh in Fig. 10, which is still simple but contains a few edges that, when merged, induce a high error. We see that the random method seldom reaches the simplification level of other methods, as it is prone to get trapped in suboptimal states due to topological restrictions. We also observed that all naive methods can prematurely stop the merging process because one of the earlier-merged edges introduces a higher error (still below ϵ) which consumes much of the allowable error budget. In contrast, methods using more advanced heuristics consistently found the "perfectly" simplifiable largest region as suitable for merging, resulting in a coarser final mesh.

The quality of simplification using grid and dual-grid methods can vary greatly on more complex meshes depending on the starting corner of the algorithm. In Fig. 13, we show two iterations where grid and dual-grid methods produced significantly different results due to the choice of this starting corner. Grid and dual-grid also have an issue of error propagation, where the same patch may merge repeatedly with its adjacent patch, causing the error to build up and spread across a larger region. In contrast, the random method does not suffer from this issue, as the likelihood of the same patch being selected for repeated merging is much smaller.

GQE and 1-step GQE noticeably outperformed all other methods, including the motorcycle method, which also uses mesh partitioning (see Figs. 12 and 13). The main limitation of the motorcycle method is that $\sum_{e \in \text{region}} E_i$ does not accurately estimate the error introduced by the entire merged region. Single-edge error E_i alone cannot account for the error propagation across multiple faces within a region — see Fig. 12 for a visible change in gradient over a partition. The motorcycle method's partitioning is also hard to control: Too many marked edges (strict partitioning) rule out many merge edges; if conditions are looser, a high-error edge still has a chance to be merged early. Finding a good balance between the two remains challenging, and in contrast, GQE more effectively finds feasible regions to merge. Still, in most tests, motorcycle significantly outperformed all naive methods (see Tables 2 and 3).

Apart from GQE and 1-step GQE, all other methods eventually end once a visible artifact appears in the mesh. In contrast, GQE-based methods produce much coarser simplifications with less noticeable regions of error, as simplification is more evenly spread over the mesh. Based on these findings, we chose GQE as the default method for subsequent experiments. Notably, in all our tests, GQE yielded the same results as 1-step GQE but was significantly faster for larger meshes.

Using GQE ($\omega = 0.75$), we also tested the effects of adjusting the parameters of the merge error function (Section 5), i.e., the choice of error metric and the error threshold ϵ. Table 4 shows simplifications of the same set of meshes using both SSIM and ꟻLIP while varying ϵ. ꟻLIP typically produces more conservative results compared to SSIM. For instance, in Fig. 14, achieving a simplification equivalent to SSIM at $\epsilon = 0.005$ requires ꟻLIP to use an ϵ value of at least 0.01. ꟻLIP is also much better at preserving color gradients. In Fig. 14, even at a relatively high ϵ of 0.03, ꟻLIP keeps smooth color transitions throughout the mesh. In contrast, SSIM at $\epsilon = 0.005$ shows a clear color discontinuity in the leaf region, and visible non-smooth regions for simplifications with higher thresholds. Yet, SSIM is significantly twice as fast as ꟻLIP.

We further tested the simplification levels of ꟻLIP for varying ϵ values (see Table D.6 in Appendix D). Figs. 15 and 1 show a selection of the tested meshes. We found minimal visual differences between $\epsilon = 0.001$ and $\epsilon = 0.005$, both which closely resemble the ground truth. Artifacts start to appear at $\epsilon = 0.01$, mainly in regions with relatively small patches, such as the tip of the banana or the ends of the markers in Fig. 15. At higher ϵ values, these areas are further degraded as the simplification sacrifices detail to achieve larger merges.

Our final test examines the GQE method for varying ω values. We test on the mesh in Fig. 16 using ꟻLIP and $\epsilon = 0.01$. The two extremes

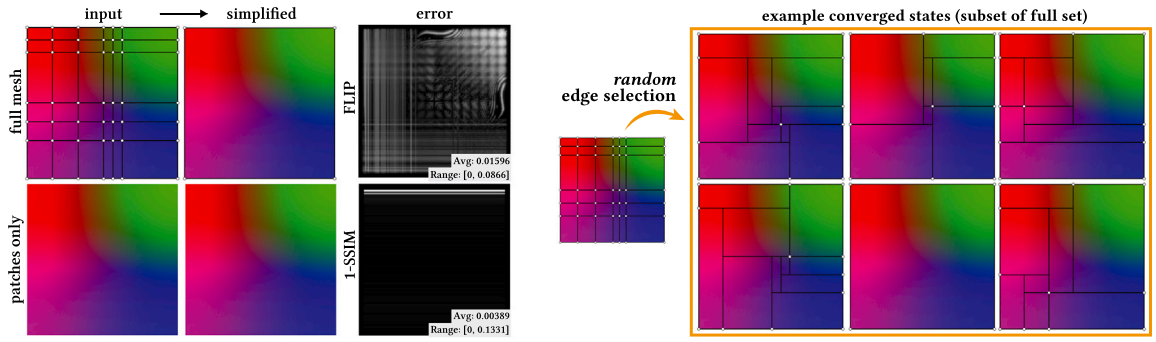


Fig. 10. Results and min–max error map of fully simplifying a "perfectly mergeable" 6 × 6 face mesh (left). All methods converge to a single patch, except for random, which can converge to other partially merged states (right).

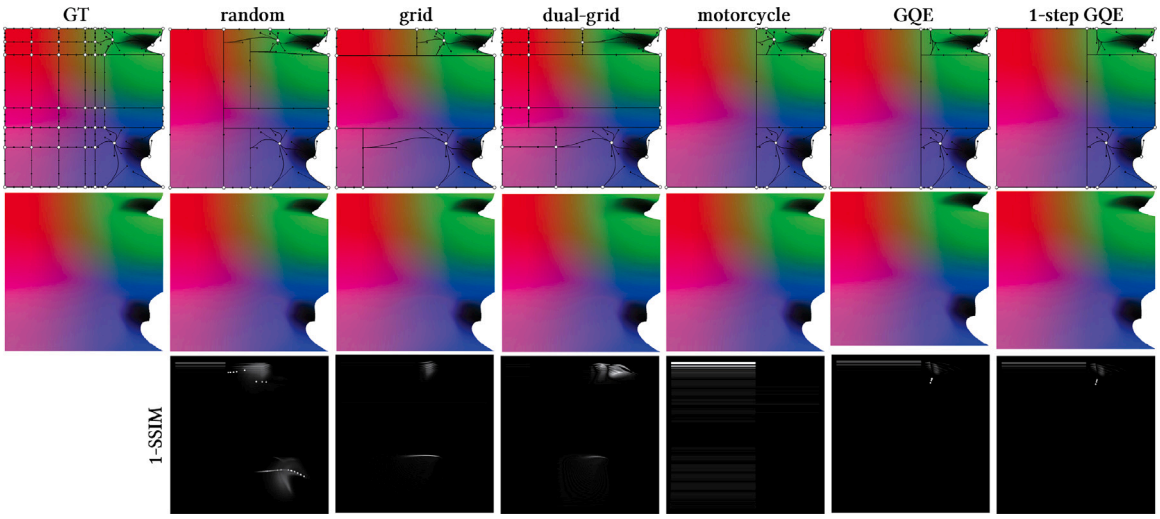


Fig. 11. Comparing all six edge selection methods on a perturbed 6 × 6 face mesh. Random, grid, and dual-grid methods are shown with a result from a single iteration. GQE and 1-step GQE consistently perform the best, followed by motorcycle. The naive algorithms can reach similar quality simplifications but depend on deterministic choices.

Table 2
Results showcasing all six edge selection methods, using the SSIM error metric, $\epsilon = 0.005$, and 700 × 700 raster resolution (#P = number of patches, #SP = number of patches in simplified mesh (lower is better), E = error, (if nondeterministic, the error of the most simplified result), T = time in seconds, 1SGQE = 1-step GQE). The best result(s) for each measure is bolded.

Mesh	#P		Random	Grid	Dual-Grid	Motorcycle	GQE	1SGQE
Fig. 10: Perfect6	36	#SP	2.75	**1**	**1**	**1**	**1**	**1**
		E	0.00389	0.00389	0.00389	0.00389	0.00389	0.00389
		T	26.8	**24.4**	**24.4**	36.8	675.6	685.6
Fig. 11: Perturbed6	36	#SP	13.95	12.9	11.85	10	**9**	**9**
		E	**0.00158**	0.00452	0.0050	**0.00158**	0.00241	0.00241
		T	24.0	**22.8**	23.2	36.0	374.0	376.4
Fig. 15: Banana	92	#SP	68.3	76.25	69.8	66	**53**	**53**
		E	0.005	0.005	0.005	0.005	**0.00484**	**0.00484**
		T	52.8	49.6	**48.8**	68.4	862.8	879.6
Fig. 12: Watermelon	221	#SP	104.4	188.15	171.85	120	**78**	**78**
		E	0.005	0.005	0.005	0.005	**0.00478**	**0.00478**
		T	221.6	**205.2**	210.0	300.8	4461.6	18 125.6

– favoring only the largest quad regions or solely minimizing error – perform the worst. The former allows large regions to dominate the error budget (e.g., the stem artifact at $\omega = 0$ in Fig. 16). The latter focuses on small, low-error regions, filling the independent set with suboptimal merges and excluding slightly larger regions with better size-to-error trade-offs. As Fig. 16 shows, $\omega = 0.75$ performs significantly better than other weights. Additional tests on other meshes, such as the one in Fig. 18, using SSIM and $\epsilon = 0.005$, confirm this trend. For these tests, ω values in the range of 0.75 to 0.9 achieve significantly better simplifications. We conclude that the heuristic performs best for $\omega \in [0.7, 1)$.

Scalability. Let us now focus on computational scalability. We further explored this running our tests on two different configurations as follows:

- *configuration C1:* Intel i7-865OU CPU, Intel UHD Graphics 620 GPU, 16 GB RAM. Both SSIM and ꟻLIP use single-threaded unoptimized CPU code;

Table 3
Results showcasing all six methods, using the ꟻLIP error metric, $\epsilon = 0.02$, and 700×700 raster resolution, with notations consistent with Table 2.

Mesh	#P		Random	Grid	Dual-Grid	Motorcycle	GQE	1SGQE
Fig. 10: Perfect6	36	#SP	3.90	**1**	**1**	**1**	**1**	**1**
		E	0.01596	0.01596	0.01596	0.01596	0.01596	0.01596
		T	**54.6**	60.7	55.4	99.2	1023.5	1024.7
Fig. 11: Perturbed6	36	#SP	18.6	17.6	15.45	12	**9**	**9**
		E	0.01903	0.01785	0.01053	**0.0056**	0.01566	0.01566
		T	61.7	66.4	**56.8**	92.2	836.6	836.9
Fig. 15: Banana	92	#SP	68.6	66.8	62	47	**37**	**37**
		E	0.01999	0.01999	0.01999	**0.01404**	0.01965	0.01965
		T	145.2	**134.1**	135.6	168.5	3729.2	3751.4
Fig. 13: Duck	230	#SP	171.8	204.35	161.7	145	**77**	**77**
		E	0.02	0.02	0.02	**0.1991**	0.01995	0.01995
		T	180.1	**167.2**	173.2	231.7	22 621.1	46 602.1

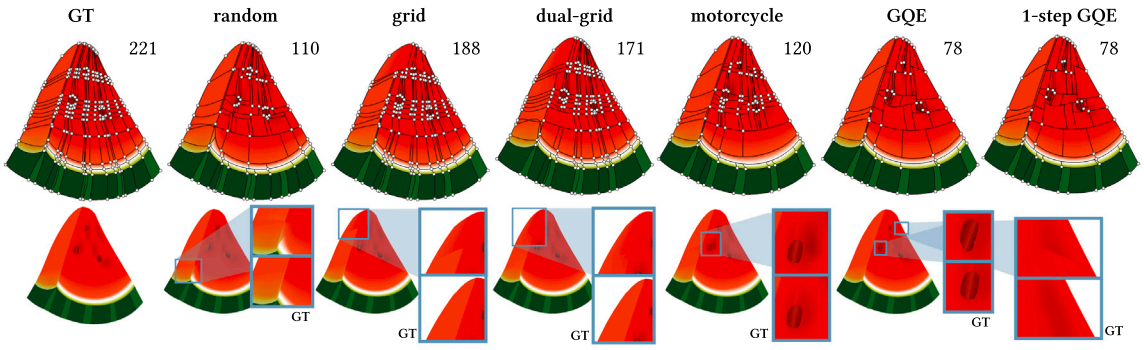


Fig. 12. Comparing all six edge selection methods on a mesh with 221 patches, where each simplified mesh is annotated with its number of remaining patches. Random, grid, and dual-grid methods are shown with a result from a single iteration. GQE and 1-step GQE perform best and achieve the same exact simplification. The most visible artifacts are zoomed in on and compared with ground truth.

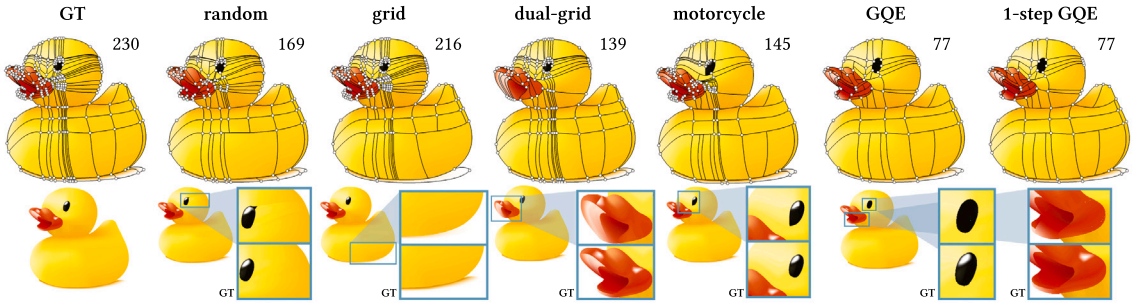


Fig. 13. Comparing all six edge selection methods on a mesh with 230 patches (same as Fig. 1). Random, grid, and dual-grid methods are shown with a result from a single iteration. GQE and 1-step GQE perform best and achieve the same simplification. Annotations follow Fig. 12.

Table 4
Number of patches in simplified mesh for SSIM and ꟻLIP at various error thresholds ϵ.

Mesh	#P	0.0005		0.001		0.005		0.01		0.02		0.03	
		SSIM	ꟻLIP	SSIM	ꟻLIP	SSIM	ꟻLIP	SSIM	ꟻLIP	SSIM	ꟻLIP	SSIM	ꟻLIP
Fig. 15: Raindrop	25	22	22	20	21	17	21	14	18	10	17	9	16
Fig. 14: Apple	80	72	76	68	74	56	67	50	57	41	51	37	47
Fig. 16: Tulips	102	66	88	64	79	54	63	45	55	37	49	33	44

- *configuration C2:* Intel i7-8700K CPU, Nvidia GeForce RTX 2080 GPU, 32 GB RAM. Code is heavily optimized with SSIM using OpenMP multithreading (12 threads) and ꟻLIP using CUDA 10.1.

As Table D.6 (column PPT) shows, the by far most costly step of our method is the preprocessing which computes the tensor product regions. In turn, this is due to the high cost of the image-based metrics SSIM and ꟻLIP. On configuration C1, the PPT phase exceeds 10 min for all tested meshes. On configuration C2, we see a significant speed-up of one to two orders of magnitude — we factor in here the fact that the CPU of C2 is roughly 3 times faster than the CPU of C1. We also see that this speed-up *drops* as the patch count P increases. This is expected as C2 only speeds up the image error metrics and not the entire computational pipeline.

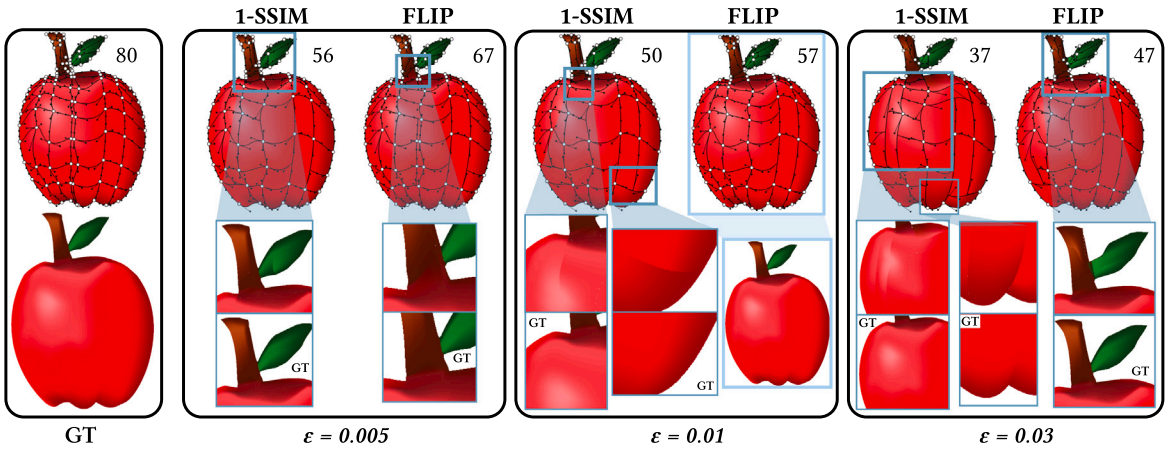


Fig. 14. Results on a mesh with 80 patches using the GQE method over several values of ϵ and comparing SSIM and ꟻLIP. Annotations follow Fig. 12. ꟻLIP is more conservative and generally simplifies with less visible artifacts, whereas SSIM is more prone to noticeable discontinuous gradient artifacts.

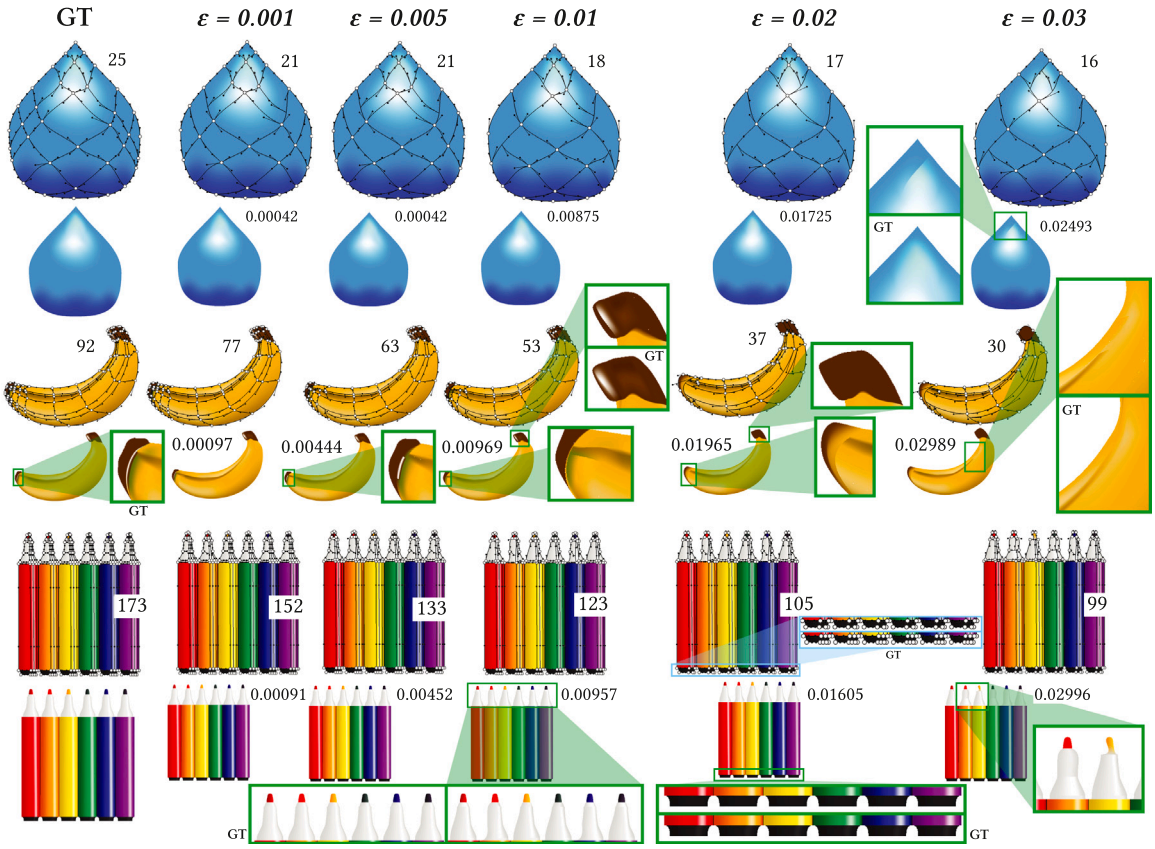


Fig. 15. Comparing simplification across a handful of meshes and varying ϵ values, using the GQE method and ꟻLIP error metric. Annotations follow Fig. 12, with exact error values also annotated next to the patches-only mesh.

We also show that our method can handle dense input meshes and simplify them based on user-selected metrics and thresholds; see Fig. 17. Our examples include a refined mesh (originally with 1225 patches) with randomly adjusted vertices and modified colors, and a chestnut model (1462 patches).

Evaluating edge-selection methods. The different edge-selection methods have different advantages and disadvantages. Random, grid, and dual-grid achieve the fastest partial simplification, with random allowing for the largest potential reduction. When a user needs a balance between simplification quality and computational cost, the motorcycle method provides the most effective compromise. When visual fidelity and maximal simplification are the priority, GQE should be employed, acknowledging its higher computational expense.

Relation to prior work. Having presented our results, we now discuss their relation to relevant state of the art methods in gradient mesh processing (cf. Section 3). The first method to allow local refinement in gradient meshes was presented in [5], but it was limited to dyadic refinement and did not provide automatic methods for mesh simplification. Several methods are based on subdivision surfaces [18,26,27], but these do not provide local simplification and are not compatible with

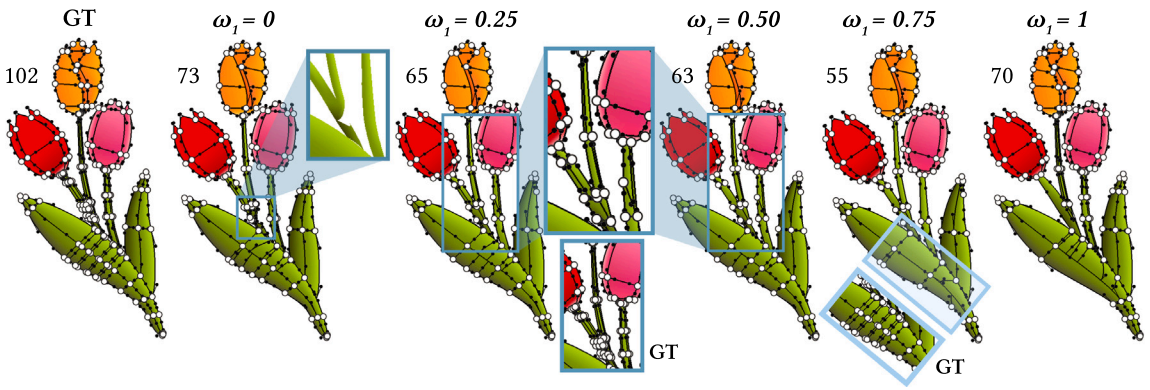


Fig. 16. Simplification results with GQE over varying ω_1 values, using the ꟻLIP error metric, $\epsilon = 0.01$, and 700 × 700 raster resolution. Annotations follow Fig. 12. The two extremes, $\omega_1 = 0$ and $\omega_1 = 1$, perform the worst, while intermediary values reach coarser simplifications, particularly doing the most simplification in the stem. $\omega_1 = 0.75$ reaches the best simplification, with the most noticeable difference in the leaf as annotated.

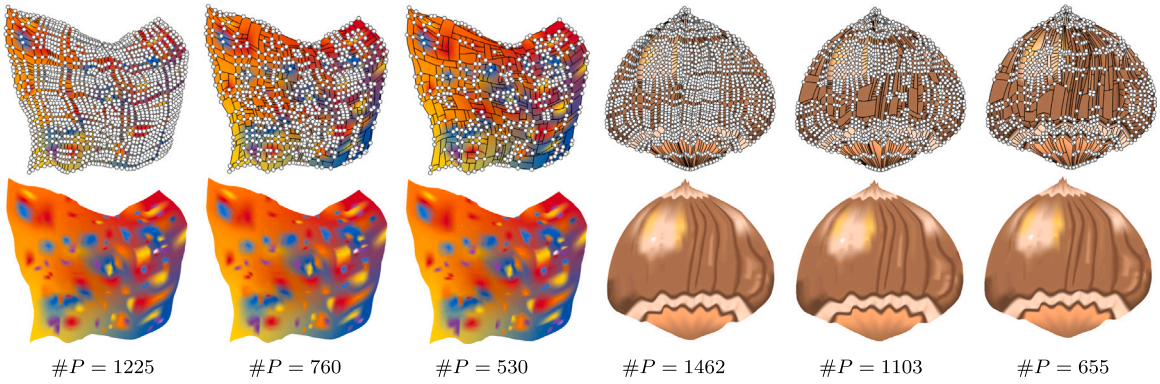


Fig. 17. Simplification results on two dense meshes. Top row: meshes with control points. Bottom row: rendered meshes. The randomly refined mesh (left) was simplified using SSIM and the chestnut model (right) was simplified using ꟻLIP. In both cases, we show the original model (left), the model simplified with $\epsilon = 0.001$ (middle), and with $\epsilon = 0.01$ (right). Under each pair, $\#P$ reports the number of patches in the (simplified) mesh.

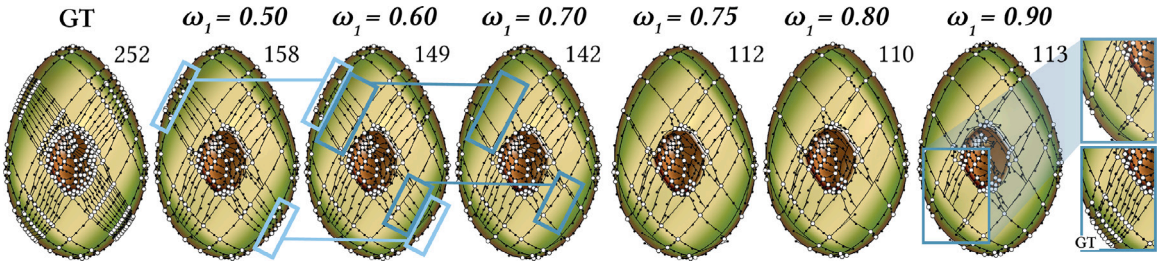


Fig. 18. Simplification results with GQE over varying ω_1 values, using the SSIM error metric, $\epsilon = 0.005$, and 700 × 700 raster resolution. Annotations follow Fig. 12. ω_1 in the range $[0.5, 0.7]$ perform significantly worse than higher values of ω_1 in the range [0.75, 0.9]. Regions that coarsen most significantly over increasing ϵ values are highlighted.

traditional gradient meshes. Finally, the method presented in [21] does provide mesh simplification, but the resulting multisided patches are often of very complex shapes and thus difficult to manually edit [21, Figure 8], and as the method relies on cubic mean value coordinates, it is not compatible with standard vector graphics formats.

In contrast, our method works directly with bicubic patches and thus enjoys full compatibility with standard vector graphics specifications, including SVG and PDF.

8. Conclusions and future work

We have presented a robust approach for simplifying traditional gradient meshes by iteratively merging adjacent patches with local refinement. Our key goal was to reduce the number of patches to a minimum while preserving the visual quality of the original mesh. We achieved this goal through our proposed merge error function and various algorithms designed to optimize the merging order.

A key limitation of our methods, especially the best-performing one (GQE), is computational speed. Processing all tensor product regions scales quadratically with respect to the number of mesh edges, yielding noticeable slowdowns when applying image distance metrics. Porting the full method, and not only the computation of the SSIM and ꟻLIP errors, to the GPU (or multithreaded CPU) would further enhance performance.

Another direction for future work is to develop methods for identifying high-error edges in a single pass rather than computing errors

iteratively for each edge. This would further accelerate the simplification process. Also, rather than computing the error globally across the mesh, it would be interesting to explore how a local error function could be applied to distribute the error more evenly across different mesh regions.

CRediT authorship contribution statement

E. Kato: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **T. Ophelders:** Writing – review & editing, Supervision, Methodology. **A. Telea:** Writing – review & editing, Validation, Supervision, Resources, Project administration. **J. Kosinka:** Writing – review & editing, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jiri Kosinka reports a relationship with University of Groningen that includes: board membership. Given his role as an Editorial Board member, Jiří Kosinka had no involvement in the peer review of this article and had no access to information regarding its peer review. Full responsibility for the editorial process for this article was delegated to another journal editor. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Exact splitting and merging

A.1. Splitting

Refice [9] extended [5] by enabling local refinement in gradient meshes for an arbitrary $t \in [0,1]$. This is done by remapping the full parameter space $[0,1]$ of a patch to two separate parameter spaces of the new patches, $[0,t]$ and $[t,1]$. Splitting a patch defined by $H(u,v)$ (see Eq. (1)) in the u direction at parameter t produces a top patch control matrix $\mathbf{T} = H(tu,v)$ and bottom patch control matrix $\mathbf{B} = H(t+(1-t)u,v)$. Let $s = 1-t$. These can be expanded to:

$$\mathbf{T} = \begin{pmatrix} H(0,0) & H_v(0,0) & H_v(0,1) & H(0,1) \\ H_u(0,0)t & H_{uv}(0,0)t & H_{uv}(0,1)t & H_u(0,1)t \\ H_u(t,0)t & H_{uv}(t,0)t & H_{uv}(t,1)t & H_u(t,1)t \\ H(t,0) & H_v(t,0) & H_v(t,1) & H(t,1) \end{pmatrix} \tag{A.1}$$

$$\mathbf{B} = \begin{pmatrix} H(t,0) & H_v(t,0) & H_v(t,1) & H(t,1) \\ H_u(t,0)s & H_{uv}(t,0)s & H_{uv}(t,1)s & H_u(t,1)s \\ H_u(1,0)s & H_{uv}(1,0)s & H_{uv}(1,1)s & H_u(1,1)s \\ H(1,0) & H_v(1,0) & H_v(1,1) & H(1,1) \end{pmatrix} \tag{A.2}$$

[9] derives the split in the other parametric direction similarly. Splitting patch $H(u,v)$ across v at parameter t, the left control matrix $\mathbf{L} = H(u,tv)$ and the right control matrix $\mathbf{R} = H(u,t+(1-t)v)$, which can be expanded to:

$$\mathbf{L} = \begin{pmatrix} H(0,0) & H_v(0,0)t & H_v(0,t)t & H(0,t) \\ H_u(0,0) & H_{uv}(0,0)t & H_{uv}(0,t)t & H_u(0,t) \\ H_u(1,0) & H_{uv}(1,0)t & H_{uv}(1,t)t & H_u(1,t) \\ H(1,0) & H_v(1,0)t & H_v(1,t)t & H(1,t) \end{pmatrix} \tag{A.3}$$

$$\mathbf{R} = \begin{pmatrix} H(0,t) & H_v(0,t)s & H_v(0,1)s & H(0,1) \\ H_u(0,t) & H_{uv}(0,t)s & H_{uv}(0,1)s & H_u(0,1) \\ H_u(1,t) & H_{uv}(1,t)s & H_{uv}(1,1)s & H_u(1,1) \\ H(1,t) & H_v(1,t)s & H_v(1,1)s & H(1,1) \end{pmatrix} \tag{A.4}$$

The corner points and the tangents orthogonal to the split direction are directly reused in the split patch. The tangents aligned with the split direction, along with all twist vectors, are also reused but scaled by a factor of t ($\mathbf{L},\mathbf{T}$) or $1-t$ ($\mathbf{R},\mathbf{B}$).

A.2. Merging

In Eq. (5), the approximation of two patches merged into a single patch across the v direction is exactly the inverse of the above split operations (Eqs. (A.3) and (A.4)). Let $\alpha = \frac{1}{r}$ and $\beta = \frac{1}{1-r}$. The same approximation across the v direction can similarly be derived from the inverse of Eqs. (A.1) and (A.2), defined as:

$$\mathbf{C}_u = \begin{pmatrix} \mathbf{m}^0 & \mathbf{m}^0_v & \mathbf{m}^2_v & \mathbf{m}^2 \\ \alpha\mathbf{m}^0_u & \alpha\mathbf{m}^0_{uv} & \alpha\mathbf{m}^2_{uv} & \alpha\mathbf{m}^2_u \\ \beta\mathbf{n}^1_u & \beta\mathbf{n}^1_{uv} & \beta\mathbf{n}^3_{uv} & \beta\mathbf{n}^3_u \\ \mathbf{n}^1 & \mathbf{n}^1_v & \mathbf{n}^3_v & \mathbf{n}^3 \end{pmatrix}$$

The parameter r, representing the splitting factor of two patches, is given by Eq. (4). The derivation of this equation is detailed below, where we establish a set of C^1 conditions for two patches expressed as linear functions of r. To ensure C^1 continuity across the v-direction, consider two adjacent Hermite patches $H_l(u,v)$ and $H_r(u,v)$ with control matrices $\mathbf{L}$ and $\mathbf{R}$, and elements denoted as $\mathbf{m}$ and $\mathbf{n}$ respectively. We solve for continuity conditions at the shared boundary:

$$\lim_{v\to r^-} \frac{d}{dv} H_l\left(u, \frac{v}{r}\right) = \lim_{v\to r^+} \frac{d}{dv} H_r\left(u, \frac{v-r}{1-r}\right) \tag{A.5}$$

$$\alpha \cdot \frac{d}{dv} H_l(u,v)\Big|_{v=1} = \beta \cdot \frac{d}{dv} H_r(u,v)\Big|_{v=0} \tag{A.6}$$

$$\alpha \cdot \mathbf{uBLB}^\top \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} = \beta \cdot \mathbf{uBRB}^\top \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{A.7}$$

$$(1-r)\mathbf{uBL} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = r\mathbf{uBR} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \tag{A.8}$$

Note that to compute $\frac{d}{dv}H(u,v)$, we use the derivative parameter vector $\begin{pmatrix} 0 & 1 & 2v & 3v^2 \end{pmatrix}$. A sufficient condition for Eq. (A.8) is:

$$(1-r)\mathbf{Y} \begin{pmatrix} \mathbf{m}^2_v \\ \mathbf{m}^2_{u,v} \\ \mathbf{m}^3_{u,v} \\ \mathbf{m}^3_v \end{pmatrix} = r\mathbf{Y} \begin{pmatrix} \mathbf{n}^0_v \\ \mathbf{n}^0_{u,v} \\ \mathbf{n}^1_{u,v} \\ \mathbf{n}^1_v \end{pmatrix}, \tag{A.9}$$

$$\text{where } \mathbf{Y} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & -\frac{1}{3} & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{A.10}$$

This provides expansions for the four conditions necessary for C^1 continuity. [11] uses the conversion matrix $\mathbf{Y}$ to transform from Hermite to Bézier representations. Solving these continuity conditions for r yields four estimations, as shown in Eq. (4). To minimize inconsistencies, these estimations are averaged, as recommended in [22].

Appendix B. User interface

We now discuss how many aspects of our research come together in our tool's UI. These include the gradient mesh representation as an augmented DCEL (Section 2.2), the ability for users to adjust parameters such as the edge selection algorithm (Section 6) and the merge error function (Section 5), as well as the preprocessing pipeline for edge selection algorithms detailed in Sections 6.2 and 6.3.

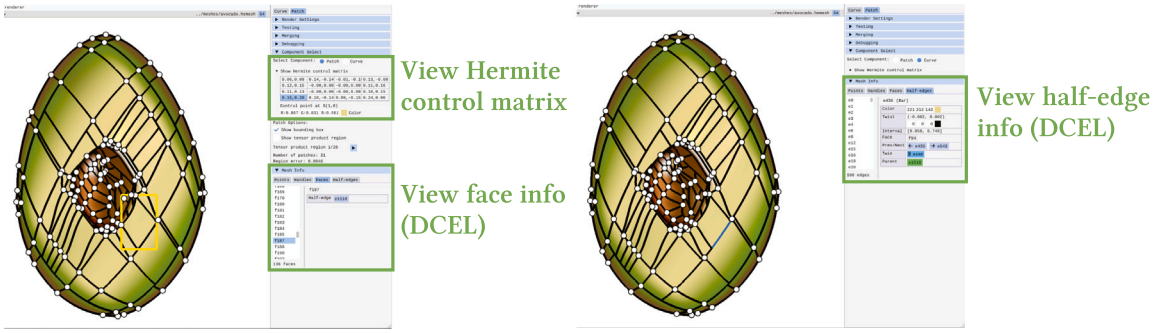


Fig. A.19. Interacting with the DCEL by clicking on a patch (yellow bounding box) and viewing its corresponding Hermite control matrix and face component information in the list of faces (left) and clicking on a curve (blue) and viewing its half-edge component information in the list of half-edges (right).

B.1. Interacting with the data structure

As detailed in Section 2.2, we employ an augmented DCEL as the core mesh data structure, which is visualized in the UI as four interconnected lists: points, handles, faces, and half-edges. This design allows users to interactively explore and navigate between the mesh and its underlying data structure. For example, in Fig. A.19, selecting a patch in the mesh reveals its direct information, such as the Hermite control matrix, and highlights the corresponding entry in the face list. Conversely, selecting a face (or any component) in the list visually highlights its counterpart in the mesh. This interactive approach extends to curves, which provide detailed half-edge information, including references to their next, previous, twin, and parent edges as *buttons*. From there, users can navigate to these related components by directly clicking the displayed pointers.

B.2. Merge controls

The user interface provides controls to adjust key parameters of the merging process, including the edge selection method and merge error function parameters (see Fig. B.20, left). These controls are located in the right pane of the UI that include sliders and dropdown menus. Additionally, we introduce a novel feature designed for manual merging, tailored for artists. Similar to the curve selection process described in Appendix B.1, the artist can left-click on a curve to select it, then right-click to preview the merge and view the resulting error introduced to the mesh. This preview enables the artist to make informed decisions about whether to proceed with the merge.

After the merging process, the user can view the final computed error between the original mesh and the simplified mesh, as well a side-by-side image comparison between the two (see Fig. B.20, right).

B.3. Preprocessing pipeline

For the motorcycle, GQE, and 1-step GQE edge selection algorithms described in this paper (Sections 6.2 and 6.3), preprocessing is required. Users can initiate this process through the UI (see Fig. B.21). To obtain the necessary data, users can either start preprocessing via the Edit menu and wait for it to complete or load preformatted data directly from a saved text file. For the former, a text file with the preprocessed data will be automatically saved to the build such that it can be easily reused in the future (using the latter method). Once the data is successfully obtained, the motorcycle edge selection method becomes accessible for single-edge errors, while the GQE methods are available for TPR-based errors. For GQE methods, another slider appears for the user to control the weights of the GQE function.

Table C.5
List of all input meshes used in the experiments (Section 7) of this paper (#P = the number of patches, SCT = sharp color transitions).

Name	#P	Branching	SCT
Fig. 15: Raindrop	25		
Fig. 10: Perfect6	36		
Fig. 11: Perturbed6	36		
Fig. 14: Apple	80	✓	
Fig. 15: Banana	92	✓	✓
Fig. 16: Tulips	102	✓	✓
Fig. 15: Markers	173	✓	✓
Fig. 12: Watermelon	221		✓
Fig. 1: Duck	230	✓	✓
Fig. 18: Avocado	252		

Additionally, we implemented features to effectively showcase the data generated during preprocessing. For single-edge error preprocessing, users can view the error map and traced motorcycle paths in a dedicated window. For TPR preprocessing, users can select a patch in the UI (Appendix B.1) and visualize all TPRs associated with that patch. Each patch is linked to a set of TPRs that users can cycle through, with each TPR displayed on the mesh. We also display the TPR's size (i.e., the number of patches it spans) and the error the region would introduce upon merging.

Appendix C. Input meshes

In this section, we discuss the rationale behind selecting the input meshes used to evaluate our results. Table C.5 lists all the meshes employed in our experiments, sorted by the number of patches in ascending order. Additionally, we indicate whether each mesh incorporates the novel features of branching or sharp color transitions, as described in [5]. These features introduce unique challenges for certain algorithms: branching complicates grid and dual-grid methods, which typically initiate merging from an arbitrary corner, while sharp color transitions, though irrelevant for edge selection, can yield distinct outcomes in our error function during merging.

To establish a baseline, we used two academic examples, *Perfect6* and *Perturbed6*. These examples allow us to evaluate our error function and edge selection methods under controlled conditions. *Perfect6* serves as a ground truth test case, as it represents a mesh with a known optimal solution that can be perfectly merged. *Perturbed6*, while lacking a mathematically defined optimal solution, serves as a more challenging example where a reasonable solution can be inferred. It demonstrates how prioritizing the largest total patch region (TPR) with minimal error consistently leads to a simplified mesh. Importantly, all methods that correctly identify the largest TPR yield results with the fewest patches in the simplified version. In contrast, naive edge selection methods

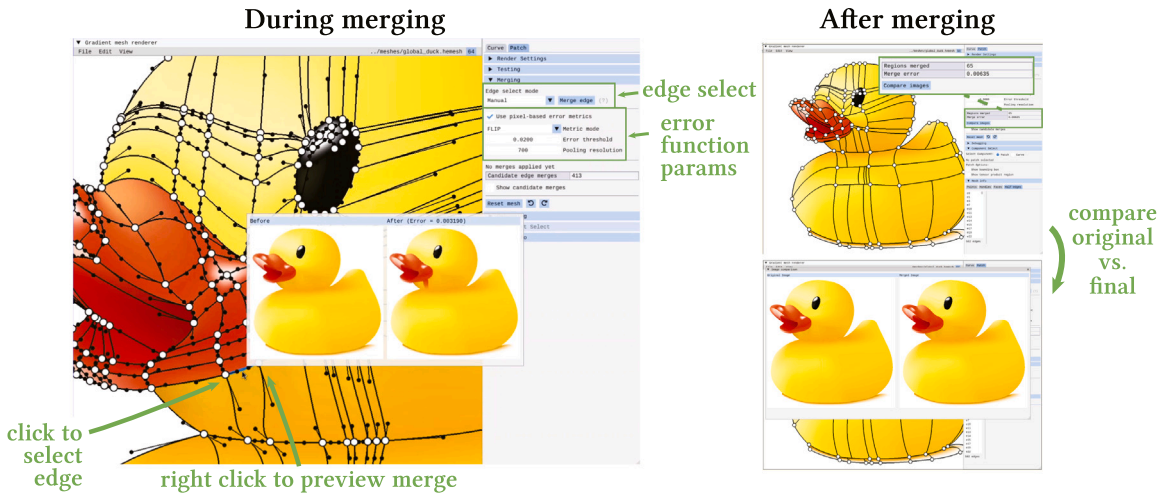


Fig. B.20. During the merge process, the user can adjust parameters in the UI, and for *manual* edge selection, the user can preview the simplified mesh and merge error before deciding to merge (left). After simplification, the user can view final error and see a final side-by-side comparison between the original image and final simplification (right).

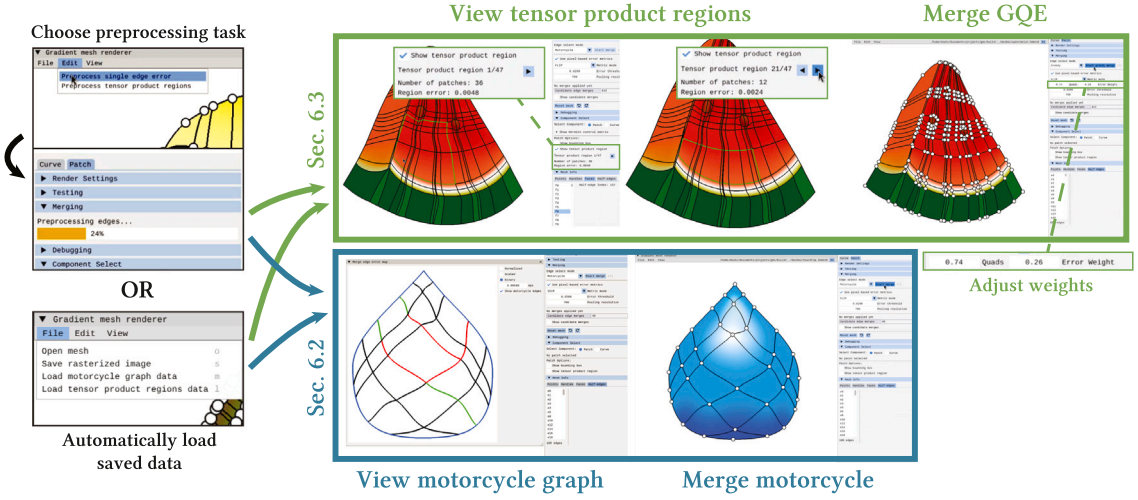


Fig. B.21. The pipeline for edge selection methods that require preprocessing is as follows. First, the user initiates a process through the Edit menu, or automatically loads saved data into the program (left). Next, the user can visualize the data directly in the program by either viewing tensor product regions or the motorcycle graph (depending on the chosen method). Lastly, the button to simplify using the chosen method becomes accessible to the user. For GQE methods, an extra UI slider also appears for controlling the function's weights.

struggle to find the largest TPR, leading to simplifications with more patches.

Beyond variations in patch count, we ensured that the selected meshes captured diverse types of patches in terms of both geometry and color. For example, meshes like *Banana*, *Tulips*, *Markers*, and *Duck* are primarily defined by concave shapes, posing unique challenges for simplifications aimed at preserving overall structure. Conversely, meshes such as *Raindrop*, *Apple*, and *Avocado* are characterized by smooth color gradients, while *Watermelon* and *Duck* feature distinct regions with sharp contrasts in both geometry and color, such as the watermelon seeds or the duck's eye. These contrasting features test our method's ability to handle varying levels of geometric and color complexity within a single mesh.

Finally, we considered the variance in patch sizes across meshes, as this significantly impacts the merge function. Some meshes, such as *Tulips*, *Raindrop*, and *Watermelon*, exhibit relatively uniform patch sizes, with the largest patches deviating minimally from the smallest. In contrast, meshes like *Duck* and *Markers* display significant patch size variance, where large regions (e.g., the duck's body or the main sections of the markers) dominate smaller, more detailed patches. This disparity amplifies the challenge of merging larger regions without incurring higher errors, underscoring the importance of our approach in balancing patch simplification with error minimization.

Appendix D. Supplementary results

Additional results can be found in Table D.6.

Data availability

https://github.com/evakato/gradmesh-simplification.

Table D.6
Comparing varying error threshold ϵ values on several input meshes, using the GQE method, ꟻLIP error metric, and 700 × 700 resolution (#P = number of patches, PPT = preprocessing time in seconds (when using both SSIM and ꟻLIP, on two different hardware platforms C1 and C2), #SP = number of patches in simplified mesh, E = error, T = real-time simplification in seconds). This table goes along with Figs. 15 and 1 in Section 7.

Mesh	#P	PPT				ϵ =	0.001	0.005	0.01	0.02	0.03
			ꟻLIP	SSIM	#SP		21	21	18	17	16
Fig. 15: Raindrop	25	C1	865.3	544.1	E		0.00042	0.00042	0.00875	0.01725	0.02493
		C2	0.86	1.36	T		22.0	22.1	22.9	23.2	22.6
			ꟻLIP	SSIM	#SP		77	63	53	37	30
Fig. 15: Banana	92	C1	4281.2	2434.3	E		0.00097	0.00444	0.00969	0.01965	0.02989
		C2	19.9	31.1	T		12.2	12.2	12.4	11.7	11.6
			ꟻLIP	SSIM	#SP		152	133	123	105	99
Fig. 15: Markers	173	C1	5449.4	3717.1	E		0.00091	0.00452	0.00957	0.01605	0.02996
		C2	32.1	25.3	T		24.7	27.7	29.3	30.2	30.2
			ꟻLIP	SSIM	#SP		169	126	99	77	58
Fig. 1: Duck	230	C1	22 600.3	13 957.8	E		0.00096	0.00492	0.00973	0.01995	0.02858
		C2	417.7	605.4	T		46.7	48.5	51.2	51.1	47.9

References

[1] X. Tian, T. Günther, A survey of smooth vector graphics: Recent advances in representation, creation, rasterization, and image vectorization, IEEE Trans. Vis. Comput. Graphics 30 (3) (2024) 1652–1671, http://dx.doi.org/10.1109/TVCG.2022.3220575.

[2] A. Inc., Adobe illustrator, 1987, URL: https://www.adobe.com/products/illustrator.html. Version history and details available online.

[3] C. Corporation, CorelDRAW graphics suite, 1989, URL: https://www.coreldraw.com. Professional graphic design software suite.

[4] I. Project, Inkscape: Open source vector graphics editor, 2003, URL: https://inkscape.org. Open source and community-driven development.

[5] P.J. Barendrecht, M. Luinstra, J. Hogervorst, J. Kosinka, Locally refinable gradient meshes supporting branching and sharp colour transitions, Vis. Comput. 34 (6–8) (2018) 949–960, http://dx.doi.org/10.1007/s00371-018-1547-1.

[6] D.R. Forsey, R.H. Bartels, Hierarchical B-spline refinement, in: Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '88, Association for Computing Machinery, New York, NY, USA, 1988, pp. 205–212, http://dx.doi.org/10.1145/54852.378512.

[7] G.J. Hettinga, J. Echevarria, J. Kosinka, Adaptive image vectorisation and brushing using mesh colours, Comput. Graph. 105 (2022) 119–130, http://dx.doi.org/10.1016/j.cag.2022.05.004.

[8] K. He, J. Roerdink, J. Kosinka, Image vectorization using a sparse patch layout, Graph. Model. 135 (2024) 101229, http://dx.doi.org/10.1016/j.gmod.2024.101229.

[9] C. Refice, Arbitrary Splitting and Rendering of Locally Refinable Gradient Meshes (Bachelor's thesis), University of Groningen, 2020.

[10] W.A.V. de la Houssaije, J. Echevarria, J. Kosinka, Palette-based recolouring of gradient meshes, Comput. Graph. Forum 43 (7) (2024) e15258, http://dx.doi.org/10.1111/cgf.15258.

[11] J. van der Vis, Simplification of Parametric-Split Gradient Meshes, Internship Report, University of Groningen, 2021.

[12] J. Sun, L. Liang, F. Wen, H.-Y. Shum, Image vectorization using optimized gradient meshes, ACM Trans. Graph. 26 (3) (2007) 11–es, http://dx.doi.org/10.1145/1276377.1276391.

[13] A. Orzan, A. Bousseau, H. Winnemoeller, P. Barla, J. Thollot, D. Salesin, Diffusion curves: A vector representation for smooth-shaded images, ACM Trans. Graph. (Proc. SIGGRAPH 2008) 56 (2008) http://dx.doi.org/10.1145/2483852.2483873.

[14] T. Xia, B. Liao, Y. Yu, Patch-based image vectorization with automatic curvilinear feature alignment, in: ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia), vol. 28, ACM, 2009, http://dx.doi.org/10.1145/1618452.1618495, 5.

[15] K. He, J. Roerdink, J. Kosinka, Monte Carlo optimization for gradient meshes, Graph. Model. 143 (2026) 101320, http://dx.doi.org/10.1016/j.gmod.2026.101320.

[16] G. Hettinga, R. Brals, J. Kosinka, Colour interpolants for polygonal gradient meshes, Comput. Aided Geom. Design 74 (2019) 101769, http://dx.doi.org/10.1016/j.cagd.2019.101769.

[17] Z. Liao, H. Hoppe, D. Forsyth, Y. Yu, A subdivision-based representation for vector image editing, IEEE Trans. Vis. Comput. Graphics 18 (11) (2012) 1858–1867, http://dx.doi.org/10.1109/TVCG.2012.76.

[18] H. Zhou, J. Zheng, L. Wei, Representing images using curvilinear feature driven subdivision surfaces, IEEE Trans. Image Process. 23 (8) (2014) 3268–3280, http://dx.doi.org/10.1109/TIP.2014.2327807.

[19] C. Loop, Smooth subdivision surfaces based on triangles, in: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, ACM Press, 1987, pp. 61–68, http://dx.doi.org/10.1145/37402.37411.

[20] H. Lieng, J. Kosinka, J. Shen, N.A. Dodgson, A colour interpolation scheme for topologically unrestricted gradient meshes, Comput. Graph. Forum (2017) http://dx.doi.org/10.1111/cgf.12862.

[21] X.-Y. Li, T. Ju, S.-M. Hu, Cubic mean value coordinates, ACM Trans. Graph. 32 (4) (2013) http://dx.doi.org/10.1145/2461912.2461917.

[22] S. Zeeman, Simplifying Gradient Meshes through the Merging of Bicubic Patches (Master's thesis), University of Groningen, 2023.

[23] Z. Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli, Image quality assessment: From error visibility to structural similarity, IEEE Trans. Image Process. 13 (4) (2004) 600–612.

[24] D. Andersson, T. Ropinski, FLIP: A difference evaluator for alternating images, ACM Trans. Graph. 39 (6) (2020) 183.

[25] J. Erickson, K. Whittlesey, Motorcycle graphs: Canonical quad mesh partitioning, in: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Geometry Processing, Eurographics Association, 2008, pp. 173–182.

[26] T.W. Verstraaten, J. Kosinka, Local and hierarchical refinement for subdivision gradient meshes, Comput. Graph. Forum 37 (7) (2018) 373–383, http://dx.doi.org/10.1111/cgf.13575.

[27] J. Zhou, G. Hettinga, S. Houwink, J. Kosinka, Feature-adaptive and hierarchical subdivision gradient meshes, Comput. Graph. Forum (2022) http://dx.doi.org/10.1111/cgf.14442.