Interactive All-Hex Meshing via Cuboid Decomposition

LINGXIAO LI, PAUL ZHANG, DMITRIY SMIRNOV, S. MAZDAK ABULNAGA, and JUSTIN SOLOMON,
Massachusetts Institute of Technology, USA

Fig. 1. A sampling of hexahedral meshes produced by our method. Our user-in-the-loop interactive pipeline facilitates the creation of high-quality hex meshes, giving the user the option to control the final result at any desired level of granularity.

Standard PolyCube-based hexahedral (hex) meshing methods aim to deform the input domain into an axis-aligned PolyCube volume with integer corners; if this deformation is bijective, then applying the inverse map to the voxelized PolyCube yields a valid hex mesh. A key challenge in these methods is to maintain the bijectivity of the PolyCube deformation, thus reducing the robustness of these algorithms. In this work, we present an interactive pipeline for hex meshing that sidesteps this challenge by using a new representation of PolyCubes as unions of cuboids. We begin by deforming the input tetrahedral mesh into a near-PolyCube domain whose faces are loosely aligned to the major axis directions. We then build a PolyCube by optimizing the layout of a set of cuboids with user guidance to closely fit the deformed domain. Finally, we construct an inversion-free pullback map from the voxelized PolyCube to the input domain while optimizing for mesh quality metrics. We allow extensive user control over each stage, such as editing the voxelized PolyCube, positioning surface vertices, and exploring the trade-off among competing quality metrics, while also providing automatic alternatives. We validate our method on over one hundred shapes, including models that are challenging for past PolyCube-based and frame-field-based methods. Our pipeline reliably produces hex meshes with quality on par with or better than state-of-the-art. We additionally conduct a user study with 21 participants in which the majority prefer hex meshes they make using our tool to the ones from automatic state-of-the-art methods. This demonstrates the need for intuitive interactive hex meshing tools where the user can dictate the priorities of their mesh.

CCS Concepts: • **Computing methodologies** → **Shape analysis**; **Mesh models**; **Volumetric models**; • **Human-centered computing** → **User interface toolkits**; **Graphical user interfaces**.

Additional Key Words and Phrases: Polycube, interactive, hex meshing

ACM Reference Format:
Lingxiao Li, Paul Zhang, Dmitriy Smirnov, S. Mazdak Abulnaga, and Justin Solomon. 2021. Interactive All-Hex Meshing via Cuboid Decomposition. *ACM Trans. Graph.* 40, 6, Article 256 (December 2021), 17 pages. https://doi.org/10.1145/3478513.3480568

Authors' address: Lingxiao Li, lingxiao@mit.edu; Paul Zhang, pzpzpzp1@mit.edu; Dmitriy Smirnov, smirnov@mit.edu; Mazdak Abulnaga, abulnaga@mit.edu; Justin Solomon, jsolomon@mit.edu, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA, 02139, US.

1 INTRODUCTION

Hexahedral (hex) meshes historically have been preferred over tetrahedral meshes in various graphics and simulation applications due to their reduced numerical error and usage of fewer elements [Shepherd and Johnson 2008]. In particular, the regular structure of layers of hex elements in a hexahedral mesh enables natural support of tensor product function bases [Liu et al. 2015] and multilevel hierarchies of nested meshes for efficient PDE solvers [Nieser et al. 2011]. However, few reliable techniques exist to generate a high-quality hex mesh that conforms to the input domain while having desirable properties such as low distortion and uniform edge lengths.

Among hex meshing methods, PolyCube-based algorithms stand out for their relative robustness. The PolyCube-based pipeline typically starts by deforming the input mesh into a *near-PolyCube*, a mesh whose surface normals are roughly axis-aligned [Huang et al.

2014]. Various heuristics and repairs can be applied to transform the near-PolyCube into a *PolyCube*, whose surface normals are exactly axis aligned [Sokolov and Ray 2015; Zhao et al. 2019]. If the deformation map from these two steps maintains bijectivity, and the PolyCube has integer corners, then one can voxelize the PolyCube and pull it back through the map to obtain a boundary-aligned hex mesh of the input domain [Fu et al. 2016; Gregson et al. 2011; Livesu et al. 2013]. The obtained hex mesh can then be improved using various techniques, from optimizing element quality [Livesu et al. 2015] to pushing boundary singularities inwards [Cherchi et al. 2019].

However, obtaining a bijective deformation map from the input domain to a PolyCube while satisfying integer constraints remains unsolved [Protais et al. 2020; Sokolov and Ray 2015]. In this work, we circumvent this difficulty by decomposing a near-PolyCube deformed from the input into a union of axis-aligned cuboids, which represents the generated PolyCube. Borrowing ideas from computer vision [Smirnov et al. 2020; Tulsiani et al. 2017], we optimize the PolyCube structure by optimizing over constituent cuboids' parameters. This representation effectively controls the complexity of the PolyCube via the number of cuboids, resulting in few corners. At the same time, it is compact and resolution-independent compared to methods based on voxelization [Yang et al. 2019; Yu et al. 2014].

Although our PolyCube generation using cuboids sidesteps the typical robustness issues in deformation-based methods [Gregson et al. 2011; Huang et al. 2014; Sokolov and Ray 2015], it comes at the cost of losing the map between the PolyCube and the near-PolyCube (and, by extension, the input mesh). Instead of trying to recover this missing link, we compute a locally injective map from a voxelized PolyCube directly to the input domain to get the final hex mesh. This is made possible by three components: a smooth distortion energy [Garanzha et al. 2021] that forces local injectivity of the map, an inversion-free pullback step that guides the hex mesh to progressively deform to the input domain, and a bi-directional proximity energy that encourages the recovery of the input surface. A mesh quality optimization step then follows to further improve the final hex mesh.

Rather than making our pipeline fully automatic, we instead create an interactive system to give the user significant freedom on how their hex meshes are generated so as to accommodate application-dependent requirements. As the resulting hex mesh largely depends on the PolyCube structure, we allow the user to build the PolyCube interactively by adding or modifying the constituent cuboids in an intuitive way, while also providing the automatic option to adjust the existing cuboids using our PolyCube optimization method. In addition to PolyCube generation, the user can substantially affect other parts of the pipeline, e.g., by digging or extruding layers on the voxelized PolyCube, modifying surface vertex positions of the final hex mesh, and exploring the trade-off among competing metrics in the refinement step.

Compared to past automatic and interactive hex-meshing methods, our system reliably generates all-hex meshes for a wide range of input domains with mesh quality on par with or surpassing that of prior work. At the same time, our system allows intuitive and extensive user control across all stages of the pipeline. We perform a user study with 21 participants; most participants are satisfied with their results over the ones from www.hexalab.net and enjoy the fact that they are able to make fine-grained adjustments.

Contributions. We present an end-to-end interactive pipeline for robust hexahedral meshing based on cuboid decomposition. Our main contributions include:

- a method for continuous optimization of PolyCube structure via cuboid decomposition and signed distance fields;
- a method for computing a low-distortion, inversion-free volumetric map from a voxelized PolyCube to the input mesh; and
- an integrated interactive system that gives the user extensive and intuitive controls over the proposed pipeline with automatic alternatives.

2 RELATED WORKS

PolyCube construction and hex meshing. Tarini et al. [2004] first suggest using PolyCube maps to create seamless texture mappings with manual PolyCube construction. Lin et al. [2008] subsequently introduce the first automatic approach to construct a PolyCube based on segmentation using Reeb graphs but with a limited set of primitives.

Gregson et al. [2011] propose a hex-meshing pipeline that deforms the input domain into a PolyCube and then pulls back a voxelized PolyCube to obtain the hex mesh. To obtain a PolyCube, they segment the surface into charts based on rotations and flatten the charts by solving a Poisson equation. Livesu et al. [2013] cast the segmentation in PolyCube generation as a multi-label graph cut problem. Huang et al. [2014] suggest using the ℓ_1-norm of surface normals to encourage cubeness during the deformation while constraining the total surface area to prevent degeneracy. Fang et al. [2016] cut tunnel loops of the input surface to allow generation of hex meshes with a much larger class of singularity patterns using PolyCubes, but their method is expensive due to consistency constraints across the cuts. Fu et al. [2016] alternate between smoothing the surface normals and deforming the surface to be axis-aligned until a valid PolyCube is generated, using the AMIPS energy [Fu et al. 2015] to enforce inversion-free deformation. More recently, Guo et al. [2020] cut PolyCube edges open to inject internal singularities connected with the boundary while preserving a set of prescribed feature curves. None of these methods guarantees the generation of a bijective map that deforms the input domain to a valid PolyCube, a challenge that our approach sidesteps, and non-exhaustive heuristic fixes are typically used. Many [Fu et al. 2016; Gregson et al. 2011; Livesu et al. 2013] also limit the type of PolyCubes to the ones where every corner is adjacent to three charts, a sufficient but not necessary condition [Eppstein and Mumford 2010], whereas our proposed pipeline has no such limitation (Fig. 19).

Yu et al. [2014] directly voxelize the near-PolyCube deformed from the input mesh, a procedure of obtaining a PolyCube with guaranteed success similar to ours. Morphological operations then simplify the voxelized PolyCube. However, they separate computation of the surface and volume components of the backward mapping, while we formulate a cohesive optimization scheme that computes both simultaneously, avoiding situations where a fixed surface mapping does not admit a low-distortion mapping of the volume. As a follow-up, Yang et al. [2019] use erasing-and-filling operators to reduce the number of corners in the voxelized PolyCube, but they are only concerned with producing a surface PolyCube map. In contrast to both voxelization-based methods, our pipeline builds the PolyCube in a top-down manner: cuboids are placed one by one to form the PolyCube whose complexity increases gradually.

Aside from robustness issues, all these fully-automatic pipelines consist of multiple stages where the correctness of each successive stage relies heavily on the success of previous stages. As the errors can accumulate unpredictably, it is strenuous to tweak the parameters to yield desirable results. In contrast, our interactive system allows the user to ensure the quality of each stage separately using intuitive controls before moving onto the next one.

Interactive hex meshing. Contrary to the extensive research in interactive quadrilateral meshing [Campen and Kobbelt 2014; Ebke et al. 2016; Jakob et al. 2015], few methods for interactive hex meshing have been proposed. Takayama [2019] introduce dual-sheet hexahedralization by asking the user to design sheets that are combinatorial duals of a hex layout followed by a primalization step that recovers the hex mesh. The representation of dual sheets as zero isosurfaces of implicit functions allows intuitive user editing and simplifies computation. Although their method can generate a large class of all-hex meshes with internal singularity patterns, the sheet configuration needs to satisfy a series of complicated conditions and requires a manual fix otherwise. Moreover, these conditions are not sufficient to obtain a valid hex topology, and generating hex meshes with uniform edge lengths is difficult in their framework.

The industry standard for hex meshing is CUBIT, an interactive tool [Quadros 2021]. CUBIT operates directly on CAD geometries and supports user-guided sweeping operations by which a quad mesh can be extruded into hexes. Usage of CUBIT requires a significant amount of training, and their technical details are unpublished.

Other hex meshing techniques. A promising line of work extends cross-field-based quadrilateral meshing [Bommes et al. 2009] to 3D by computing a smooth boundary-aligned frame field on an input domain to guide the hex layout [Huang et al. 2011; Palmer et al. 2020; Ray et al. 2016; Solomon et al. 2017]. While frame-field-based methods do not limit their singularities to the surface and can generate high-quality meshes, they frequently fail even on fairly simple domains [Viertel et al. 2016]. Corman and Crane [2019]; Liu et al. [2018] add topological constraints to frame field generation, but these methods have not managed to increase robustness of end-to-end field-based hex mesh generation.

Gao et al. [2019] use an octree to adaptively generate hex elements that fill the input domain while preserving features. However, their method is slow and cannot generate hexes of uniform size. Livesu et al. [2020] use a surface frame field to guide creation of cuts that partition the input mesh, generating hex-dominant meshes rather than all-hex meshes. We compare against these methods in Section 6.

Various postprocessing techniques [Cherchi et al. 2019; Fu et al. 2015; Gao et al. 2015; Livesu et al. 2015; Marschner et al. 2020] have been suggested to improve the quality of a hex mesh. In our pipeline, mesh refinement is incorporated seamlessly in a final stage. We further allow customization of the surface layout as well as exploration of competing quality metrics.

3 SYSTEM OVERVIEW

Our system inputs a tetrahedral mesh and, with user guidance, generates a hexahedral mesh whose surface matches that of the input. If the input is only a triangular surface mesh, we run [Hu et al. 2020] to tetrahedralize its volume as preprocessing. All input meshes are centered and rescaled to fit within a unit bounding box. Like past PolyCube-based methods, we limit the class of generated hex meshes to those with the topology of a voxelized PolyCube hex mesh; in particular, all singularities are on the surface (except for a layer of global padding). Fig. 2 illustrates our pipeline.

Our system employs several user-in-the-loop stages that give the user fine-grained control over the pipeline. After deforming the input mesh into a near-PolyCube shape whose faces are almost axis-aligned (Section 3.1, Fig. 2(b)), the system generates a PolyCube composed of a collection of cuboids whose union approximates the deformed shape in a semi-automatic fashion (Section 3.2, Fig. 2(c)). The PolyCube is then voxelized to a hex mesh, which the user can further modify (Section 3.3, Fig. 2(d)). Lastly, the voxelized PolyCube is mapped to the input mesh so that the mapped surface agrees with the input mesh surface (Section 3.4, Fig. 2(e)).

Please refer to the supplemental video for a demo of our system.

3.1 Deformation stage

In the first stage of the pipeline, an input tetrahedral mesh is deformed into a near-PolyCube shape (see the second column of Fig. 2). While the deformed shape is not a (strict) PolyCube (its face normals can deviate from the main axis directions), this deformation makes it easier to approximate the shape using cuboids in the following stage by, e.g., aligning it with the coordinate axes and reducing the numbers of corners and creases.

To achieve the goals above, a low-distortion deformation map is computed that encourages normals to be aligned with the main axis directions (Section 4.1). The user has the option to control the deformation by changing *cubeness* and *smoothness* parameters that control how PolyCube-like and smooth they want the surface to be. In addition, they can modify the parameters before resuming the deformation process. A typical use case is to start with a low cubeness value so the shape is globally axis-aligned and then gradually increase the cubeness parameter to deform the shape closer to a PolyCube with few corners (Fig. 3).

3.2 Decomposition stage

In this second stage, the user guides the creation of a PolyCube represented as a collection of axis-aligned cuboids whose union approximates the near-PolyCube from the previous stage. The quality of the generated PolyCube is determined by how closely it approximates the deformed shape and its complexity, as more complex PolyCubes (e.g., with more corners) lead to more surface singularities in the resulting hex mesh.

The user progressively builds the PolyCube by adding and modifying constituent cuboids using a combination of manual editing and automatic adjustment via our continuous PolyCube optimization (Section 4.2). This process continues until the PolyCube reaches a satisfactory level of complexity and fidelity to the near-PolyCube

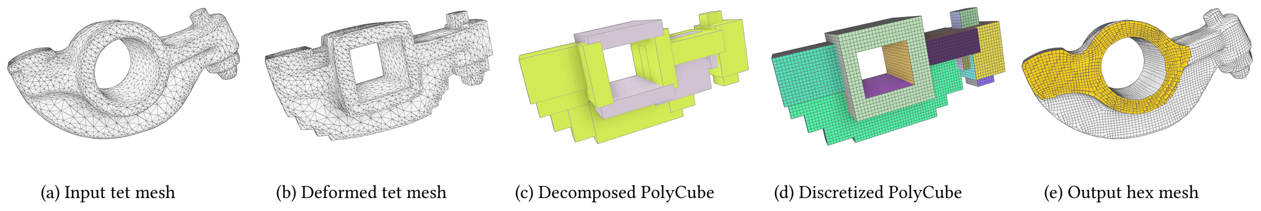


Fig. 2. Overview of our interactive hex meshing pipeline. (b)–(e) correspond to the output of the four stages of the pipeline. The output hex mesh (e) is sliced open, with yellow indicating interior quad faces for visualization.

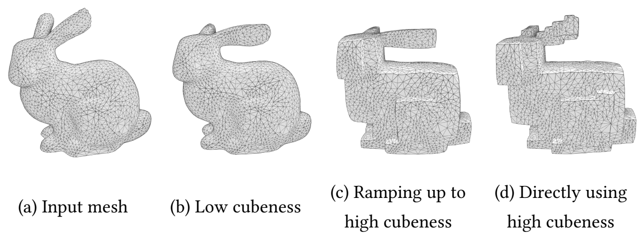

(a) Input mesh
(b) Low cubeness
(c) Ramping up to high cubeness
(d) Directly using high cubeness

Fig. 3. User-guided deformation. Starting with an input mesh (a), first using a low cubeness value (= 0.3) orients the coarse features such as the bunny ears to be axis-aligned (b). Then, increasing the cubeness value (= 3.0) deforms the shape closer to a PolyCube by creating sharp edges (c). In contrast, directly using a high cubeness value creates unnecessary stairs on the ears of the bunny (d).

shape deformed from the input. At any point, the user can perform one of the following operations (see also Fig. 4):

Add A new cuboid is placed in the scene according to one of two automatic heuristics: a distance-based heuristic places a cuboid at an uncovered point of the deformed mesh that is furthest away from any existing cuboid, while a volume-based heuristic places a cuboid of largest volume that is inside the deformed mesh and outside any existing cuboid (see Fig. 5 for comparison).

Edit The user can resize or reposition any existing cuboid using mouse-based controls. The user can also toggle *sticky mode* in which translational motion automatically snaps to align faces of the selected cuboid to that of nearby cuboids (Fig. 6(e)). Additionally, a cuboid can be removed or duplicated, and cuboid parameters can be fine-tuned through input fields.

Subtract The system can suggest a large cubic region that is over-covered by the current set of cuboids, i.e., a region that is outside the deformed mesh but contained in the union of cuboids (in Fig. 4(d)(g)). After optional user edits to the region, it is subtracted from any intersecting cuboid by splitting each cuboid into up to six non-disjoint cuboids. This is useful for recovering small topological features from over-covered regions like holes.

Reoptimize When cuboids are roughly in the right place, the user can choose to automatically optimize the parameters of all cuboids to best approximate the deformed mesh (Section 4.2). A cuboid can be *locked* to prevent it from being optimized (e.g., grey cuboids in Fig. 2(c)) if the user is satisfied with it.

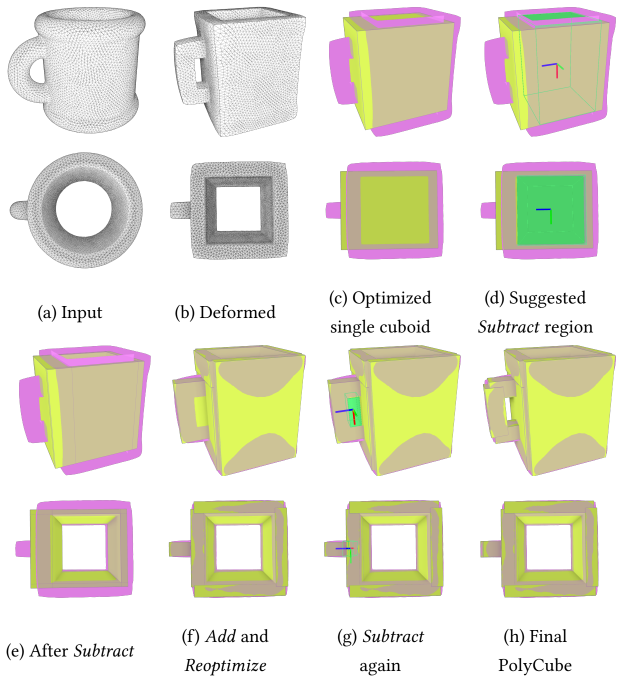

(a) Input
(b) Deformed
(c) Optimized single cuboid
(d) Suggested *Subtract* region
(e) After *Subtract*
(f) *Add* and *Reoptimize*
(g) *Subtract* again
(h) Final PolyCube

Fig. 4. An example of constructing the PolyCube in the decomposition stage using the *cup* model. A front view and a top-down view are shown for each entry. The surface of the deformed mesh is shown in translucent pink. (c) A single cuboid covers the hole of the cup after *Reoptimize*. (d) The user sees a green editable region suggested by the system indicating that this region can be subtracted out. (e) The PolyCube after performing *Subtract*. (f) The user can then proceed to add new cuboids and *Reoptimize*, which may over-cover the cup handle. (g) Another *Subtract* can be performed to recover the handle to get (h). Further *Edit* operations can be done to reduce the number of corners. Alternatively, the user can avoid using *Subtract* for this model by either directly editing the cuboids or using volume-based *Add* to put a cuboid on each wall of the deformed cup.

The implementation of each heuristic is detailed in Appendix A.1.

We found two successful strategies requiring minimal user intervention: (1) alternate between *Add* with the distance-based heuristic and *Reoptimize*, or (2) apply *Add* repeatedly with the volume-based heuristic to cover significant regions, *Reoptimize*, and then make

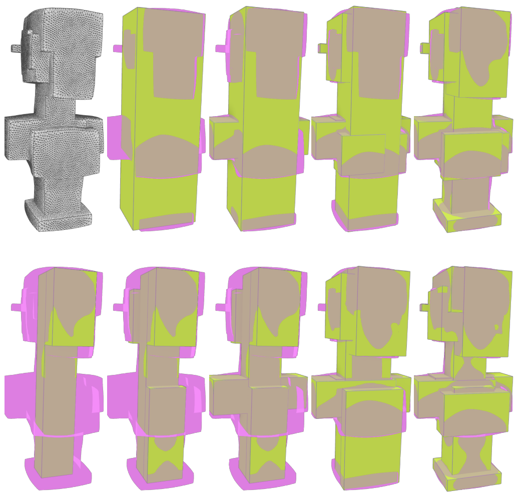


Fig. 5. Comparison of two cuboid adding heuristics. The surface of the deformed mesh is shown in translucent pink. No user editing is involved in this example. In each row, the number of PolyCubes increases from left to right. In the top row we show the deformed *buste* model followed by progressively generated PolyCubes after alternating distance-based *Add* and *Reoptimize* with 1, 3, 6, 11 cuboids, respectively. The first three columns of the bottom row are PolyCubes after performing volume-based *Add* 2, 4, 6 times. While avoiding over-coverage, volume-based *Add* cannot capture small details such as braids, so we resort back to distance-based *Add* and *Reoptimize*, as shown in the last two columns.

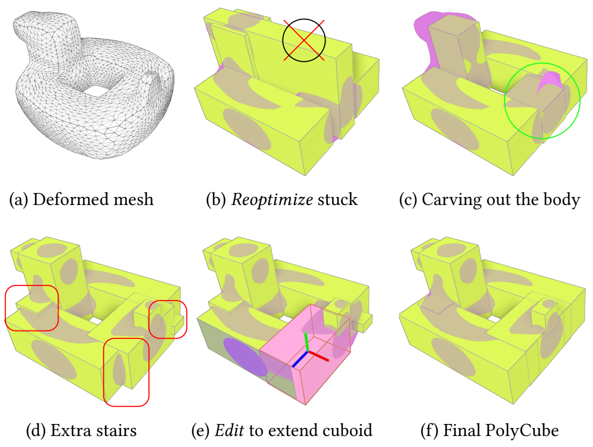


Fig. 6. An example where user interaction is helpful. From a deformed *bob* model (a), the optimization gets stuck in a local minimum (b). To resolve this, the user can remove the center cuboid (marked with red cross) and then add a new one near the tail (c). The optimization finds a configuration (d) that minimizes the approximation error but has unnecessary stairs. The user can remove the stairs by snapping the surrounding cuboids' faces together using *sticky mode* (e) to get the final PolyCube (f).

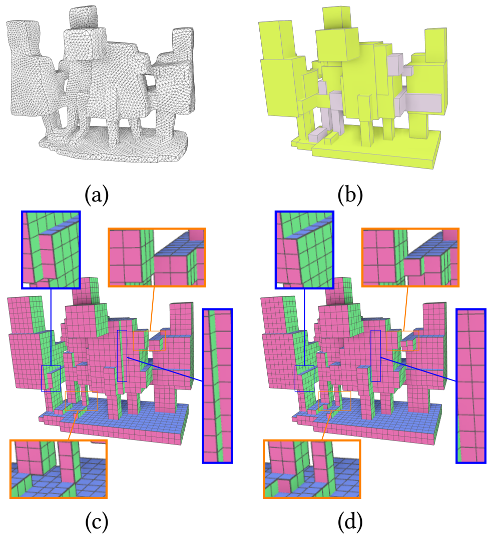


Fig. 7. Fixing topological problems and removing unnecessary stairs. (a) Deformed mesh. (b) PolyCube after the decomposition stage (grey indicates fixed cuboids). (c) Direct discretization by snapping PolyCube corners to an integer grid may result in disconnected regions (shown in orange boxes) even if the cuboids are not disconnected. There can also be unnecessary stairs (shown in blue boxes). (d) By adding and removing individual voxels or layers of voxels, the user can fix the topological problems and remove unnecessary stairs.

fine adjustment to small regions. See Fig. 5 for a comparison of these strategies.

Allowing user interaction rather than fully automating this stage offers two distinct advantages. First, for a fixed number of cuboids, the optimization from Section 4.2 can get stuck in local minima; it is often easy for the user to modify the configuration to suggest a better optimum (Fig. 6 (b)(c)). Second, there can be different configurations of cuboids that yield similar approximation error, yet one may be preferred over the others depending on the user's application. For example, it is often desirable to sacrifice a small amount of approximation error for a simpler PolyCube structure, e.g., one with fewer stairs, even for a fixed number of cuboids (Fig. 6 (d)). Thanks to our choice of representation, editing the PolyCube structure is intuitive and transparent by editing individual cuboids.

3.3 Discretization stage

In this stage, the PolyCube made in the previous stage is voxelized into a hex mesh. To accomplish this, we first snap all cuboid corners to a regular grid of a user-specified edge length. Then, the user can edit the voxelized PolyCube by adding or removing voxels, either one at at time or by editing an entire layer. Once the user is satisfied, a layer of global padding is added in the same way as in [Gregson et al. 2011].

User interaction is helpful in two ways. First, snapping PolyCube corners to integers may erase small topological features [Protais et al. 2020] that the user can easily identify and fix. Second, the user has one more opportunity to simplify the PolyCube structure, e.g., by removing unnecessary stairs. See Fig. 7 for an example.

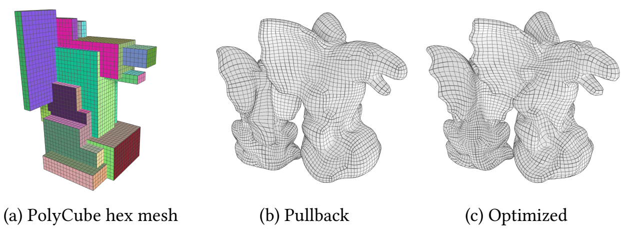


Fig. 8. Demonstration of the hexahedralization stage using the *gargoyle* model. (a) PolyCube hex mesh from the discretization stage. (b) Visualization of the front and the back of the initialized hex mesh obtained from our inversion-free pullback (scaled Jacobians: $J_{\min} = 0.014$, $J_{\text{avg}} = 0.781$; Hausdorff distance: $d_{\max} = 46.292$). (c) Optimized version of (b) with a large *details* parameter while improving worst hex element quality ($J_{\min} = 0.253$, $J_{\text{avg}} = 0.771$, $d_{\max} = 19.095$). Notice how the details around the wings and the ears are recovered during optimization.

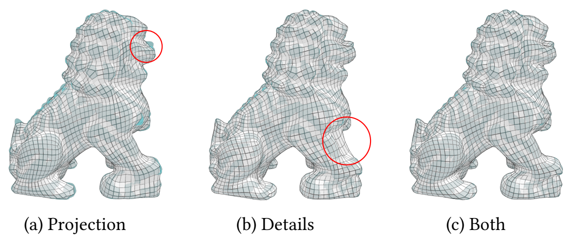


Fig. 9. Comparison of surface metrics in the hexahedralization stage on the *Chinese lion* model. (a) Setting a large projection parameter and a small details parameter makes the surface vertices of the resultant hex mesh stay on the input surface, but fine details are not recovered ($d_{\max} = 22.263$, $d_{\text{avg}} = 1.297$). (b) Finer details, such as the lion's nose, are better recovered using a large details parameter, but there is over-coverage at the front leg due to a small projection parameter ($d_{\max} = 44.952$, $d_{\text{avg}} = 0.861$). (c) When using large parameters for both projection and details, our resulting surface approximates the input surface the best ($d_{\max} = 9.342$, $d_{\text{avg}} = 1.297$).

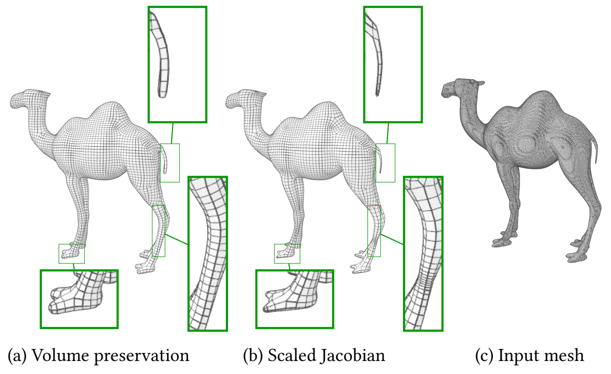


Fig. 10. Trade-off between volume preservation (*authalic* parameter) and scaled Jacobian (*custom* parameter). (a) Result when using a high *authalic* parameter for hex-meshing the *camel* model. The tail and toes are bloated compared to the input to preserve the volume of each hex element. (b) Result when using a low *authalic* parameter but with a high *custom* parameter to improve the scaled Jacobian. The shin part of the camel has more cubic hexes with varying volume, which are preferred by the scaled Jacobian. (c) Input tet mesh for comparison.

3.4 Hexahedralization stage

In the final stage, the PolyCube hex mesh from the previous stage is deformed to obtain the output hex mesh. The deformation matches the surface of the PolyCube hex mesh with the input surface while retaining hex element quality.

The system initializes the output hex mesh by pulling back the PolyCube hex mesh in an inversion-free manner (Section 4.3, Fig. 8(a)). Next, the user guides the system to optimize the hex mesh while aligning its surface to that of the input mesh. The user has control over a range of quality parameters, divided into surface and hex element quality metrics. For surface metrics, the user can control the smoothness of the surface, the closeness of the surface to the input, and the level of detail of the surface (Fig. 9). For hex element quality metrics, the user can choose how much they want the elements to be angle-preserving or volume-preserving, or they can supply an application-dependent custom metric (Fig. 10). We choose the scaled Jacobian as the custom metric, but other types can be easily incorporated (Section 4.4). The user may also choose to optimize over the worst elements or over the average. These controllable parameters are summarized in Table 1. Similar to Section 3.1, the user can change the parameters before continuing the optimization of the mesh (Fig. 8(c)).

As these metrics compete with each other, we leave it to the user to explore the trade-off landscape (e.g., Figs. 16 and 17). To guide user exploration, a hex quality metric (scaled Jacobian by default) of the current hex mesh and the Hausdorff distance to the input mesh surface are displayed in a window to help the user choose the desired trade-off. Additionally, the input mesh surface is shown as a translucent shell as a visual aid (Fig. 9).

To enable fine-grained control over the surface, we provide a tool for the user to mark and reposition surface vertices as *landmarks*, which are then fixed during the optimization. This is helpful for guiding the optimization, e.g., by preventing points from getting projected onto the wrong side of the surface (Fig. 11).

The user has the additional option of choosing how the surface vertices are parameterized during optimization using one of three modes: *free*—surface vertices are free to move off the input surface, *constrained*—surface vertices are constrained to move only along the input surface, and *fixed*—surface vertices are fixed during the optimization. See the corresponding mathematical formulation in Section 4.4. The *free* mode is useful for most situations since it can effectively alleviate or prevent foldovers, while *constrained* and *fixed* modes are good for final refinement.

For visualization, we provide the user with tools to filter away hex elements, including a slicing plane and a quality threshold, similar to Hexalab [Bracci et al. 2019]. For instance, the user can easily identify where the bad hex elements are using the quality threshold filter to fix them using landmark tools or by going back to the previous stage to make topological changes to the voxelized PolyCube.

4 OPTIMIZATION

In this section, we present our mathematical formulation of the optimization problems faced in the pipeline stages above. We use Adam

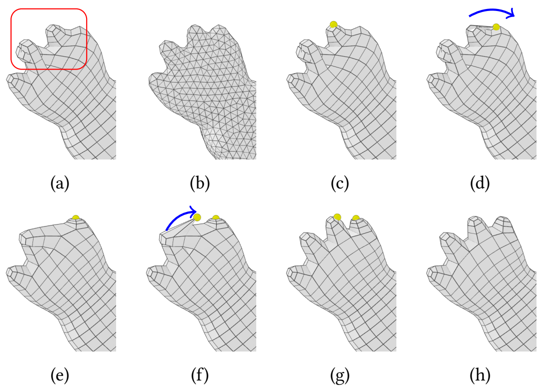


Fig. 11. Fine-grained surface control using landmarks. (a) Optimized *armadillo* model has fingers intersecting. (b) Input tet mesh for reference. It suggests we should shift the fingers in the red box the right to align them with the input. (c)-(d) We put a landmark on the rightmost finger and then move it to the right. (e) Reoptimization then moves the rightmost finger into roughly the correct position in order to reduce distortion. (f)-(g) We repeat the process with the other finger. (h) Finally we clear the landmarks and reoptimize again.

Table 1. User-controlled parameters during the hexahedralization stage. The symbol column corresponds to the mathematical notation used in the energy terms from Section 4.4.

parameter	description	symbol
smoothness	smoothness of result surface	λ_{lap}
projection	closeness of result surface to input	$\lambda_{m\to 0}$
details	level of recovered surface details	$\lambda_{0\to m}$
conformal	angle-preserving level	λ_{angle}
authalic	volume-preserving level	λ_{vol}
custom	custom mesh quality metric	λ_{custom}

[Kingma and Ba 2014] as the optimizer for all problems described in this section to achieve interactive speed.

Notation. We denote a tetrahedral mesh as $\mathcal{M} = (V, T)$ where V and T are the sets of vertices (represented as elements in $\mathbb{R}^3$) and tetrahedra (represented as 4-tuples of indices), respectively. We use $\partial\mathcal{M}$ to denote the boundary triangular mesh induced by $\mathcal{M}$ as $\partial\mathcal{M} := (\partial V, F)$ where ∂V and F are the surface vertices restricted from V and surface faces, respectively. We use similar notations for hexahedral meshes. For instance, $\mathcal{M} = (V, H)$ represents a hex mesh with a vertex set V and a hex set H, and $\partial\mathcal{M} := (\partial V, Q)$ denotes its surface quadrilateral mesh with vertices ∂V and quads Q.

We use $\text{vol}(V, t)$ to denote the volume of tetrahedron $t \in T$ computed using the corresponding vertices in V, and we write $\text{vol}(\mathcal{M}) := \sum_{t\in T} \text{vol}(V, t)$. Similarly, we define $\text{area}(V, f)$ to be the area of the face $f \in F$ and $\text{area}(\partial\mathcal{M}) := \sum_{f\in F} \text{area}(V, f)$. We call vectors $(\pm 1, 0, 0)$, $(0, \pm 1, 0)$, $(0, 0, \pm 1)$ the *major axis directions*.

An $\mathbb{R}^3$-valued piecewise-linear map f on a tetrahedral mesh $\mathcal{M} = (V, T)$ is determined uniquely by $f(V)$, its values on the vertices V. Hence, we parameterize such maps using the mapped vertex positions, and, with slight abuse of notation, we use f to denote both the map on the mesh domain and the discrete map on vertices. We denote the (constant) Jacobian of f in a tetrahedron $t \in T$ as $J_t(f)$, which is a linear function of $f(V)|_t$, the 4 mapped vertex positions of t.

4.1 Deformation to a near-PolyCube

In the deformation stage (Section 3.1), the input shape is deformed into a near-PolyCube shape to guide the placement of cuboids in the decomposition stage (Section 3.2). The deformation map is later used in the hexahedralization stage (Section 3.4) to initialize the PolyCube hex mesh. Hence, we want the deformation map to be of low distortion and inversion-free while encouraging axis-alignment.

Let $\mathcal{M}_0 = (V_0, T_0)$ denote the input tetrahedral mesh. We look for a low-distortion piecewise-linear deformation $f_d : V_0 \longrightarrow \mathbb{R}^3$ that maps the input mesh $\mathcal{M}_0$ to the PolyCube-like deformed shape $\mathcal{M}_d := (V_d, T_d)$, with $V_d := f_d(V_0)$ and $T_d := T_0$. For $t \in T_0$, let $J_t := J_t(f_d)$ denote the Jacobian of f_d restricted to the tetrahedron t. To obtain f_d, we minimize a deformation energy defined as $E := E_{\text{iso}} + E_{\text{align}}$, where E_{iso} is the distortion energy defined in Eq. (1) and E_{align} is the axis-alignment energy defined in Eq. (3). We take the identity map as the initial f_d.

Distortion energy. We want a smooth distortion energy that encourages angle and volume preservation of the deformation map. As our initial f_d is inversion-free, we also want our distortion energy to blow up when inversion happens. We choose the regularized distortion energy introduced by Garanzha et al. [2021]:

$$E_{\text{iso}} := \sum_{t\in T_0} \frac{\text{vol}(V_0, t)}{\text{vol}(\mathcal{M})} \left(\lambda_{\text{angle}} \frac{\operatorname{tr} J_t^\top J_t}{(R_\epsilon(\det J_t))^{2/3}} + \lambda_{\text{vol}} \frac{\det^2 J_t + 1}{R_\epsilon(\det J_t)} \right), \tag{1}$$

where $\lambda_{\text{angle}}, \lambda_{\text{vol}}$ are constants and $R_\epsilon : \mathbb{R} \longrightarrow \mathbb{R}_{>0}$ is a regularizer that forces the energy to blow up when $\det J_t$ is close to zero or negative:

$$R_\epsilon(x) := \frac{x + \sqrt{x^2 + \epsilon^2}}{2}. \tag{2}$$

The first term in the summand of Eq. (1) favors angle-preserving maps, while the second term favors volume-preserving maps. We find $\lambda_{\text{angle}} = \lambda_{\text{vol}} = 1.0$ and $\epsilon = 10^{-3}$ sufficient in most cases, but we let the user change them if needed.

Alignment energy. We add an alignment energy for the surface vertices ∂V_d to encourage deformation into an axis-aligned near-PolyCube. This energy has two terms, one favoring cubeness and the other favoring a smooth transition of normals on nearby faces, similar to the one used by Fang et al. [2016]:

$$E_{\text{align}} := \lambda_{\text{cube}} E_{\text{cube}} + \lambda_{\text{smooth}} E_{\text{smooth}}, \tag{3}$$

where

$$E_{\text{cube}} := \sum_{f\in\partial F_0} \frac{\text{area}(V_0, f)}{\text{area}(\partial\mathcal{M}_0)} \Phi(\hat{n}_f), \tag{4}$$

$$E_{\text{smooth}} := \sum_{\substack{f_i, f_j \in \partial F_0 \\ f_i, f_j \text{ adjacent}}} \frac{\text{area}(V_0, f_i) + \text{area}(V_0, f_j)}{3\text{area}(\partial\mathcal{M}_0)} \left\| \hat{n}_{f_i} - \hat{n}_{f_j} \right\|_2^2. \tag{5}$$

Here we use $\hat{n}_f$ to denote the unit normal obtained by normalizing $(v_1 - v_0) \times (v_2 - v_0)$, where v_0, v_1, v_2 are deformed vertex positions of the face f. We choose a smooth cubeness function

$$\Phi(n) = n_x^2 n_y^2 + n_y^2 n_z^2 + n_z^2 n_x^2$$

from Fu et al. [2016] to penalize deviation of normals from the major axis directions. We found this energy better at globally orienting the shape (see Fig. 3 (b)) and more stable during optimization compared to the ℓ_1-norm from Huang et al. [2014]. To prevent collapsing to a point, we use the original vertex positions to calculate the area weights.

4.2 Continuous PolyCube optimization

In the decomposition stage (Section 3.2), the user guides the construction of a PolyCube using cuboids whose union approximates the deformed near-PolyCube shape $\mathcal{M}_d$. Such construction is facilitated by the a continuous PolyCube optimization scheme described below that automatically adjusts existing cuboids' parameters.

Our formulation is inspired by the distance-field-based approach by Smirnov et al. [2020], where the discrepancy between two shapes is measured by comparing their distance fields. For an arbitrary set $S \subset \mathbb{R}^3$, the *signed distance field* of S is defined to be

$$d_S(x) := (-1)^{\mathbb{1}_{x \in S}} \inf_{y \in S} \|x - y\|_2.$$

In particular, $d_S(x) \geq 0$ if $x \notin S$, and $d_S(x) \leq 0$ if $x \in S$. For an axis-aligned cuboid C with center $c \in \mathbb{R}^3$ and side lengths $h \in \mathbb{R}^3$, its signed distance field is given by

$$d_C(p) = \|\max(d, 0)\|_2 + \min(\max(d_x, d_y, d_z), 0),$$

where $d = (|p_x - c_x|, |p_y - c_y|, |p_z - c_z|) - h$ [Smirnov et al. 2020].

With slight abuse of notation, we use $\mathcal{M}_d$ to denote the deformed shape as a subset of $\mathbb{R}^3$, so that $d_{\mathcal{M}_d}$ denotes its signed distance field. Let $\mathcal{P} := \bigcup_{i=1}^k C_i$ denote a PolyCube consisting of k cuboids $C_1, \ldots, C_k$. Defining the signed distance field of $\mathcal{P}$ in terms of the signed distance fields of C_i's is difficult. In the case $x \notin \mathcal{P}$, however, $d_{\mathcal{P}}$ is given by a straightforward expression:

$$d_{\mathcal{P}}(x) = \min_{1 \leq i \leq k} d_{C_i}(x)\ \forall x \notin \mathcal{P}.$$

Define $\widetilde{d_{\mathcal{P}}}(x) := \min_{i=1}^k d_{C_i}(x)$. We have $\widetilde{d_{\mathcal{P}}}(x) \leq d_{\mathcal{P}}(x)$, but the gap can be arbitrary large (Fig. 12(a)). Even though $\widetilde{d_{\mathcal{P}}} \neq d_{\mathcal{P}}$, we have $\{x : \widetilde{d_{\mathcal{P}}}(x) \leq 0\} = \{x : d_{\mathcal{P}}(x) \leq 0\}$, so they represent the same interior shape.

Let $A \subset \mathbb{R}^3$ be a finite set of points that we call *anchors* on which we compare $d_{\mathcal{M}_d}$ and $d_{\mathcal{P}}$. We design our energy to be a combination of two terms: $E := \lambda_+ E_+ + \lambda_- E_-$, where E_+ and E_- are defined by Eq. (6) and Eq. (7) respectively.

Discrepancy energy E_+. For $p \in \mathbb{R}^3 \setminus \mathcal{M}_d$, we want cuboids to avoid covering p, i.e., $d_{C_i}(p) \geq 0$ for every i. But if this is the case, then $\widetilde{d_{\mathcal{P}}}(p) = d_{\mathcal{P}}(p)$, so we can compare $d_{\mathcal{P}}(p)$ with $d_{\mathcal{M}_d}(p)$ by using $\widetilde{d_{\mathcal{P}}}(p)$ in place of $d_{\mathcal{P}}(p)$. Thus we define

$$E_+ := \sum_{p \in A \setminus \mathcal{M}_d} \left(d_{\mathcal{M}_d}(p) - \widetilde{d_{\mathcal{P}}}(p) \right)^2. \tag{6}$$

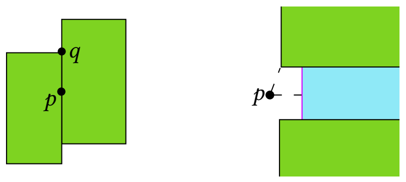
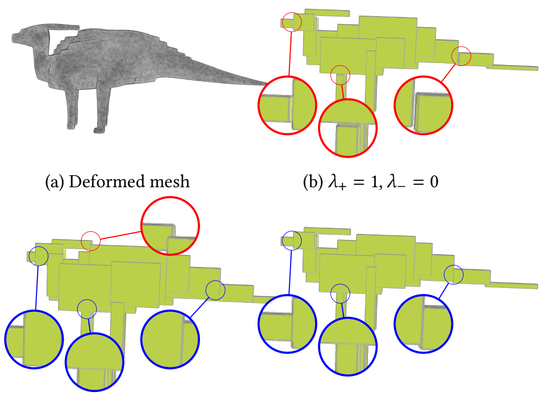


Fig. 12. (a) An illustration showing that the gap between $\widetilde{d_{\mathcal{P}}}$ and $d_{\mathcal{P}}$ can be arbitrarily large. Consider the PolyCube comprised of the two green cuboids. For the point p, $\widetilde{d_{\mathcal{P}}}(p) = 0$, but $d_{\mathcal{P}}(p) = -\|q - p\| < 0$. (b) An example showing that the discrepancy energy E_+ alone may create gaps. In this scenario, the light blue indicates $\mathcal{M}_d$, and the PolyCube consists of two green cuboids. The distance from p to $\mathcal{M}_d$ and to $\mathcal{P}$ is the same, so $(d_{\mathcal{M}_d}(p) - \widetilde{d_{\mathcal{P}}}(p))^2 = 0$, and p could be a local optimum of E_+ (Eq. (6)). This explains why E_- is needed to close the gaps.



(a) Deformed mesh
(b) $\lambda_+ = 1, \lambda_- = 0$
(c) $\lambda_+ = 0, \lambda_- = 1$
(d) $\lambda_+ = 0.1, \lambda_- = 1$

Fig. 13. Comparison of PolyCube optimization parameters λ_+, λ_-. (a) Deformed mesh of *dilo*. (b) If we use only discrepancy energy (Eq. (6)), then small gaps are created in the circled regions. (c) If we use only gap-closing energy (Eq. (6)), then gaps from (b) are not be created, but the PolyCube is too coarse and has undesired intersections (red circle). (d) With the right parameters, we can close the gaps without making the PolyCube too coarse.

Gap-closing energy E_-. For $p \in \mathcal{M}_d$, we want p to be contained in at least one cuboid, i.e., $\widetilde{d_{\mathcal{P}}}(p) \leq 0$. Therefore, we let

$$E_- := \sum_{p \in A \cap \mathcal{M}_d} \mathbb{1}_{\widetilde{d_{\mathcal{P}}}(p) \geq 0} \left(\widetilde{d_{\mathcal{P}}}(p) \right)^2 \tag{7}$$

to help close the gaps caused by using only E_+ (Fig. 12(b)).

In our experiments, we choose anchors to be a combination of a uniform grid and perturbed points from the surface of $\mathcal{M}_d$, though the user can modify the anchors if desired. In practice, we found that it is good to start with $\lambda_+ = \lambda_- = 1.0$ to prevent over-coverage in the beginning and reduce λ_+ while increasing λ_- before reoptimizing to close the gaps. Hence we let the user adjust the parameters λ_+, λ_- (see Fig. 13). An alternative to fix the gaps is through manually editing and locking the cuboids (Section 3.2).

4.3 Inversion-free pullback of PolyCube

In the beginning of the hexahedralization stage (Section 3.4), we want to find a volumetric map that deforms the PolyCube hex mesh from the discretization stage (Section 3.3) back into the input mesh geometry. We start with a two-step initialization to an inversion-free map from the PolyCube hex mesh to the input volume. The idea is to use the deformation map from the deformation stage as a guide to pull back the PolyCube hex mesh gradually, while incorporating a barrier function to prevent foldovers.

In the first step, we deform the PolyCube mesh into the near-PolyCube mesh from the deformation stage to account for the discrepancy introduced in discretization (Fig. 14(d)). In the second step, we deform back to the input mesh using barycentric pullback as a guide (Fig. 14(f)). We do not change the mesh connectivity throughout.

Let $\mathcal{M}_p := (V_p, H_p)$ denote the hexahedral mesh of the voxelized PolyCube from the discretization stage, whose hex elements are regular cubes with the same length on all sides.

4.3.1 Deforming to $\mathcal{M}_d$. In the first step, we look for a map $f_{d'} : V_p \to \mathbb{R}^3$ that sends $\mathcal{M}_p$ to $\mathcal{M}_{d'} := (V_{d'}, H_p)$ so that $\partial\mathcal{M}_{d'}$ is close to $\partial\mathcal{M}_d$. Let T_p denote the collection of tetrahedra formed by putting one tetrahedron on each of the 8 corners of every hex in H_p. For $t \in T_p$, like in Section 4.1, we use $J_t := J_t(f_{d'})$ to denote the Jacobian of $f_{d'}$ on tetrahedron t. Note that there are cases where $\det J_t > 0$ for all 8 tetrahedra in a hex but the induced trilinear mapping is not locally injective [Zhang 2005]. We have not observed such cases in our experiments, but, if necessary, we can augment T_p to include the 32 tetrahedra defined in [Zhang 2005] for each hex to make the induced trilinear map inversion-free everywhere.

We initialize $f_{d'}$ to be the identity map and optimize $f_{d'}$ by minimizing $E := E_{\text{hex-iso}} + E_{\text{prox}} + E_{\text{lap}}$, where each energy term is described below.

Distortion energy. Since we want to prevent foldovers, we use a barrier-like distortion energy similar to Eq. (1) but with uniform weights, due to the fact that volumes of all hexes in H_p (and by extension all tets in T_p) are the same:

$$E_{\text{hex-iso}} := \sum_{t \in T_p} \left(\lambda_{\text{angle}} \frac{\operatorname{tr} J_t^\top J_t}{(R_\epsilon(\det J_t))^{2/3}} + \lambda_{\text{vol}} \frac{\det^2 J_t + 1}{R_\epsilon(\det J_t)} \right). \tag{8}$$

Proximity energy. We introduce a term that measures the bi-directional distance between $\partial\mathcal{M}_{d'}$ and $\partial\mathcal{M}_d$:

$$E_{\text{prox}} := \lambda_{d' \to d} E_{d' \to d} + \lambda_{d \to d'} E_{d \to d'}, \tag{9}$$

with

$$E_{d' \to d} := \sum_{v \in \partial V_{d'}} \| v - \operatorname{proj}(v, \partial\mathcal{M}_d) \|_2^2, \tag{10}$$

$$E_{d \to d'} := \int_{\partial\mathcal{M}_d} \| v - \operatorname{proj}(v, \partial\mathcal{M}_{d'}) \|_2^2 \mathrm{d}v, \tag{11}$$

where $\operatorname{proj}(v, \partial\mathcal{M})$ denotes the projected point of v onto the surface $\partial\mathcal{M}$. We choose uniform weights in Eq. (10) because the quad surface before and after applying $f_{d'}$ should consist of quads with similar areas. To evaluate Eq. (11) during each gradient step, we use a uniformly sampled batch of $|\partial V_{d'}|$ points on $\partial\mathcal{M}_d$ before computing the gradient.

Smoothness energy. We finally add a Laplacian-smoothing energy that helps maintain a smooth surface during the deformation:

$$E_{\text{lap}} := \lambda_{\text{lap}} \sum_{v \in \partial V_{d'}} \left\| v - \frac{1}{|N(v)|} \sum_{u \in N(v)} u \right\|_2^2, \tag{12}$$

where $N(v)$ denotes the 1-ring neighborhood of v on $\partial V_{d'}$.

In this step, we set all weights λ_* to be 1 and $\epsilon = 10^{-4}$. Fig. 14(d) shows an example of this step.

4.3.2 Deforming back to $\mathcal{M}_0$. Now we have a hex mesh $\mathcal{M}_{d'} = (V_{d'}, H_p)$ that is close to the near-PolyCube mesh $\mathcal{M}_d$. In this step, we look for a map $f_m : V_p \to \mathbb{R}^3$ that extends $f_{d'}$ from Section 4.3.1 to obtain $\mathcal{M}_m = (V_m, H_p)$ with $V_m := f_m(V_p)$, so that its surface $\partial\mathcal{M}_m$ approximates the input surface $\partial\mathcal{M}_0$. For $v \in \mathbb{R}^3$, let $\operatorname{pull}(v) \in \mathcal{M}_0$ denote the result of projecting v to the closest tetrahedron in $\mathcal{M}_d$ and then pull back to $\mathcal{M}_0$ via $f_d : \mathcal{M}_0 \to \mathcal{M}_d$ from Section 4.1 using barycentric coordinates. If $\mathcal{M}_{d'}$ is exactly $\mathcal{M}_d$, then $(\operatorname{pull}(V_{d'}), H_p)$ gives an inversion-free hex mesh of the input mesh. However, this is often not the case, and naïvely using projection and then pulling back can result in foldovers or points being projected onto the wrong side of the surface (Fig. 14 (e)).

Instead, we use $\operatorname{pull}(V_{d'})$ only as a guide and deform $\mathcal{M}_{d'}$ to gradually reduce the distance between V_m and $\operatorname{pull}(V_{d'})$ while avoiding inversion. We find f_m by initializing it to be $f_{d'}$ and then minimizing $E := E_{\text{hex-iso}} + E_{\text{pullback}} + E_{\text{lap}}$, where $E_{\text{hex-iso}}$ and E_{lap} are the same as Eq. (8) and Eq. (12) respectively, except the variables are now V_m. The new energy term E_{pullback} is defined as

$$E_{\text{pullback}} := \lambda_{\text{pullback}} \sum_{v \in V_m} \| v - \operatorname{pull}(v_{d'}) \|_2^2, \tag{13}$$

where $v_{d'}$ is position of the vertex in $V_{d'}$ with the same index as v. We find $\lambda_{\text{pullback}} = 1$ sufficient. The result of this step is shown in Fig. 14(f).

4.4 Constrained mesh quality optimization

After initializing an inversion-free hex mesh that approximates the input domain (Section 4.3), the user can further improve the mesh quality in various ways.

To achieve this, we take f_m from Section 4.3.2 and further optimize it. In contrast to past hex-mesh quality improvement work [Fu et al. 2015; Livesu et al. 2015], where it is typically assumed that the surface vertices are either fixed or do not move around substantially, in our case the surface vertices ∂V_m can move significantly during optimization to support more aggressive improvement strategies (e.g., Fig. 11). To constrain the surface vertices ∂V_m to move along the input surface $\partial\mathcal{M}_0$, we use the bi-directional proximity energy E_{prox} from Eq. (9) but with $\partial\mathcal{M}_0$ in place of $\partial\mathcal{M}_d$ and with variables V_m instead of $V_{d'}$:

$$E_{\text{prox}} := \lambda_{m \to 0} E_{m \to 0} + \lambda_{0 \to m} E_{0 \to m}, \tag{14}$$

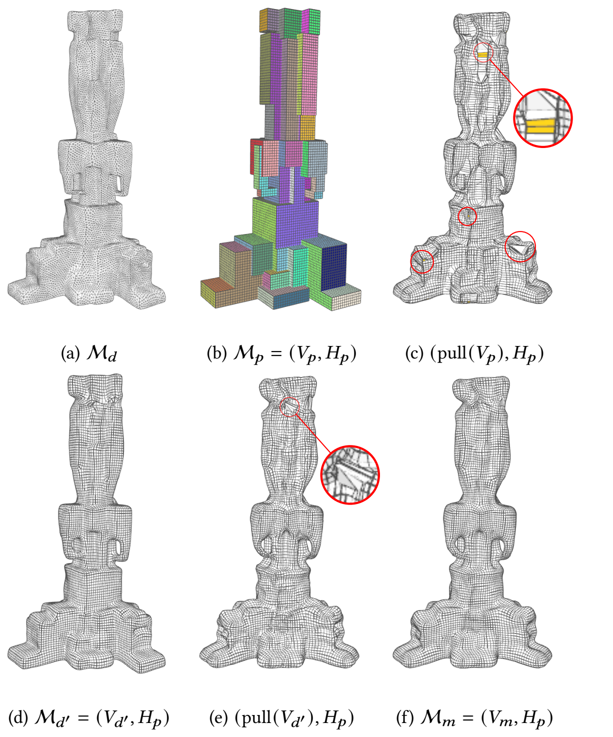

(a) $\mathcal{M}_d$
(b) $\mathcal{M}_p = (V_p, H_p)$
(c) $(\mathrm{pull}(V_p), H_p)$
(d) $\mathcal{M}_{d'} = (V_{d'}, H_p)$
(e) $(\mathrm{pull}(V_{d'}), H_p)$
(f) $\mathcal{M}_m = (V_m, H_p)$

Fig. 14. Comparison of pullback strategies. (a) Deformed mesh of *Thai statue*. (b) Generated PolyCube with distinctly colored charts. (c) Directly projecting and pulling back results in many inverted hexes as well as large distortion. (d) Result of the first step of our inversion-free pullback (Section 4.3.1). (e) If we project and pull back directly after first step, we can still obtain inverted hexes ($J_{\min} = -0.883$). (f) Result of the second step of our inversion-free pullback (Section 4.3.2) with no inverted hexes ($J_{\min} = 0.048$ before optimize mesh quality using Section 4.4).

where

$$E_{m\to 0} := \sum_{v\in\partial V_m} \|v - \mathrm{proj}(v, \partial\mathcal{M}_0)\|_2^2, \tag{15}$$

$$E_{0\to m} := \int_{\partial\mathcal{M}_0} \|v - \mathrm{proj}(v, \partial\mathcal{M}_m)\|_2^2 \mathrm{d}v. \tag{16}$$

We further include the distortion energy $E_{\text{hex-iso}}$ from Eq. (8) and the smoothness energy E_{lap} from Eq. (12), similar to Section 4.3.2.

We also allow the user to add in a custom energy E_{custom} that favors user-specific mesh quality. For instance, if the user wants the resulting mesh to have a high scaled Jacobian (Fig. 10(b)), then we set

$$E_{\text{custom}} := -\lambda_{\text{custom}} \sum_{t\in T_p} \det \widehat{J_t}, \tag{17}$$

where $\widehat{J_t}$ is obtained from J_t by normalizing each column. Any smooth metric that can be computed using vertex positions can be accommodated this way.

We let the user control the weights $\lambda_{\text{lap}}, \lambda_{m\to 0}, \lambda_{0\to m}, \lambda_{\text{angle}}, \lambda_{\text{vol}}, \lambda_{\text{custom}}$, which correspond to the interpretable parameters *smoothness*, *projection*, *details*, *conformal*, *authalic*, *custom*, respectively from Table 1.

Our final energy is $E := E_{\text{hex-iso}} + E_{\text{prox}} + E_{\text{lap}} + E_{\text{custom}}$. One option we provide is to allow the user to optimize for the worst hex element quality instead of the average one. This is inspired by the AMIPS energy from Fu et al. [2015], but instead of directly exponentiating the summands, we use the following log-sum-exp modification of Eq. (8) to improve numerical stability:

$$E_{\text{hex-lse}} := \log\left(\sum_{t\in T_p} \exp\left(\frac{\lambda_{\text{angle}} \operatorname{tr} J_t^\top J_t}{(R_\epsilon^{\det}(\det J_t))^{2/3}} + \frac{\lambda_{\text{vol}}(\det^2 J_t + 1)}{R_\epsilon^{\det}(\det J_t)}\right)\right). \tag{18}$$

We find that when $\mathcal{M}_m$ has no inverted hex, using $E_{\text{hex-lse}}$ in place of $E_{\text{hex-iso}}$ can effectively improve the worst hex element. We provide similar options for the custom loss, for instance, if the user wants to improve the minimum scaled Jacobian (Eq. (17)).

The three modes of parameterizing surface vertices ∂V_m described in Section 3.4 correspond to the following:

Free. This is the default option where surface vertices are free to move in $\mathbb{R}^3$ while relying on E_{prox} (Eq. (14)) to make them stay close to $\partial\mathcal{M}_0$.

Constrained. In this mode, we force ∂V_m to be on $\partial\mathcal{M}_0$ during the optimization. For each $v \in \partial V_m$, we create a latent variable $z \in \mathbb{R}^3$, so that $v := \mathrm{proj}(z, \partial\mathcal{M}_0)$. Let Z denote the set of all latent variables. Then, we use $\mathrm{proj}(z, \partial\mathcal{M}_0) \in \mathcal{M}_0$ in place of vertex v when calculating the total energy, and we update z by differentiating through the projection operator during each gradient step. A caveat is that there are regions where $\partial\mathrm{proj}(z, \partial\mathcal{M}_0)/\partial z$ vanishes, such as when z is above a ridge formed by two neighboring faces. To prevent gradient-based optimization from getting stuck in these situations, we use instead the constant non-zero gradient of $\partial\mathrm{proj}(z, \partial\mathcal{M}_0)/\partial z$ as if the closest triangle extends to a plane. We also modify Eq. (15) to

$$\sum_{z\in Z} \|z - \mathrm{proj}(z, \partial\mathcal{M}_0)\|_2^2, \tag{19}$$

so that the wandering latent variables will stay close to the input surface. Comparing to alternatives like projected gradient descent, this parameterization makes it seamless to use momentum-based optimizers (Section 5).

Fixed. In this mode the positions of ∂V_m are fixed during the optimization. This is useful when the user is happy with the surface but wants to further improve the interior mesh quality.

5 IMPLEMENTATION DETAILS

We implement our interactive system in C++ with Vulkan and GUI library ImGui.[1] Most of the optimization is powered by the C++ frontend of PyTorch [Paszke et al. 2019] and runs on the GPU for interactive speed. We utilize CPU-level parallelism to accelerate intensive computations like ray tracing for in-scene mouse control and hex element filtering. We use Adam [Kingma and Ba 2014] as our optimizer with a default learning rate of 10^{-3} in the deformation stage and 10^{-4} in the hexahedralization stage, and $\beta_1 = \beta_2 = 0.9$, although the user can change these parameters as well as the number of gradient steps if needed. We implement custom CUDA functions

[1] https://github.com/ocornut/imgui

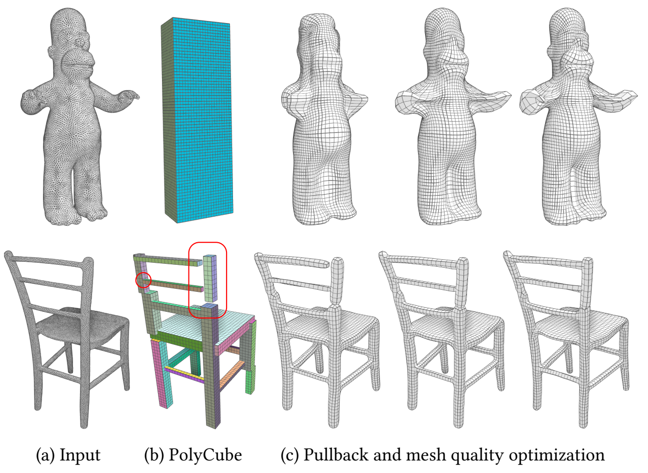


Fig. 15. Robustness of our inversion-free pullback and mesh quality optimization. The three columns in (c) show the result after the pullback followed by two more steps of mesh quality optimization. For the *homer* model (top row), we intentionally use an extremely crude PolyCube consisting of a single cuboid. The hexahedralization stage is capable of recovering a hex mesh that is similar to the input (except it fails to recover the highly non-convex gap between the legs). For the *chair* model (bottom row), the PolyCube has isolated cuboids that are disconnected from the rest (circled red). As a result, the generated hex mesh also has isolated components and may result in intersection. Note this is not the intended usage of the pipeline—the user should fix the connectivity during either the decomposition stage or the discretization stage. For both models, no inverted hex occurs throughout the process.

to enable fast signed distance fields computation and point-to-mesh projection with back-propagation support to make optimization of terms like Eq. (14) possible on meshes with about 10^5 hexes. See Appendix A.2 for more details.

For rendering, we use multi-sampled anti-aliasing and screen-space ambient occlusion in a deferred pipeline to strike a balance between visual clarity and efficiency, as we are also heavily utilizing the GPU for optimization. All figures in the paper are generated directly by our rendering pipeline.

Our system can run comfortably on mid-range NVIDIA graphics cards, such as the GTX 1080 and RTX 2060. The implementation of our system is available at https://github.com/lingxiaoli94/interactive-hex-meshing.

6 RESULTS

Inversion-free pullback stress test. We test the robustness of our hexahedralization stage (Section 3.4) when the voxelized PolyCube from the previous stage is too coarse or has topological deficiencies (Fig. 15). In both cases, our method is capable of pulling back the voxelized PolyCube in an inversion-free manner.

Trade-off exploration. Our mesh quality optimization component of the hexahedralization stage allows the user to explore and choose their preferred trade-off. In Fig. 16, we demonstrate the exploration of the trade-off between angle preservation and scaled Jacobian values by simply changing the relative weights before optimizing. In Fig. 17, we show the exploration of the trade-off between hex element quality and approximation level of the input surface. Since trade-off exploration is incorporated seamlessly within the system, the user can easily try out any combination of the parameters from Table 1, change the surface parameterization mode, or put landmarks to alter surface vertex positions manually (Fig. 11) before reoptimization.

Challenging models. Existing deformation-based PolyCube hex meshing methods rely heavily on the deformation and may fail when correct stairs cannot be created [Sokolov and Ray 2015]. In comparison, the continuous PolyCube optimization in our method always results in a valid PolyCube and resolves stairs correctly most of the time, while allowing user intervention if needed. We test our method on some challenging examples from Sokolov and Ray [2015], and our pipeline reliably generates good hex meshes with limited user interaction (Fig. 18).

Larger class of PolyCubes. Compared to previous PolyCube-based methods [Fu et al. 2016; Livesu et al. 2013] where the heuristic repairs limit the PolyCube structure to have exactly three adjacent charts on every corner, our pipeline allows PolyCubes to have corners adjacent to more than three charts (Fig. 19).

Comparison. We test our pipeline on a number of models and quantitatively compare the obtained results with recent hex meshing methods [Fang et al. 2016; Fu et al. 2016; Guo et al. 2020; Livesu et al. 2020; Takayama 2019; Yu et al. 2014]. The statistics and timings are shown in Table 2. For all models tested except for *joint* and *sculpt*, we obtain better minimum scaled Jacobian while having competing numbers in other metrics. The reported runtime for our method is the recorded total time for an expert user for each model. The detailed timings for each stage vary across different models; empirically the decomposition and discretization stages take the most time.

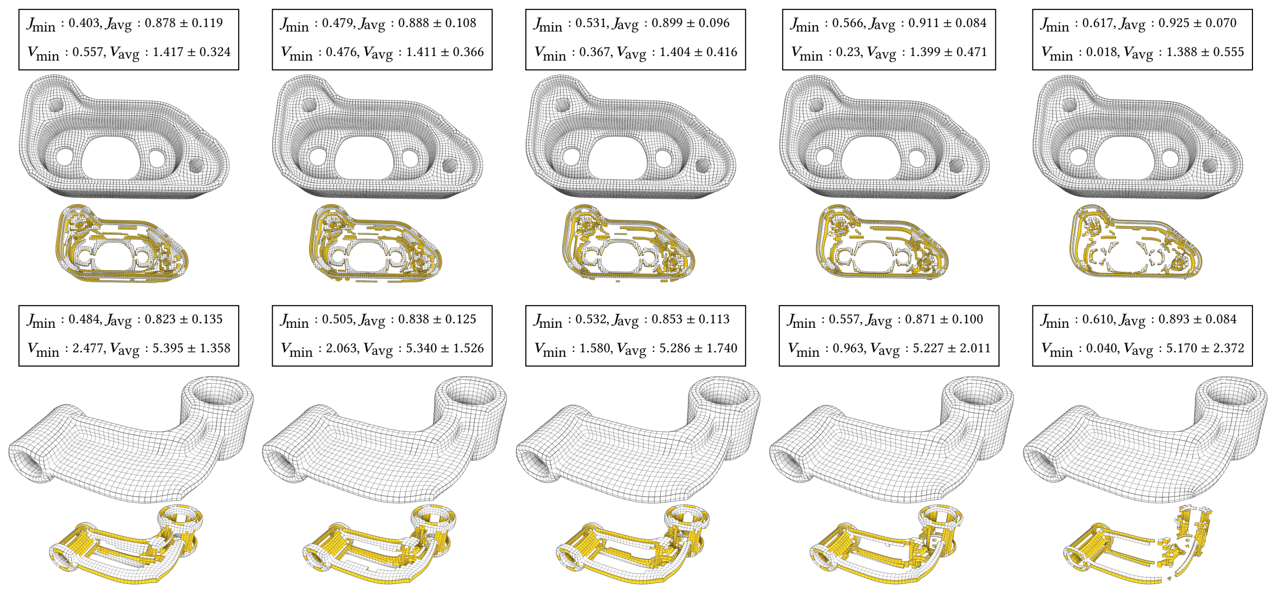
User Study. Our system offers an interactive user-in-the-loop experience for producing hex meshes. Unlike previous algorithms, a key advantage of our method is the ability for the user to make an intuitive choice at each juncture, all of which contribute to the output. To evaluate this aspect of our work, we conduct a user study. Each participant accessed a build of our software preinstalled on a remote server. They were first asked to complete a detailed tutorial, which outlined the process of obtaining a hex mesh using our tool (taking the *spot* model as an example). Users were also provided with a video walk-through of the steps described in the tutorial.

Upon finishing the tutorial, users were asked to select one or more additional meshes (from the publicly available models of Fu et al. [2016]) to experiment with and produce a hex mesh using our software. Finally, they completed a survey about their prior experience with 3D modeling, their overall experience with our tool, and their experience with each mesh that they chose.

A total of 21 participants took part in the user study, with a self-reported average score of 2.2 for familiarity with 3D modeling tools and 2.0 familiarity with hex meshing (both on a 1-5 scale). The users spent an average of 20.1 minutes on each mesh that they worked with after the tutorial. On average, 81% of the users were satisfied with the hex meshes that they obtained. In particular, 18 out of the 21

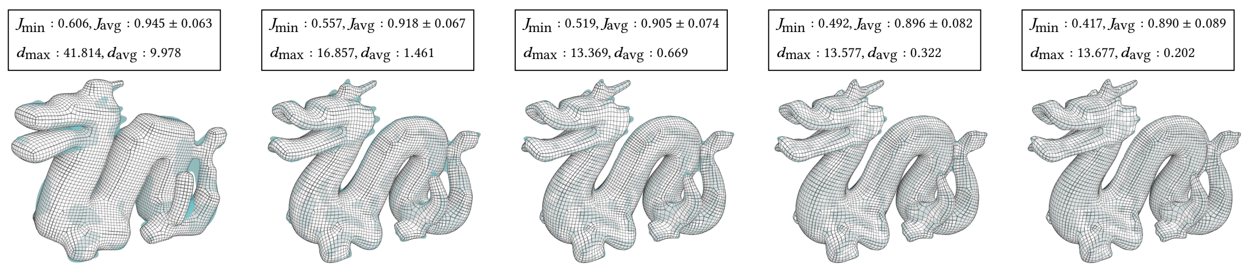
Table 2. Comparison of hex mesh quality between our results and other recent methods. We report the amount of time spent in minutes (explained later), the number of vertices and hexes of the generated hex meshes, the minimum and average scaled Jacobian $J_{\text{min}}, J_{\text{avg}}$, and the maximum and mean Hausdorff distance $d_{\text{max}}, d_{\text{avg}}$ (scaled by the diagonal length of the bounding box of the input mesh). Both d_{max} and d_{avg} are computed from the point-to-mesh distances of $50k$ uniformly sampled points from the considered mesh surfaces. The results of Fang et al. [2016]; Fu et al. [2016]; Guo et al. [2020]; Livesu et al. [2020]; Takayama [2019]; Yu et al. [2014] are obtained from their papers and www.hexalab.net [Bracci et al. 2019]. The unavailable entries are marked as '-'. For Fu et al. [2016], the reported time is only for the PolyCube construction step and does not include the final hex meshing step. The reported time of our method is the recorded amount of time for an expert user to generate the corresponding hex mesh. Since the results for different methods are generated on different machines, the reported runtimes are not directly comparable, and we include them only for the sake of completeness. When optimizing for hex quality in our method, the expert prioritizes improving the worst scaled Jacobian values (large λ_{custom}) and minimizing the Hausdorff distance (large $\lambda_{m\to 0}, \lambda_{0\to m}$).

Model	Time (m)	# vert/# hex	$J_{\text{min}}/J_{\text{avg}}$	$d_{\text{max}}/d_{\text{avg}}(\times 10^{-3})$
Armadillo [Fu et al. 2016]	≥ 0.46	87303/78376	0.265/0.909 ± 0.080	4.59/-
Armadillo [Gao et al. 2019]	320.97	72728/60340	0.159/0.779 ± 0.023	4.9/-
Armadillo (Ours)	21	21836/24709	**0.569**/**0.928** ± 0.071	**2.806**/0.238
Bimba [Fu et al. 2016]	≥ 0.40	73104/67039	0.361/0.935 ± 0.068	**3.69**/-
Bimba [Gao et al. 2019]	168.90	63679/55035	0.056/0.792 ± 0.026	4.9/-
Bimba [Livesu et al. 2020]	-	1973/1560	0.408/0.881 ± 0.026	-/-
Bimba (Ours)	8	26191/28920	**0.542**/**0.952** ± 0.056	4.590/0.160
Bunny [Yu et al. 2014]	14.7	47549/42543	-0.948/0.900 ± 0.177	18.8/-
Bunny [Fu et al. 2016]	≥ 0.46	65603/59841	0.422/0.942 ± 0.062	4.28/-
Bunny [Takayama 2019]	-	3724/2832	-0.771/0.749 ± 0.242	-/-
Bunny (Ours)	6	17230/19145	**0.601**/**0.944** ± 0.060	**3.280**/0.276
Buste [Fu et al. 2016]	≥ 0.51	86595/79294	0.439/0.943 ± 0.057	**3.24**/-
Buste (Ours)	8	22973/25514	**0.620**/**0.949** ± 0.055	3.432/0.218
Dancing children [Fu et al. 2016]	≥ 0.27	36353/30691	0.251/0.878 ± 0.099	**5.11**/-
Dancing children (Ours)	21	26718/30770	**0.502**/**0.907** ± 0.074	16.718/0.172
Dilo [Gao et al. 2019]	89.61	100110/84977	0.100/0.791 ± 0.022	**4.2**/-
Dilo (Ours)	15	32622/36500	**0.551**/**0.949** ± 0.067	6.611/0.100
Dragon [Fang et al. 2016]	125.93	127360/114178	0.162/0.919 ± 0.084	**12.6**/0.278
Dragon [Fu et al. 2016]	≥ 0.76	118610/106244	0.265/0.862 ± 0.106	16.1/-
Dragon (Ours)	19	35845/40421	**0.391**/**0.927** ± 0.073	13.438/0.154
Fandisk [Takayama 2019]	-	2404/1774	0.217/0.905 ± 0.114	-/-
Fandisk [Guo et al. 2020]	5.69	45156/39858	0.242/**0.959** ± 0.020	4.151/0.044
Fandisk (Ours)	7	32488/35841	**0.500**/0.942 ± 0.077	**3.993**/0.056
Hanger [Gao et al. 2019]	25.60	33002/26918	0.155 /0.828 ± 0.028	**2.3**/-
Hanger [Takayama 2019]	-	2229/1382	0.333 /**0.944** ± 0.094	-/-
Hanger [Guo et al. 2020]	8.76	9411/7080	0.412 /0.881 ± 0.118	-/-
Hanger (Ours)	10	8798/10500	**0.559**/0.870 ± 0.108	3.154/0.111
Joint [Fang et al. 2016]	5.10	5181/3785	0.778/**0.984** ± 0.032	8.08/0.254
Joint [Livesu et al. 2020]	-	751/456	**0.798** /0.949 ± 0.045	-/-
Joint [Takayama 2019]	-	2711/2010	0.249 /0.927 ± 0.118	-/-
Joint [Guo et al. 2020]	3.33	39658/33565	0.683/0.980 ± 0.032	-/-
Joint (Ours)	8	37227/41768	0.735/0.976 ± 0.049	**3.995**/0.120
Kitten [Fang et al. 2016]	2.30	14459/11941	0.435/0.922 ± 0.077	12.10/0.652
Kitten [Livesu et al. 2020]	-	2126/1728	-0.002/0.744 ± 0.204	-/-
Kitten (Ours)	8	11531/12929	**0.648**/**0.943** ± 0.059	**4.604**/0.251
Lock [Guo et al. 2020]	6.33	7634/5224	0.052 /0.887 ± 0.122	-/-
Lock (Ours)	11	35502/40645	**0.545**/**0.899** ± 0.093	1.996/0.060
Rocker arm [Yu et al. 2014]	16.6	10078/7874	-0.506/0.869 ± 0.210	7.08/-
Rocker arm [Fang et al. 2016]	61.85	20680/17594	0.464/0.936 ± 0.071	8.06/0.421
Rocker arm [Fu et al. 2016]	≥ 0.15	27174/23421	0.477/0.924 ± 0.074	5.27/-
Rocker arm [Takayama 2019]	-	2651/1858	-0.189/0.805 ± 0.200	-/-
Rocker arm (Ours)	14	40929/45789	**0.689**/**0.940** ± 0.055	**3.696**/0.163
Sculpt [Livesu et al. 2020]	-	327/168	**0.806**/0.918 ± 0.044	-/-
Sculpt [Guo et al. 2020]	4.35	25562/21695	0.528/**0.949** ± 0.044	**1.927**/0.108
Sculpt (Ours)	13	18995/21488	0.614/0.917 ± 0.075	6.321/0.202


$J_{\min}$: 0.403, J_{avg} : 0.878 ± 0.119
$V_{\min}$: 0.557, V_{avg} : 1.417 ± 0.324
$J_{\min}$: 0.479, J_{avg} : 0.888 ± 0.108
$V_{\min}$: 0.476, V_{avg} : 1.411 ± 0.366
$J_{\min}$: 0.531, J_{avg} : 0.899 ± 0.096
$V_{\min}$: 0.367, V_{avg} : 1.404 ± 0.416
$J_{\min}$: 0.566, J_{avg} : 0.911 ± 0.084
$V_{\min}$: 0.23, V_{avg} : 1.399 ± 0.471
$J_{\min}$: 0.617, J_{avg} : 0.925 ± 0.070
$V_{\min}$: 0.018, V_{avg} : 1.388 ± 0.555
$J_{\min}$: 0.484, J_{avg} : 0.823 ± 0.135
$V_{\min}$: 2.477, V_{avg} : 5.395 ± 1.358
$J_{\min}$: 0.505, J_{avg} : 0.838 ± 0.125
$V_{\min}$: 2.063, V_{avg} : 5.340 ± 1.526
$J_{\min}$: 0.532, J_{avg} : 0.853 ± 0.113
$V_{\min}$: 1.580, V_{avg} : 5.286 ± 1.740
$J_{\min}$: 0.557, J_{avg} : 0.871 ± 0.100
$V_{\min}$: 0.963, V_{avg} : 5.227 ± 2.011
$J_{\min}$: 0.610, J_{avg} : 0.893 ± 0.084
$V_{\min}$: 0.040, V_{avg} : 5.170 ± 2.372

(a) $(\lambda_{\text{angle}}, \lambda_{\text{custom}}) = (1.0, 0.0)$ (b) $(\lambda_{\text{angle}}, \lambda_{\text{custom}}) = (0.75, 0.25)$ (c) $(\lambda_{\text{angle}}, \lambda_{\text{custom}}) = (0.5, 0.5)$ (d) $(\lambda_{\text{angle}}, \lambda_{\text{custom}}) = (0.25, 0.75)$ (e) $(\lambda_{\text{angle}}, \lambda_{\text{custom}}) = (0.0, 1.0)$

Fig. 16. Trade-off between angle preservation (large *conformal* parameter λ_{angle}) and scaled Jacobian values (large *custom* parameter λ_{custom} with the scaled Jacobian energy). For each hex mesh we show the minimum scaled Jacobian $J_{\min}$, the average scaled Jacobian J_{avg}, the minimum (unscaled) Jacobian $V_{\min}$, and the average (unscaled) Jacobian V_{avg}. The standard deviation across hexes is shown after the ± sign. Both $V_{\min}$ and V_{avg} are multiplied by 10^5. The bottom image of each row shows the hex elements with scaled Jacobian less than 0.8. Yellow indicates an interior quad. As we gradually decrease the λ_{angle} and increase λ_{custom}, we see that $J_{\min}$ and J_{avg} increase while $V_{\min}$ decreases, and the standard deviation of the (unscaled) Jacobian grows, indicating the hex mesh has more unevenly sized elements. The user can conduct such trade-off exploration interactively to decide whether they want more volume preservation or better scaled Jacobian values. To produce each mesh, we start from the same initialized inversion-free hex mesh and use default weights for all parameters except for λ_{angle} and λ_{custom}. We then run the optimization for 1000 steps. We use Eq. (18) to improve the worst element for both the distortion and the custom energy.


$J_{\min}$: 0.606, J_{avg} : 0.945 ± 0.063
$d_{\max}$: 41.814, d_{avg} : 9.978
$J_{\min}$: 0.557, J_{avg} : 0.918 ± 0.067
$d_{\max}$: 16.857, d_{avg} : 1.461
$J_{\min}$: 0.519, J_{avg} : 0.905 ± 0.074
$d_{\max}$: 13.369, d_{avg} : 0.669
$J_{\min}$: 0.492, J_{avg} : 0.896 ± 0.082
$d_{\max}$: 13.577, d_{avg} : 0.322
$J_{\min}$: 0.417, J_{avg} : 0.890 ± 0.089
$d_{\max}$: 13.677, d_{avg} : 0.202

(a) $\lambda_{m\to 0} = \lambda_{0\to m} = 0.0$ (b) $\lambda_{m\to 0} = \lambda_{0\to m} = 0.25$ (c) $\lambda_{m\to 0} = \lambda_{0\to m} = 1.0$ (d) $\lambda_{m\to 0} = \lambda_{0\to m} = 4.0$ (e) $\lambda_{m\to 0} = \lambda_{0\to m} = 16.0$

Fig. 17. Trade-off between hex element quality and the approximation level of the input surface (controlled by the *projection* parameter $\lambda_{m\to 0}$ and the *details* parameter $\lambda_{0\to m}$). We add a blue translucent shell of the input mesh surface to visualize the discrepancy. For each hex mesh we show the minimum scaled Jacobian $J_{\min}$, the average scaled Jacobian J_{avg}, the symmetric Hausdorff distance $d_{\max}$, and the average Hausdorff distance $d_{\min}$ (computed similarly to the ones in Table 2). The standard deviation across hex elements is shown after the ± sign. As we increase $\lambda_{m\to 0}$ and $\lambda_{0\to m}$, the average Hausdorff distance becomes smaller but at the cost of worst hex quality (i.e., reduced $J_{\min}$ and J_{avg}). The experiment setup is the same as that of Fig. 16, except the parameters being changed are $\lambda_{m\to 0}$ and $\lambda_{0\to m}$.

participants agreed that they were able to to make fine-grained adjustments to their final mesh, and 13 out of 21 preferred their meshes compared to hex meshes obtained from automatic algorithms (on https://www.hexalab.net). Users "appreciated the speed and the interpretability of [the] optimizers and hyperparameters" and were "particularly impressed with the final hex mesh." Participants wrote that "[the tool] allowed a lot of manual changes but the default parameters worked well which was great as an inexperienced user," and "the tool provides a good balance of personal customization (i.e., what parts to capture/focus on) and computer-assisted optimization (no need to worry too much about placement and sizing of the cubes

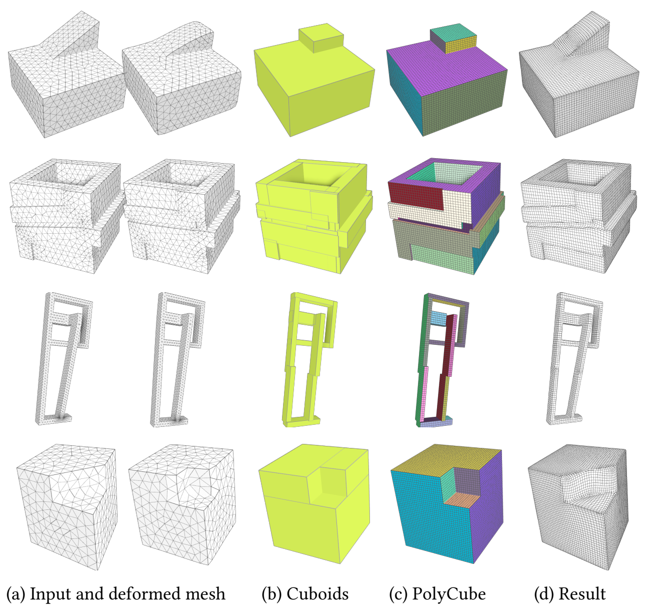


Fig. 18. Challenging cases for deformation-based PolyCube methods and frame-field-based hexahedral meshing. In the first three rows the deformation fails to produce the needed stairs, even with a large *cubeness* value. Frame-field-based hexahedral meshing approaches invariably produce unmeshable singularities on the mesh in the fourth row [Viertel et al. 2016]. Our method of distance-field-based PolyCube optimization successfully meshes each case. A small amount of user interaction (less than 5 minutes total) is needed for the models in the middle two rows. *Ramp* (top row): 47224 hexes, $J_{\min} = 0.392, J_{\text{avg}} = 0.942, d_{\max} = 3.350$. *Ex14* (second row): 48756 hexes, $J_{\min} = 0.472, J_{\text{avg}} = 0.896, d_{\max} = 5.465$. *Ex16* (third row): 9256 hexes, $J_{\min} = 0.478, J_{\text{avg}} = 0.872, d_{\max} = 7.244$. *Notch7* (last row): 68704 hexes, $J_{\min} = 0.579, J_{\text{avg}} = 0.957, d_{\max} = 3.861$.

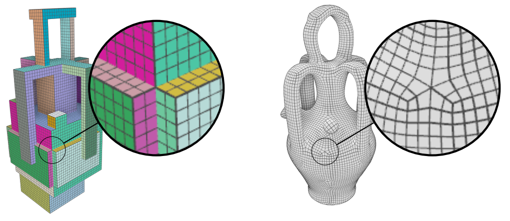


Fig. 19. An example where the PolyCube has corners incident to 6 charts (left), and the resulting hex mesh (right).

[...])." This highlights the fact that our method enables low-level control without adding superfluous complexity.

Fig. 20 shows some hex meshes that were produced during the user study.

7 CONCLUSION AND FUTURE WORK

The task of converting a tetrahedral mesh to a hex mesh is a highly under-constrained problem, and many trade-offs must be considered to produce a satisfactory result. Different applications and aesthetics motivate different choices for the placement and number of singularities, articulation or smoothing of sharp features, simplicity versus fidelity, and so on. Rather than proposing a one-size-fits-all algorithm like many previous works, we instead design a comprehensive interactive system, consisting of several stages. At each stage, the user can intervene, making the design choices that influence the final output. While our system has a shallow learning curve for the non-expert user due to robust default parameter choices, as demonstrated by our user study, it also allows extremely fine-grained control, yielding high-quality meshes that achieve or exceed state-of-the-art quantitative metrics.

Our system generates high-quality hex meshes on all the test models. Below we discuss some exciting directions to further extend our pipeline in future work.

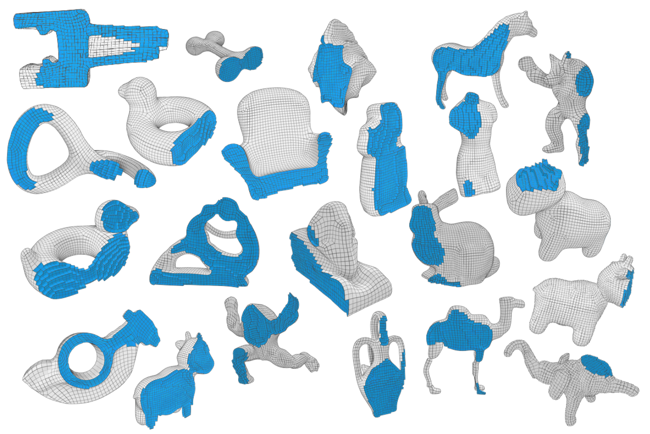


Fig. 20. A sampling of hex meshes produced by novice users as part of our user study.

Interior singularity support. Like other PolyCube-based methods, the singularities on the hex meshes produced by our system are restricted to the surface (except for a layer of global padding). Allowing interior singularities could improve mesh quality (e.g., results of *sculpt* from Livesu et al. [2020] and Guo et al. [2020] in Table 2). One strategy would be to augment our pipeline by employing the selective padding of Cherchi et al. [2019] instead of global padding or by injecting interior singularities by cutting PolyCube edges open [Guo et al. 2020]. Alternatively, we could follow Fang et al. [2016] by breaking all tunnel loops of the input shape and adding additional constraints during optimization to glue back the input.

Invariance to orientation. Although our user-guided deformation (Fig. 3) can orient coarse features of the input, the deformation still largely depends on the initial orientation of the shape. For instance, for the *dragon* model, an ideal deformation map would straighten the body instead of creating unnecessary zig-zag patterns. A potential solution is to use a frame field to guide the deformation, like in [Fang et al. 2016].

Sharp feature preservation. For certain classes of input shapes, it may be desirable to preserve sharp features explicitly. One way to achieve this would be to integrate feature-aware PolyCube generation from Guo et al. [2020] by allowing the user to draw feature curves on the input surface, which would then get mapped to PolyCube edges. Then, during the hexahedralization stage, we can include another energy term that favors pairs of orthogonal faces that meet on a feature curve, such as Eq. (5) from Guo et al. [2020].

Topological consistency. We rely on the user to make sure the topology is correct, allowing them to repair issues during the decomposition stage (Fig. 6) or the discretization stage (Fig. 7). While directly incorporating topological priors into gradient-based optimization is a challenging open problem, we could detect incorrect topology and interactively alert the user to ease the process.

Concavities and tunnels. For highly complex models with concavities and tunnels, it could be strenuous to manually place cuboids in the decomposition stage while avoiding all the empty regions. One promising future direction is to allow users to construct a PolyCube for the complement of the shape and then to cut out this complement from the primal PolyCube. Our distance-field-based formulation is particularly suitable for this approach. For instance, the signed distance field of the complement can be obtained by reversing signs, and getting the signed distance field of the subtracted PolyCube is similarly straightforward. Orthogonally, we can extend the discretization stage to allow users to make interior edits by introducing ways to hide layers of voxels.

Additional UI features. The feedback gathered during our user study could be incorporated to further enhance the user experience. Several users commented that the PolyCube optimization may create gaps (Fig. 12(b)) or unnecessary stairs (Fig. 6(d)) despite already being at a local minimum. We could add a postprocessing step after the decomposition stage to glue nearby cuboids together and remove superfluous stairs, e.g., using the erasing-and-filling operators from Yang et al. [2019].

The experience of navigating between the various stages of our pipeline could be improved by propagating visual cues across the stages. For instance, if the user notices a region that needs changing (such as an undesired singularity or topological issue) in the final stage of the pipeline only after visualizing the generated hex mesh, it may be hard to locate the corresponding region in the decomposition or discretization stage. Visualizing correspondences between the final hex mesh and the voxelized PolyCube would simplify this workflow.

Some users reported difficulties avoiding inversions when placing landmarks. One way to avoid inversions caused by arbitrarily placing and fixing surface vertices would be to to encode the new positions as a soft constraint and use an energy term similar to Eq. (13) to gradually deform the mesh so that it satisfies the new positional constraints.

ACKNOWLEDGMENTS

The MIT Geometric Data Processing group acknowledges the generous support of Army Research Office grant W911NF2010168, of Air Force Office of Scientific Research award FA9550-19-1-031, of National Science Foundation grants IIS-1838071 and CHS-1955697, from the CSAIL Systems that Learn program, from the MIT–IBM Watson AI Laboratory, from the Toyota–CSAIL Joint Research Center, from a gift from Adobe Systems, from an MIT.nano Immersion Lab/NCSOFT Gaming Program seed grant, and from the Skoltech–MIT Next Generation Program. We would also like to thank Zoya Bylinskii for valuable advice on designing and conducting the user study.

REFERENCES

David Bommes, Henrik Zimmer, and Leif Kobbelt. 2009. Mixed-integer quadrangulation. *ACM Transactions On Graphics (TOG)* 28, 3 (2009), 1–10.

Matteo Bracci, Marco Tarini, Nico Pietroni, Marco Livesu, and Paolo Cignoni. 2019. HexaLab. net: An online viewer for hexahedral meshes. *Computer-Aided Design* 110 (2019), 24–36.

Marcel Campen and Leif Kobbelt. 2014. Dual strip weaving: Interactive design of quad layouts using elastica strips. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 1–10.

Gianmarco Cherchi, Pierre Alliez, Riccardo Scateni, Max Lyon, and David Bommes. 2019. Selective padding for polycube-based hexahedral meshing. In *Computer graphics forum*, Vol. 38. Wiley Online Library, 580–591.

Etienne Corman and Keenan Crane. 2019. Symmetric moving frames. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–16.

Hans-Christian Ebke, Patrick Schmidt, Marcel Campen, and Leif Kobbelt. 2016. Interactively controlled quad remeshing of high resolution 3D models. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–13.

David Eppstein and Elena Mumford. 2010. Steinitz theorems for orthogonal polyhedra. In *Proceedings of the twenty-sixth annual symposium on Computational geometry*. 429–438.

Xianzhong Fang, Weiwei Xu, Hujun Bao, and Jin Huang. 2016. All-hex meshing using closed-form induced polycube. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–9.

Xiao-Ming Fu, Chong-Yang Bai, and Yang Liu. 2016. Efficient volumetric polycube-map construction. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 97–106.

Xiao-Ming Fu, Yang Liu, and Baining Guo. 2015. Computing locally injective mappings by advanced MIPS. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–12.

Xifeng Gao, Zhigang Deng, and Guoning Chen. 2015. Hexahedral mesh re-parameterization from aligned base-complex. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–10.

Xifeng Gao, Hanxiao Shen, and Daniele Panozzo. 2019. Feature Preserving Octree-Based Hexahedral Meshing. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 135–149.

Vladimir Garanzha, Igor Kaporin, Liudmila Kudryavtseva, François Protais, Nicolas Ray, and Dmitry Sokolov. 2021. Foldover-Free Maps in 50 Lines of Code. *ACM Trans. Graph.* 40, 4, Article 102 (July 2021), 16 pages. https://doi.org/10.1145/3450626.3459847

James Gregson, Alla Sheffer, and Eugene Zhang. 2011. All-hex mesh generation via volumetric polycube deformation. In *Computer graphics forum*, Vol. 30. Wiley Online Library, 1407–1416.

Hao-Xiang Guo, Xiaohan Liu, Dong-Ming Yan, and Yang Liu. 2020. Cut-enhanced PolyCube-maps for feature-aware all-hex meshing. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 106–1.

Yixin Hu, Teseo Schneider, Bolun Wang, Denis Zorin, and Daniele Panozzo. 2020. Fast tetrahedral meshing in the wild. *ACM Transactions on Graphics (TOG)* 39, 4 (2020), 117–1.

Jin Huang, Tengfei Jiang, Zeyun Shi, Yiying Tong, Hujun Bao, and Mathieu Desbrun. 2014. l1-based construction of polycube maps from complex shapes. *ACM Transactions on Graphics (TOG)* 33, 3 (2014), 1–11.

Jin Huang, Yiying Tong, Hongyu Wei, and Hujun Bao. 2011. Boundary aligned smooth 3D cross-frame field. *ACM transactions on graphics (TOG)* 30, 6 (2011), 1–8.

Wenzel Jakob, Marco Tarini, Daniele Panozzo, and Olga Sorkine-Hornung. 2015. Instant field-aligned meshes. *ACM Trans. Graph.* 34, 6 (2015), 189–1.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

Juncong Lin, Xiaogang Jin, Zhengwen Fan, and Charlie CL Wang. 2008. Automatic polycube-maps. In *International Conference on Geometric Modeling and Processing*. Springer, 3–16.

Heng Liu, Paul Zhang, Edward Chien, Justin Solomon, and David Bommes. 2018. Singularity-constrained octahedral fields for hexahedral meshing. *ACM Trans. Graph.* 37, 4 (2018), 93–1.

Lei Liu, Yongjie Zhang, Yang Liu, and Wenping Wang. 2015. Feature-preserving T-mesh construction using skeleton-based polycubes. *Computer-Aided Design* 58 (2015), 162–172.

Marco Livesu, Nico Pietroni, Enrico Puppo, Alla Sheffer, and Paolo Cignoni. 2020. LoopyCuts: Practical Feature-Preserving Block Decomposition for Strongly Hex-Dominant Meshing. *ACM Trans. Graph.* 39, 4, Article 121 (July 2020), 17 pages. https://doi.org/10.1145/3386569.3392472

Marco Livesu, Alla Sheffer, Nicholas Vining, and Marco Tarini. 2015. Practical hex-mesh optimization via edge-cone rectification. *ACM Transactions on Graphics (TOG)* 34, 4 (2015), 1–11.

Marco Livesu, Nicholas Vining, Alla Sheffer, James Gregson, and Riccardo Scateni. 2013. PolyCut: monotone graph-cuts for PolyCube base-complex construction. *ACM Transactions on Graphics (TOG)* 32, 6 (2013), 1–12.

Zoë Marschner, David Palmer, Paul Zhang, and Justin Solomon. 2020. Hexahedral Mesh Repair via Sum-of-Squares Relaxation. In *Computer Graphics Forum*, Vol. 39. Wiley Online Library, 133–147.

Matthias Nieser, Ulrich Reitebuch, and Konrad Polthier. 2011. Cubecover–parameterization of 3d volumes. In *Computer graphics forum*, Vol. 30. Wiley Online Library, 1397–1406.

David Palmer, David Bommes, and Justin Solomon. 2020. Algebraic Representations for Volumetric Frame Fields. *ACM Trans. Graph.* 39, 2, Article 16 (April 2020), 17 pages. https://doi.org/10.1145/3366786

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

François Protais, Maxence Reberol, Nicolas Ray, Etienne Corman, Franck Ledoux, and Dmitry Sokolov. 2020. Robust Quantization for Polycube Maps. (2020).

Roshan Quadros. 2021. The CUBIT Geometry and Meshing Toolkit. https://cubit.sandia.gov/.

Nicolas Ray, Dmitry Sokolov, and Bruno Lévy. 2016. Practical 3D frame field generation. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–9.

Jason F Shepherd and Chris R Johnson. 2008. Hexahedral mesh generation constraints. *Engineering with Computers* 24, 3 (2008), 195–213.

Dmitriy Smirnov, Matthew Fisher, Vladimir G Kim, Richard Zhang, and Justin Solomon. 2020. Deep parametric shape predictions using distance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 561–570.

Dmitry Sokolov and Nicolas Ray. 2015. *Fixing normal constraints for generation of polycubes.* Research Report. LORIA. https://hal.inria.fr/hal-01211408

Justin Solomon, Amir Vaxman, and David Bommes. 2017. Boundary element octahedral fields in volumes. *ACM Transactions on Graphics (TOG)* 36, 4 (2017), 1.

Kenshi Takayama. 2019. Dual sheet meshing: An interactive approach to robust hexahedralization. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 37–48.

Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. 2004. Polycube-maps. *ACM transactions on graphics (TOG)* 23, 3 (2004), 853–860.

Shubham Tulsiani, Hao Su, Leonidas J Guibas, Alexei A Efros, and Jitendra Malik. 2017. Learning shape abstractions by assembling volumetric primitives. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2635–2643.

Ryan Viertel, Matthew L Staten, and Franck Ledoux. 2016. *Analysis of Non-Meshable Automatically Generated Frame Fields.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Yang Yang, Xiao-Ming Fu, and Ligang Liu. 2019. Computing Surface PolyCube-Maps by Constrained Voxelization. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 299–309.

Wuyi Yu, Kang Zhang, Shenghua Wan, and Xin Li. 2014. Optimizing polycube domain construction for hexahedral remeshing. *Computer-Aided Design* 46 (2014), 58–68.

Shangyou Zhang. 2005. Subtetrahedral test for the positive Jacobian of hexahedral elements. *preprint* (2005).

Hui Zhao, Xuan Li, Wencheng Wang, Xiaoling Wang, Shaodong Wang, Na Lei, and Xiangfeng Gu. 2019. Polycube Shape Space. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 311–322.

A MORE IMPLEMENTATION DETAILS

A.1 Heuristics in Section 3.2

For the two heuristics in the *Add* operation of Section 3.2, we create a $32 \times 32 \times 32$ uniform grid over the shape and then compute the signed distances of all grid points to both the PolyCube and the deformed mesh. For the distance-based heuristic, we can then detect an uncovered point of the deformed mesh that is furthest away from the PolyCube. For the volume-based heuristic, to find the largest cuboid inside the deformed mesh and outside the PolyCube, we extend the algorithm for finding the largest rectangle in a histogram to 3D with time complexity $O(n^4)$ with $n = 32$, similar to Yang et al. [2019]. For the *Subtract* operation, the implementation is similar.

A.2 Custom CUDA functions

We implement the following CUDA functions to enable our pipeline to run at interactive speed. Each functions takes (P, M) as the input, where P is a list of query points and M is a representation of a triangular or tetrahedral mesh, and returns a quantity for each query point. In our case, P can contain tens of thousands of points, and M can have up to 10^5 cells. In our CUDA code, we use a thread block of dimension 128 for each query point and evenly divide the mesh triangles or tetrahedrons among the 128 threads to process the query point in a thread block. The results from all threads are collected using parallel reduction.

Below are details of the implementations for each CUDA device function for a pair of point and triangle/tetrahedron.

Point-triangle projection. Let $p \in \mathbb{R}^3$ be the point we want to project onto a triangle with vertices $v_0, v_1, v_2 \in \mathbb{R}^3$. Denote $e_1 = v_1 - v_0$ and $e_2 = v_2 - v_0$. Consider first projecting p onto the plane spanned by v_0, v_1, v_2. This can be done by solving the following least-squares problem:

$$d(p, \{v_i\}) := \min_{w_1, w_2 \in \mathbb{R}} \|p - (v_0 + w_1 e_1 + w_2 e_2)\|^2. \tag{20}$$

The closed-form solution is given by, for $i = 1, 2$,

$$w_i = \frac{(p - v_0) \cdot \left(e_i - \frac{(e_{2-i} \cdot e_i) e_{2-i}}{\|e_{2-i}\|^2}\right)}{\|e_i\|^2 - \frac{(e_{2-i} \cdot e_i)^2}{\|e_{2-i}\|^2}}. \tag{21}$$

Let $w_0 = 1 - w_1 - w_2$. Note (w_0, w_1, w_2) is the barycentric coordinate of p in this triangle. If $w_i \geq 0$ for $i \in \{0, 1, 2\}$, then the projection of p lands inside the triangle. Otherwise, we project p onto line segment $(p_0, p_1), (p_1, p_2), (p_2, p_0)$ and choose the projection with the smallest distance.

When backpropagating through the proximity energy in Eq. (9), we need to compute the gradient of the projected point q with respect to both p and v_0, v_1, v_2:

- If $w_i \geq 0$ for $i \in \{0, 1, 2\}$, i.e., q lands inside the triangle, then we have $q = \sum_{i=0}^2 w_i v_i$, so $\frac{\partial q_k}{\partial p} = \sum_{i=0}^2 \frac{\partial w_i}{\partial p} (v_i)_k$, where $k \in \{x, y, z\}$, and each $\frac{\partial w_i}{\partial p}$ is a constant deduced from Eq. (21) that can be precomputed for each triangle ahead of time. The case for when q is outside the triangle can be handled similarly.
- To compute the gradient of q with respect to v_i's, direct calculation from Eq. (21) can be cumbersome. This is only needed for Eq. (11), so we just need to differentiate $d := d(p, \{v_i\})$ from Eq. (20) with respect to the v_i's. By the Envelope Theorem, we have $\frac{\partial d}{\partial v_i} = w_i(q - p)$ for each i.

Point-tetrahedron projection. To compute the pullback energy from Eq. (13), we need to project a point onto the input tetrahedral mesh. Consider projecting a point p to a single tetrahedron with vertices $\{v_i\}_{i=0}^3$. Similar to Eq. (20), we can find the barycentric coordinates of the projection by solving a least-squares problem but with three variables rather than two:

$$d(p, \{v_i\}) := \min_{w_1, w_2, w_3 \in \mathbb{R}} \|p - (v_0 + w_1 e_1 + w_2 e_2 + w_3 e_3)\|^2. \tag{22}$$

Taking the derivative and setting it equal to zero gives a linear system of the form $Aw = b$, where A is invertible and does not depend on p. We can precompute the inverse of A for each tetrahedron. If any $w_i < 0$, then we project p onto the four triangular faces of the tetrahedron in the same way as in the last paragraph. We do not need gradient information for point-tetrahedron projection.

Point-in-mesh inclusion test. To compute the signed distance field of the deformed tet mesh in Eq. (6), in addition to computing point-triangle projection, we need to determine the sign for each query point, i.e., whether it is inside the mesh or not. Our implementation relies on the fact that we know the volume mesh on which we want to test: we check if the query point is inside any tetrahedron, which can be done by checking if the point is on the correct side of each face of the tetrahedron. We found this approach robust even with single-precision floating-point computation.