Data-Driven Interactive Quadrangulation

Giorgio Marcias[*] CNR of Italy, University of Pisa
Kenshi Takayama[†] National Institute of Informatics
Nico Pietroni[‡] CNR of Italy
Daniele Panozzo[§] ETH Zurich
Olga Sorkine-Hornung[¶] ETH Zurich
Enrico Puppo[‖] University of Genova
Paolo Cignoni[**] CNR of Italy

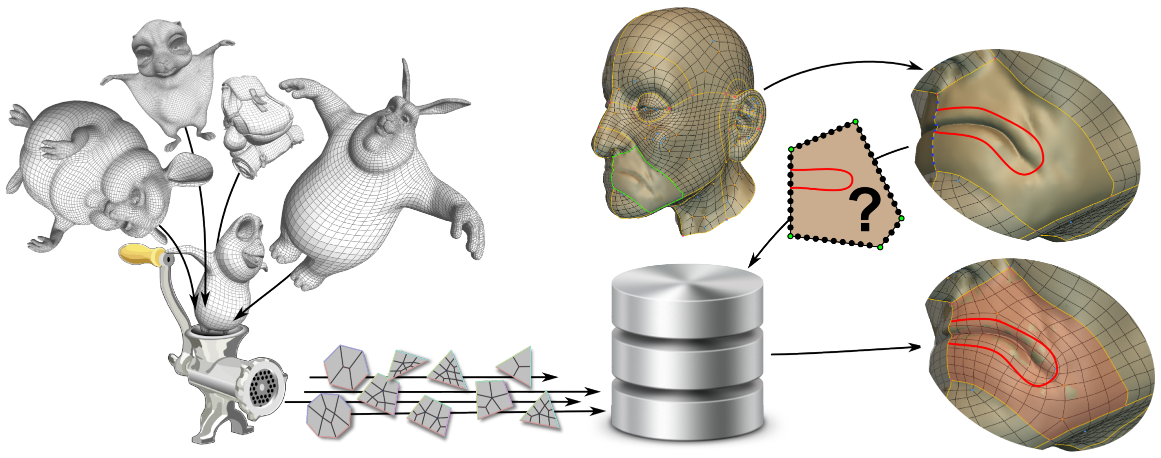

Figure 1: *Given a set of hand-made quadrangulated input models, our algorithm learns the quadrangulation patterns used to design them. This knowledge is employed in a sketch-based retopology tool to interactively quadrangulate user-sketched patches with subdivided edges. The user can sketch strokes inside a patch to suggest a a specific edge-flow and the system automatically selects a quadrangulation following it.*

Abstract

We propose an interactive quadrangulation method based on a large collection of patterns that are learned from models manually designed by artists. The patterns are distilled into compact quadrangulation rules and stored in a database. At run-time, the user draws strokes to define patches and desired edge flows, and the system queries the database to extract fitting patterns to tessellate the sketches' interiors. The quadrangulation patterns are general and can be applied to tessellate large regions while controlling the positions of the singularities and the edge flow. We demonstrate the effectiveness of our algorithm through a series of live retopology sessions and an informal user study with three professional artists.

CR Categories: I.3.5 [Computer Graphics]: Computational geometry and object modeling—Curve, surface, solid and object representations.

Keywords: quad meshing, polygon quadrangulation, retopology

1 Introduction

Quadrilateral meshes are ubiquitously used in the animation and CAD industry as control grids for subdivision surfaces and NURBS. Many approaches have been proposed to convert an unstructured, triangulated surface into a high-quality polygonal mesh, ranging from local decimation strategies to global field-aligned optimizations [Bommes et al. 2013b]. While dense quadrilateral meshes can be robustly created with existing methods, the creation of coarse patch layouts, or so-called *retopology*, is a challenging open problem, since the connection between the geometry of a surface and its ideal, application-dependent quad mesh is weak or nonexistent. For example, if the mesh was intended to be used for animation, its connectivity should be tailored to its articulation and optimized to reduce skinning deformation artifacts; such properties are impossible to automatically extract from static meshes.

In the industry, coarse quad layouts are manually created by professional designers, who employ their semantic knowledge and experience to adjust the layout in the context of the particular application

[*]e-mail:giorgio.marcias@isti.cnr.it
[†]e-mail:takayama@nii.ac.jp
[‡]e-mail:nico.pietroni@isti.cnr.it
[§]e-mail:panozzo@inf.ethz.ch
[¶]e-mail:sorkine@inf.ethz.ch
[‖]e-mail:puppo@disi.unige.it
[**]e-mail:paolo.cignoni@isti.cnr.it

needs. Typical modeling systems used in the industry [Autodesk 2007; Pilgway 2013; Pixologic 2013] allow the user to manually draw vertices and edges on a surface. Since this manual procedure is time-consuming and error-prone, a series of sketch-based retopology approaches [Campen and Kobbelt 2014; Takayama et al. 2014] have been proposed to assist the user, automating a large part of the process while allowing the user to efficiently modify the topology of the layouts without having to start from scratch.

We propose a novel data-driven method for interactive quadrangulation that supports the design of complex tessellations from sparse sketches. The contributions of this paper can be summarized as follows:

- A sketch-based user interface to specify the desired edge flow in the final tessellation;
- A data-driven approach to explore a wide range of quadrangulations of a polygonal patch.

The data-driven exploration approach is based on a large collection of tessellated exemplars that are extracted from a database composed of manually designed quad meshes. The effectiveness of our approach lies in the integration of these two contributions: the use of a database composed of manually-designed quadrangulations offers both generality and plausibility of the proposed tassellations; concurrently the use of the sketch interface allows the artist to designate the final solution in a very efficient manner. While independently interesting from a research perspective, the two contributions must be combined to effectively design quadrangulations: on one hand, the sketching interface is useful to select among a large set of possible quadrangulations and, on the other hand, having a large collection of tessellations becomes impractical without an efficient exploration strategy (Section 5).

We integrate our algorithm into the interactive sketch-based retopology system of [Takayama et al. 2013], and we demonstrate its practical impact in an informal user study, where three professional artists used the system to retopologize three high-resolution models.

Our algorithm can automatically fill large polygonal regions (up to 34 sides with our current database) while allowing careful tuning of small details in critical areas. The different tessellations presented to the user are sorted taking into account user-controlled structural and geometric criteria.

We made an initial step towards learning-based quadrangulation by limiting our focus on the topological aspect. Our database is based on mesh connectivity because it is the most crucial quality measure in retopology and the most difficult due to its combinatorial nature (see, e.g., Figure 14).

2 Related work

Automatic quad meshing. Algorithms for the automatic generation of quad meshes have been studied extensively; see [Bommes et al. 2013b] for a recent comprehensive survey. In particular, many methods have been proposed to design coarse quad layouts [Bommes et al. 2011; Tarini et al. 2011; Campen et al. 2012; Bommes et al. 2013a]. The method proposed by [Marcias et al. 2013] drives the remeshing process by considering the deformation the mesh undergoes during an animated sequence, to obtain a mesh that approximates well all animation frames. While the quality of these methods is high in terms of singularity placement and coarseness, they are difficult to apply in a production pipeline due to the lack of control. Small changes in the user-provided constraints may completely alter the final quadrangulation due to the global nature of the problem; the combination of this global behavior with the non-interactive nature of these algorithms makes the tuning of their parameters an unintuitive and time-consuming task.

User-assisted methods. Our work is mostly related to sketch-based retopology techniques [Takayama et al. 2013; Peng et al. 2014; Takayama et al. 2014]. Their general idea is to interactively sketch polygonal patches over the input surface, which are then automatically quadrangulated. The patches can have several sides, and each side is subdivided into a number of edges prescribed by the user [Schaefer et al. 2004; Nasri et al. 2009; Yasseen et al. 2013]. The filling patterns are procedurally generated using a greedy algorithm in [Peng et al. 2014]. In [Takayama et al. 2013; Takayama et al. 2014], a set of manually designed patterns are expanded to tessellate arbitrary polygons with up to 6 sides. Our method differs from previous approaches in the way patterns are obtained: we learn them from manually designed models and we fetch them on-the-fly according to the user constraints.

Inspired quadrangulation [Tierny et al. 2011] is also related to our method. Although we share the basic motivation, their approach is quite different: it transfers quadrangulations between surfaces on a per-part basis (e.g., head, arm, torso) via cross-parameterization. This approach does not provide precise local control over the mesh layout, while our method enables localized and flexible retopology reuse with fine-grained control over the resulting quadrangulations.

Connectivity editing operations have been developed to enable users to modify existing quad meshes by moving pairs of irregular vertices [Peng et al. 2011]. This method provides lower-level local operators, and it can be integrated with methods of the previous class to fine-tune the mesh topology. It is optimized to edit existing models and cannot be used to retopologize models from scratch.

A global, sketch-based retopology algorithm has been recently proposed in [Campen and Kobbelt 2014], which assists the user in creating edge loops; these are then transformed into chains of quadrilaterals. This method requires less manual input than the previous works, but it sacrifices the ability to locally modify the quadrangulation, since it is only possible to insert entire chains of quads.

3 Method

Our method is based on the pattern-based quadrangulation framework proposed in [Takayama et al. 2014], briefly summarized in Section 3.1. Their general idea is that an unstructured triangle mesh is retopologized incrementally by automatically quadrangulating polygonal regions sketched by the user using manually designed quadrangulation patterns. The user can impose constraints on the number of edge subdivisions at the boundary of the patch. Our method generalizes this approach by learning complex quadrangulation patterns from manually designed mesh models.

Our data-driven approach relies only on patterns which are used in manually designed meshes. This strategy ensures a more effective sampling of commonly used patterns with respect to the approaches based on procedural generation.

By analyzing meshes in a learning dataset, we extract a large number of patterns (477k in our current implementation) containing complex constellations of singularities with a plethora of internal edge-flows; this learning phase is executed off-line on a number of representative models to build a database of patterns (Section 3.2). Every pattern is uniquely represented by a code, which constitutes a fast, compressible, encoding of the pattern and acts as a primary key for database records (Section 3.3). Queries are executed at interactive rates to find the patterns that satisfy the given boundary conditions and such that the pattern edges are aligned with the desired edge flow. The

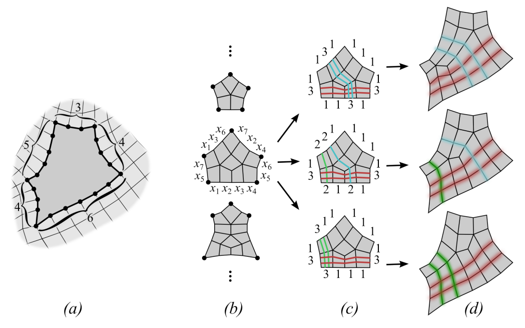
Figure 2: *The basic patch filling pipeline. A region to be filled defines a boundary constraint: five sides with their respective numbers of subdivisions (a). We search all five-sided patterns in the database (b); for each pattern, we find all possible polychord expansions (colored edge chains) matching the given boundary constraint (c); we fit the expanded patches to the input geometry (d).*

retrieved patterns are presented to the user in a sorted sequence that favors patches with higher quality (Section 4).

3.1 Pattern-based quadrangulation

The pattern-based quadrangulation approach of [Takayama et al. 2014] relies on the relation between a *boundary constraint* in the form of a polygon with subdivided *sides*, which the user draws on the surface (Figure 2(a)), and a *pattern* in the form of a quad mesh filling a polygonal domain (Figure 2(b)). The pattern can be adapted to the boundary constraint if it is possible to expand its *edge chains*, i.e., sequences of consecutive edges that make no turns (a.k.a. *edge loops* in the modeling practice), into *polychords*, i.e., corresponding chains of quads, so that the boundary of the expanded pattern matches the subdivision of the boundary constraint (Figure 2c). Finding whether or not a pattern matches a given boundary constraint has been formalized in [Takayama et al. 2014] as an integer linear programming problem (ILP), which can be solved efficiently using [Achterberg 2009].

While [Takayama et al. 2013; Takayama et al. 2014] rely on a small set of predefined patterns, we rely on a large database of learned patterns, which is queried as depicted in Figure 2. Note that we consider many possible matching patterns (Figure 2(b)), and for each pattern that matches the boundary constraint, we generate all possible expanded patterns (Figure 2(c)).

Due to the large size of our database, a query is likely to retrieve hundreds of matching patches. We provide an intuitive interface to restrict this search to patterns that follow user-defined edge-flows; moreover, we sort the patches according to user-defined topological and geometric criteria in order to facilitate selection (see Section 4 and Figure 3).

3.2 Learning quadrangulation patterns

We extract the patterns used in all our experiments from 40 manually modeled meshes of the Blender open movies “Big Buck Bunny” (25) [Blender Found. 2008] and “Sintel” (15) [Blender Found. 2010]. Our learning phase is divided into two main steps (see Figure 4):

1. Patch enumeration: we extract disk-like patches made of connected sets of quads from each mesh in the training set.
2. Pattern reduction: each patch is reduced to a pattern by collapsing its polychords into edge chains.

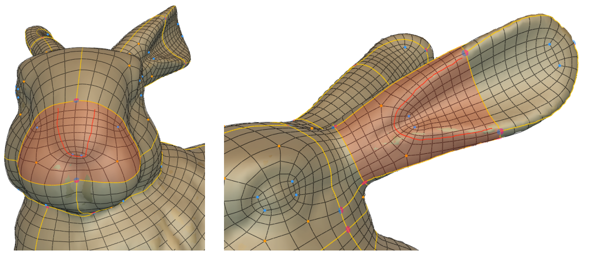
Figure 3: *The user can sketch the desired edge-flow (red line): the system retrieves from the DB only those patterns that conform the given flow and sort the generated patches accordingly.*

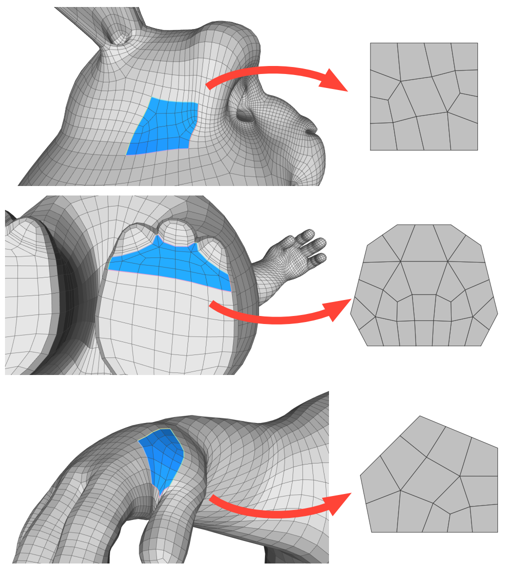
Figure 4: *Overview of the learning phase. A patch is sampled from the original mesh, and the corresponding pattern is generated by polychord collapse before storing its encoded connectivity in the database.*

This process involves a number of technical details that are discussed in the following.

Patch enumeration. Patches are enumerated by a depth-first traversal of a mesh, starting at each quad and growing a patch by attaching new quads to its boundary. The basic algorithm generates patches by adding one quad at a time. However, the combinatorial explosion of this scheme quickly becomes intractable, so we limit the application of this scheme to patches with at most 7 quads. This is sufficient to generate patterns with a simple structure but possibly complicated boundary. A similar enumeration scheme generates patches by expanding a whole side of an existing patch at a time, adding all quads beyond a corner-to-corner chain, as in the blue patches of Figure 5. This allows us to generate larger patches with up to 12 quads, possibly with a concave boundary. A further specialization of this approach allows us to extract even larger convex patches with up to 40 quads: every time a reflex corner (i.e., a corner with valence 4 or higher) appears on the boundary of the expanded patch, we expand it further by adding quads in the star of the reflex corner, and we repeat this process until the patch becomes convex.

Convex patches would be sufficient for the entire retopology task, since concave patches can be decomposed into convex parts. However, we found that an expanded database that includes patches with a concave boundary may improve the user experience. Our choice to

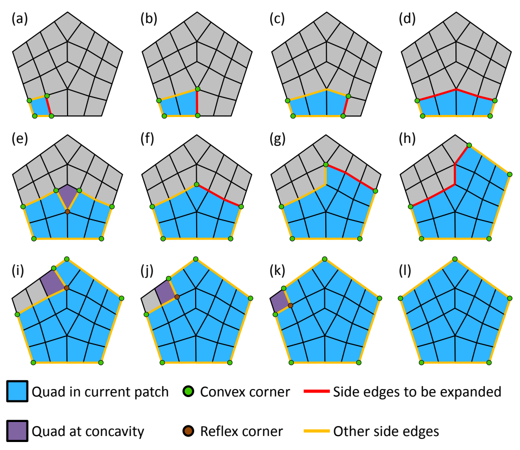

Figure 5: *Whole-side and convex patch expansion along one path in the tree. Two steps of expansion between the first and the second image are omitted. All blue patches are generated during whole-side expansion; while during convex patch expansion, patches (e), (i), (j), (k) that contain reflex vertices do not generate any pattern.*

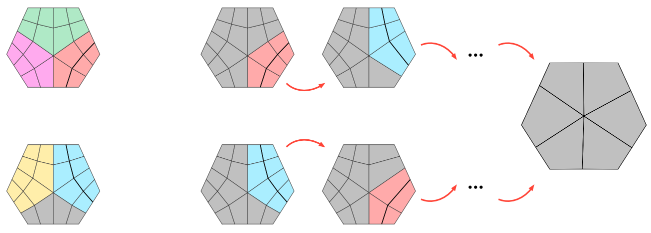

Figure 6: *A patch with bundles of polychords in color; the result of valid polychord collapses is order-independent.*

balance the content of the database between large convex patches and smaller non-convex patches provides a trade-off between flexibility and efficiency of the system. Note that we are able to reproduce exactly the results as in [Takayama et al. 2014], since our algorithm automatically learns the 15 patches that they manually encoded in their method.

Pattern reduction. Patches are reduced to patterns by the polychord collapse operation. A polychord can be collapsed into an edge chain if: it is not self-intersecting, it does not have extraordinary vertices on both edge-chains bounding it at its sides, and the collapse does not change the number of sides of the patch. Polychords are collapsed as long as the conditions above can be met. The result is order-invariant, as shown in the following (see also Figure 6). Note that a polychord can be collapsed only if it is parallel to another polychord with the same structure. Thus, we may organize our patch in bundles of parallel polychords, where each quad belongs to two distinct bundles. If we collapse any polychord in a bundle, we do not prevent subsequent collapse of other polychords in the same bundle, until the bundle reduces to a single polychord. Therefore, the order in which we collapse the polychords belonging to a single bundle is irrelevant. On the other hand, collapsing a given polychord shortens other polychords that it orthogonally intersects by one quad each, and the collapsed polychord necessarily intersects all polychords in the same bundle, thus preserving the structure of the bundle. Therefore, the order in which we process the different bundles does not affect the outcome.

Mesh preprocessing. The majority of extracted patches are equivalent up to polychord collapse; this equivalence is exactly what

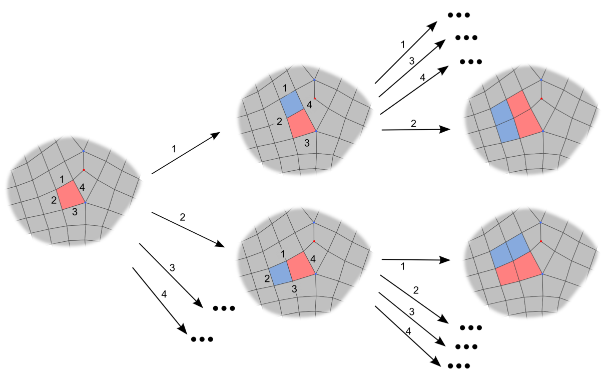

Figure 7: *During patch enumeration, we may encounter the same patch along different paths.*

our patterns factor out. We can thus simplify each mesh, removing all non-intersecting polychords where no singularity collapse occurs. This preprocessing has a twofold benefit. It reduces the search space, thus speeding up the generation process: for instance, on the main character of the BigBuckBunny movie, the number of generated patches decreases from ca. 40 millions to ca. 22 millions, and processing time reduces from ca. 90 minutes to ca. 53 minutes. More importantly, it greatly increases the number of unique patterns learned: for instance, on the same dataset, the number of extracted patterns increases from ca. 2,400 to ca. 17,200. The reason of this latter benefit is that extraordinary vertices become closer to each other after polychord collapse: in spite of using relatively low thresholds for the maximum number of quads in a patch, this fact allows us to capture complex patterns that would require traversing much larger patches on the original mesh.

Managing duplicates. The patch enumeration procedure often generates the same patch multiple times, since the same set of patches can be found starting from different seed faces (Figure 7). To speed up the extraction, we keep track of all the visited patches through hashing: for every visited patch we compute a signature using the MD5 hashing of the sorted string obtained by concatenating the IDs of the boundary vertices in lexicographic order. Checking whether a patch has already been created is fast, and it greatly reduces the learning time, since it allows us to prune the enumeration tree. For instance, in the *BigBuckBunny* dataset, the learning time decreases from almost 7 hours to mere 2 minutes, using the basic algorithm and a limit of 7 quads per patch.

Database organization. The database is partitioned into classes of patterns, each class corresponding to a given number of sides. Within each class, patterns are sorted by the number of extraordinary vertices they contain, and for the same number of extraordinary vertices they are sorted by the frequency with which they were encountered during the learning.

3.3 Encoding patterns

Storing and retrieving millions of patterns requires an efficient encoding scheme to reduce the memory and disk footprints and to support efficient querying. We convert each pattern into a string describing its connectivity. Note that geometric information is not relevant here and is not encoded. The code must be non-ambiguous and unique: if two patterns are identical, we want to efficiently detect this to avoid storing duplicates.

We propose a code inspired by Edgebreaker [Rossignac 1999]: we navigate the pattern and, as we traverse it, we encode the operations in a string. We perform a breadth-first visit of the vertices of the

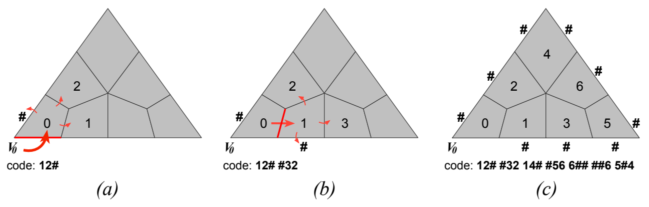

Figure 8: *The enumeration of faces in a template pattern, starting the visit at corner V_0. Special symbol $\#$ is used to represent boundary edges in the code.*

pattern, starting from a corner and always visiting the vertices in a counterclockwise order, starting from a boundary or the parent node (Figure 8(a)). Each face that is encircled during the visit gets an increasing index (Figure 8(b)). The code is formed by traversing the faces in order and appending to the code the index of the neighboring faces, proceeding in counterclockwise order and starting from the parent face (Figure 8(c)). If a face does not have a neighbor, the special character $\sharp$ is used instead. The code obtained with this algorithm is unique up to the choice of the starting corner, and it is lossless: the connectivity is trivial to reconstruct from the code. To make the code invariant to the choice of the starting corner, we compute it starting at all corners and pick the first code according to lexicographic order.

4 Sketching queries

The user sketches curves on the surface to identify boundary constraints (Figure 2(a)), as well as additional edge-flow constraints, explained in the following. The system generates the set of patches that satisfy the given constraints on-the-fly and proposes them to the user in a sorted sequence.

For the simple case of just a boundary constraint without any edge-flow constraints, we perform a query on the database to retrieve all the patterns with the given number of sides (Figure 2(b)), then we test them by solving the ILP and expand them into patches, as shown in Figure 2(c). To provide an interactive response, we process the patterns on all available CPU cores in parallel and visualize the valid patches as soon as they are found. By default, patches come in the same order as they are stored in the database, with the rationale that patterns containing fewer extraordinary vertices and encountered more frequently during learning phase are more likely to be used. The user can set some thresholds, e.g., on the maximum number of extraordinary vertices in a patch, or on the range of valences of extraordinary vertices, which helps to restrict the search and speed up the query. However, it can still happen that hundreds of patches match a given boundary constraint, while the best patches for the user's needs are not top-ranked by this default criterion. We thus allow the user to provide additional edge-flow constraints: each constraint is either a simple stroke touching the patch boundary at its endpoints, or a loop inside the patch. On the one hand this additional feature greatly enhances the performance of the system as it reduces the search space, and on the other hand it allows us to sort the retrieved patterns in a way that provides the most desirable patch among the first few in the sequence.

Edge-flow constraints. For each pattern in the database, we store an additional code that describes the behavior of the edge-flows traversing it. Each boundary edge is labeled with a symbol (Figure 9a) in such a way that two edges have the same symbol if and only if they are crossed by the same polychord. We also add a flag to mark whether or not the pattern contains a loop polychord in its interior. By using this code, we immediately reject patterns that cannot satisfy the sketches, as demonstrated in Figure 9(b).

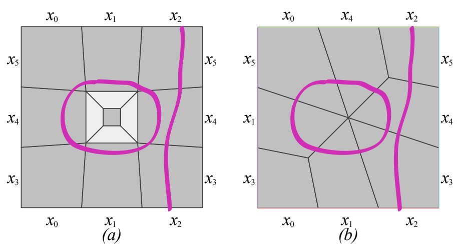

Figure 9: *Boundary labeling of two patterns: both patterns are compatible with the boundary-to-boundary edge-flow constraint, but only pattern (a) is compatible with the loop edge-flow constraint; pattern (b) is rejected.*

Furthermore, for patterns that pass the test, we restrict the solutions of the ILP by additional constrains, asking that a given boundary-to-boundary edge-flow constraint traverses the boundary close to its corresponding polychord. This allows us to improve query efficiency without losing accuracy. Let us consider a boundary-to-boundary edge-flow constraint s that starts from the i_0-th edge of side e_0 to i_1-th edge of another side e_1 of a given boundary constraint; let x_q be an integer variable corresponding to a polychord that is present on both e_0 and e_1 sides of a pattern to be tested against the given constraints. Note that patterns that do not contain such x_q have been previously filtered out, as explained above. Generally speaking, x_q stands for the width of the bundle of polychords that replace the corresponding polychord in the expanded patch. Let us consider the side e_0: the basic constraint for the ILP problem is:

$$x_0 + \ldots + x_q + \ldots + x_k = L_0, \tag{1}$$

where L_0 is the number of edges subdividing e_0 in the boundary constraint polygon. For each variable x_q that is common to both sides e_0, e_1, we set-up a different ILP system. We decompose x_q into three parts: $x_q = x_{q_0} + 1 + x_{q_1}$, where the '1' stands for the polychord that we are considering for alignment with the edge-flow, while x_{q_0} and x_{q_1} are the widths of the two parts of the bundle on its sides. The flow constraint can be expressed by imposing that the left and right parts of the bundle plus the left and right parts of the constraint (1) (e.g. the variables x_j before and after x_q) sum up to the position i_0 indicated by the sketch. In order to collect non perfectly-aligned polychords as well, we add *tolerance* variables $x_{q_{\delta_0}}, x_{q_{\delta_1}}$ to the previous conditions, one for each side e_0, e_1. We implement this by adding, for a side e_0, the following constraints:

$$\begin{aligned}
& x_q = x_{q_0} + 1 + x_{q_1}, \\
& \sum_{j=0}^{q-1} x_j + x_{q_0} - x_{q_{\delta_0}} = i_0, \\
& x_{q_{\delta_0}} + x_{q_1} + \sum_{j=q+1}^{k} x_j = L_0 - i_0 - 1, \\
& x_{q_0} \geq 0, \qquad x_{q_1} \geq 0, \\
& -K \leq x_{q_{\delta_0}} \leq +K.
\end{aligned}$$

This set of constraints is added also for the other side e_1 in a symmetric way, with an inverted left-right ordering to correctly match the x_{q_0}, x_{q_1} variables. Given that $x_{q_{\delta_0}}, x_{q_{\delta_1}}$ specify the tolerance in matching, if we set $K = 0$ then only exact matching is allowed, while for larger values of K flow-polychords can cross the boundary at a distance of at most K edges from the edge-flow constraint. In all our experiments, we use $K = 1$.

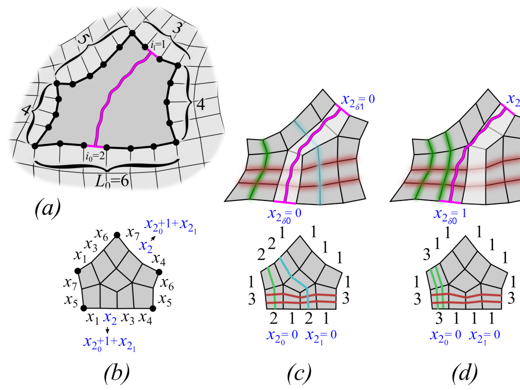

Figure 10: *Sketching an edge-flow constraint allows the user to filter out patterns, as well as to set additional constraints that reduce the set of generated patches to the ones having a polychord exactly starting from and ending to the same edges as the sketch (c) or just close to that edges (d).*

Figure 10 shows a concrete example: part (a) depicts an edge-flow constraint sketched from side e_0 at the bottom of the polygon to side e_1 at its top-right, and we have $L_0 = 6$; part (b) shows a pattern under test and the variable x_2 matches the edge-flow constraint that we are considering. Please note that also x_4 is shared between e_0, e_1 so we will set up a ILP system also for that variable in the corresponding way, for sake of brevity we will show just the x_2 case. The ILP constraint for e_0 is $x_1 + x_2 + x_3 + x_4 = 6$. We expand variable x_2 and add the following constraints for the first edge:

$$x_2 = x_{2_0} + 1 + x_{2_1},$$
$$x_1 + x_{2_0} - x_{2_{\delta_0}} = 2,$$
$$x_{2_{\delta_0}} + x_{2_1} + x_3 + x_4 = 3.$$

The first constraint imposes that the new variables are actually acting as a substitution, while the last two equations reflect the sketch condition and impose that only solutions with a polychord at distance 2 and 4 from the beginning and end of side e_0 are generated, respectively. If we do not allow any tolerance, e.g., imposing that the two variables $x_{2_{\delta_0}}, x_{2_{\delta_1}}$ are exactly 0, we get only the following solution:

$$x_{2_0} = 0, \quad x_{2_{\delta_0}} = 0, \quad x_{2_{\delta_1}} = 0, \quad x_{2_1} = 0,$$

which is shown in Figure 10(c). On the other hand, if we allow some tolerance $K > 0$, we can obtain the solution shown in (d):

$$x_{2_0} = 0, \quad x_{2_{\delta_0}} = 1, \quad x_{2_{\delta_1}} = 0, \quad x_{2_1} = 0.$$

Sorting. We sort the generated patches according to their alignment to the edge-flow constraints (Figure 11). We measure alignment in a patch by mapping each sketch to all possible polychord skeletons, i.e., the set of segments connecting all the midpoints of the transversal edges in a polychord, which connects the same sides of the patch as the sketch. This mapping is computed by parameterizing both polylines using arc-length parameterization. We then measure the integral of the distance between the sketch and all skeletons, and we pick the one with the smallest value. This procedure is repeated for all sketches, and the sum of the distances is used as the sorting criterion. Note that the sorted sequence is updated dynamically as new solutions are provided by the system. The use of sketches to filter and sort patches allows the artist to intuitively and quickly search the database, as we show in our informal user study (Section 5).

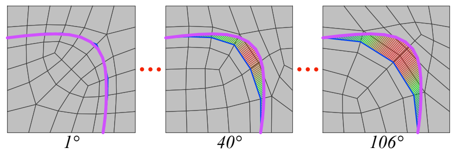

Figure 11: *Sorting solutions according to edge-flow constraints: alignment of polychords to the sketched flow is measured by integral error.*

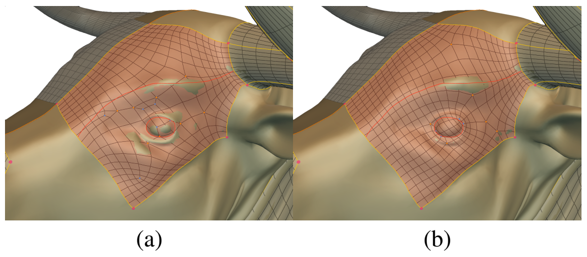

Figure 12: *The first patch found by quad quality ranking (a) has a much lower perceived quality than the first patch found by flow alignment ranking (b), which was the criteria preferred by the artists in our informal user-study.*

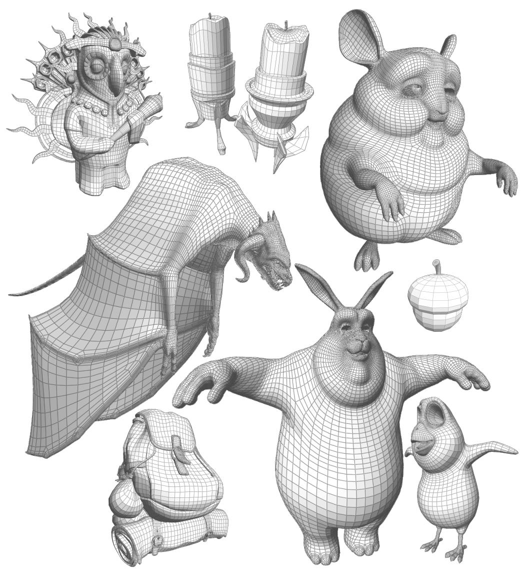

Figure 13: *Some of the meshes used to populate the database.*

Geometric sorting. We also experimented with an alternative ranking solution based on the average quality of the quads, which we measured as the squared sum of the angle deviation from right angles. However, the results are unintuitive and lead to lower quality patches (Figure 12).

Deformation. To further improve the alignment with the sketches, we deform the patch using Laplacian editing [Sorkine et al. 2004]. Note that this has to be performed on all the extracted patches, since it affects the sorting criteria.

5 Results

We implemented our algorithm in C++, and we run learning on a workstation with a 2.7 GHz 12-Core Intel Xeon E5 and 64GB of

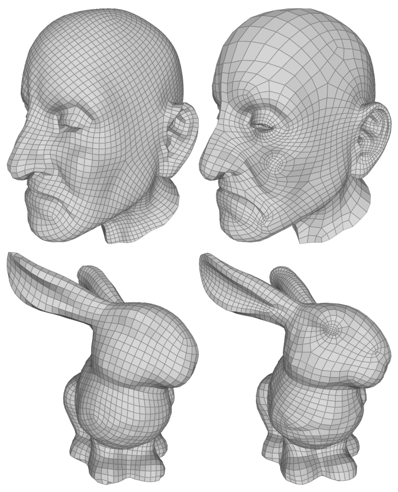

Figure 14: *A comparison between the automatic method proposed by [Bommes et al. 2009] (a) and our approach (b).*

memory. We use the Eigen library for all linear algebra operations, with the exception of the solver for the ILP, which uses [Achterberg 2009]. The timings and statistics for the off-line learning phase are presented in Table 2. We learned 477,567 template patterns from 40 manually modeled meshes, which can tessellate polygons with up to 34 sides (see Table 1). It is interesting to observe that the number of patterns is increasing up to polygons with 10 sides, then it decreases with about the same progression. The rapid growth with the number of sides reveals that complex polygons have more degrees of freedom and thus they can be tessellated with many nontrivial topologies. On the other hand, patches with many corners are only convex, because of the thresholds we put on patch expansion during learning: convex patches with very many sides are rare on manually modeled meshes, being mostly dedicated to polar configurations.

Comparisons with automatic methods. The majority of the automatic quadrangulation algorithms rely on a *guidance field*, which is extracted by a static geometric analysis. These approaches can only capture geometric features, resulting in meshes with a high regularity but missing semantic features such as the cheeks of a human face (Figure 14). Instead, our method allows the artist to manually align quads to semantic features, producing meshes that are ideal for applications in the animation and entertainment industry.

Comparisons with [Takayama et al. 2014]. To evaluate the effectiveness of combining the flow sketch interface with the large database of learned patches, we performed four tests mixing the individual contributions of this paper with the original approach proposed by [Takayama et al. 2014] (Figure 15):

- Figure 15 (a): the patch is tessellated with [Takayama et al. 2014]. When possible, the user changed manually the position of the singularities to improve the quality. This is a time consuming and difficult to master approach. In certain cases (e.g. the eye of the bunny), this manual operation is still not sufficient to produce the desired edge flow due to the minimal number of singularities introduced in each patch.
- Figure 15 (b): the patch layout is refined into smaller patches which are then tessellated using [Takayama et al. 2014]. Since the final tessellation is highly influenced by the number of subdivisions, the user needs to tune the boundary constraints to obtain a high-quality tessellation, leading to longer editing sessions.

corners	patterns	corners	patterns	corners	patterns
0	88	12	51,447	24	516
1	44	13	40,189	25	135
2	649	14	30,816	26	17
3	1,650	15	21,578	27	4
4	5,537	16	14,638	28	1
5	11,201	17	9,318	29	1
6	20,553	18	6,182	30	15
7	31,747	19	3,501	31	1
8	45,692	20	2,412	32	2
9	56,144	21	1,179	33	1
10	61,023	22	1,439	34	1
11	59,257	23	589		

Table 1: *Number of patterns stored in the database for a given number of corners.*

Mesh	Time	Mesh	Time
rabbit	00:50	alekkosinski	00:12
f.squirrel	01:05	ishtarguard	00:13
chinchilla	00:54	zoyd	00:04
bird	00:13	l.aztec	00:07
i.femme	12:30	bending tree	00:10
l.cycles	08:07	dragon	00:21
religious	07:55	butterfly	00:03
i.matron	00:23	rock	00:02
dimetrii	00:33	arrow	00:02
taylee	01:21	ground stick	00:18

Table 2: *Timings (hh:mm) of the enumeration process.*

- Figure 15 (c): we use a database composed of the 15 template patches proposed by [Takayama et al. 2014]. The patches are then selected using our novel sketch interface (Section 4). Since the database is small, the query usually returns the same result of [Takayama et al. 2014], thus not improving the edge-flow.
- Figure 15 (d): The sketch interface is combined with a large patch database, producing a quad mesh with the desired edge-flow.

An additional comparison with the approach of [Takayama et al. 2014] is provided in our informal user-study.

For the sake of comparison, we remind that [Takayama et al. 2014] can fill polygons up to 6 sides by using just 15 predefined patterns. Such patterns are sufficient to cover all possible boundary constraints, but they do not provide a sufficient variety of expanded patches to support effectively arbitrary edge-flow constraints.

Informal User-Study. We integrated our system into the publicly available sketch-based modeling user interface proposed by [Takayama et al. 2013] and performed an informal user study with three professional artists: two from a game company and one from a CGI animation studio. Each one of them retopologized one of the three models shown in Figure 16, after a short training session of 45 minutes, using both the method proposed in [Takayama et al. 2014] and our method.

The limited vocabulary of patches of [Takayama et al. 2014] forced the artists to draw by hand finer quad layouts (yellow lines) in search of the desired flow, while our complex patches and edge-flow constraints helped them find better solutions with larger patches in

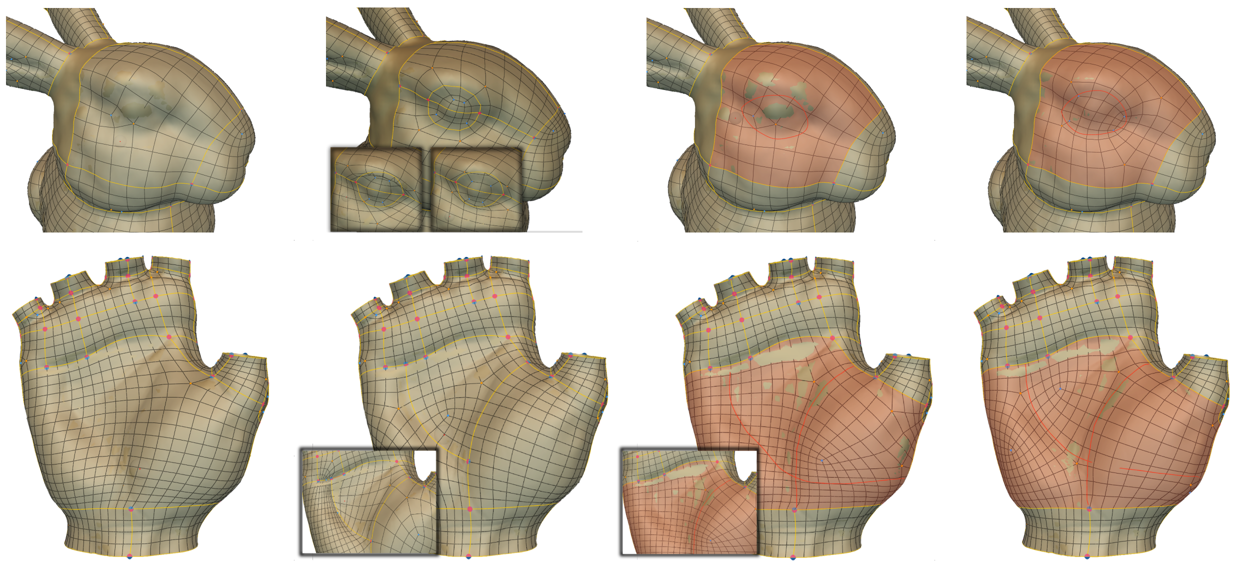

Figure 15: *A comparison between different approaches using two patches of the bunny and the hand tessellation: (a) The approach proposed by [Takayama et al. 2014]; (b) The same approach with a larger set of smaller patches; (c) User sketches are used to retrieve a tessellation using a database composed by the 15 template patches of [Takayama et al. 2014]; (d) The proposed approach.*

		Previous			New		
Artist	Dataset	Time	Patch	Quad	Time	Patch	Quad
1	Torso	80	72	3566	45	12	4522
2	Hand	80	73	1973	60	23	1755
3	Ear	80	143	758	50	25	784

Table 3: *Statistics from the informal user-study.*

shorter time. As shown in Table 3, the time needed to accomplish the task was 25% to 45% lower with our tool. Obviously, there can have been some learning effect during training, which influenced the artists. However, while a strictly objective study is impractical, this simple informal study suggests that our method has potential to speed-up productivity.

The feedback was positive, despite the short training period and a GUI not as polished as the commercially available tools artists are used to work with. We report a few excerpts from the artists' feedback, which we include in their original form in the additional material. Note that artists refer both to our contribution and to the original software of [Takayama et al. 2013; Takayama et al. 2014] as SketchRetopo in their comments, since our contribution was added as an additional set of tools in a single UI.

Artist 1 (from a game company) pointed out the limitations of fully automatic remeshing tools: "*Lately Zbrush added the command "Zremesh" that creates an automatic retopology of the mesh, but this function, that emerged as an incredible leap forward and a help to reduce production time, isn't correct enough and requires a lot of cleaning and optimization from the artist. I started to use SketchRetopo to retopologize some test mesh and I was immediately impressed about it:* [...] *the Query Database is the most impressive part of the SketchRetopo program. If just using big patches reduces retopology time, now by using this function and creating patches even bigger, then defining/suggesting the principal lines of the final wireframe and leaving the program to calculate the possible solutions, now the time of production for a character can easily move from one or two days to few hours. It is obvious that at the end there will be some hand work, but this is just for refining/cleaning few parts, but the gain is undeniable.*".

Artist 2 (from a CGI company) appreciated the edge-flow sketching tool "*The new feature with 'S' hotkey that suggests many solutions computed by the system is also really innovative. The idea of proposing many topologies from which the user can choose is really novel.*" and in particular "*I also like the ability to create a clean edge loop by drawing a circle in that mode.*".

Artist 3 (from a game company) raised concerns for very low resolution retopology tasks: "*With the need to create meshes with relatively low triangle count, I'm not sure how this tool fits into a game development pipeline. Maybe with more usage and a greater understanding of how things work I could figure out a way. I found it difficult to work on the ear to start off. It seems the tool would be great with the bigger forms of the body (torso, waist, arms and legs).*".

Sorting criteria. As we already remarked, if just boundary conditions are used to select patches, the many possible results are presented to the user in a sequence sorted by number of extraordinary vertices inside the patch and pattern frequency during learning. This topological/statistical criterion is sufficient to select simple patches with a quite regular internal structure, as depicted in the top row of Figure 17. However, if the user wishes to select a more complex internal structure, this criterion may rank the most appropriate solutions in the back positions of a long list (29,389 solutions for the top row). Adding one stroke, as depicted in the middle row of Figure 17 filters the space of solutions (only 1,310 solutions are accepted in this case) and changes the sorting criterion by considering the compatibility to the edge-flow constraint. Adding another stroke, as depicted in the bottom row of Figure 17 filters the space of solutions further to just 55 patches and updates the sorted sequence in such a way that patches in the first few positions are likely to satisfy user needs. We have experienced that with more than two strokes sometimes the search is overconstrained and may fail to find

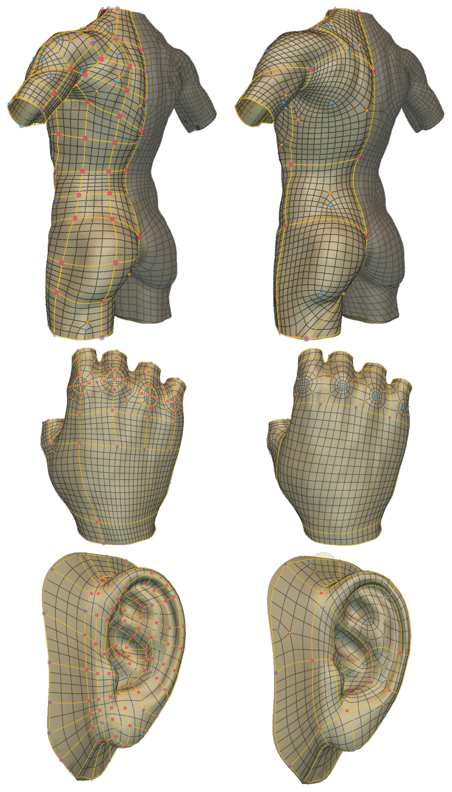

Figure 16: *Models used for the user study. Retopology with [Takayama et al. 2014] (left) forces the artist to draw by hand a much finer patch layout (in yellow) than our approach (right): our edge-flow constraints allow the user to fill large patches with complex structures following shape features.*

a reasonable solution. In this case, the user may better subdivide the patch into simpler patches.

Loop edge-flows. In some cases, the user may need to specify patches that contain edge loops in their interior. This is very useful to follow features such us eyes, knuckles and sphincter muscles in general. Figure 18 shows some examples of results obtained by specifying just a single loop constraint.

Concave vs. convex patches. Topologically convex patches are sufficient to deal with most situations and they are used in the great majority of our results. However, we also learned a moderate number of concave patterns and we experimented they are sufficient to cover quite a few non-trivial patches. Figure 19 shows an example of concave patch that we could use to cover a rather extended and morphologically complex portion of surface.

Polygons with many sides. Our system is able to tessellate polygons with a high number of sides using a nontrivial quad mesh connectivity. We demonstrate this feature in Figure 20 by filling a patch with eight sides that connects all tentacles of the *Octopus*. Note how the sphincter corresponding to the mouth of the octopus is captured by sketching a loop flow around it.

6 Limitations and concluding remarks

Designing a coarse quad layout is an application-specific task, which is difficult to solve without any user input. We proposed an interactive solution, based on a preprocessing step that learns high-quality

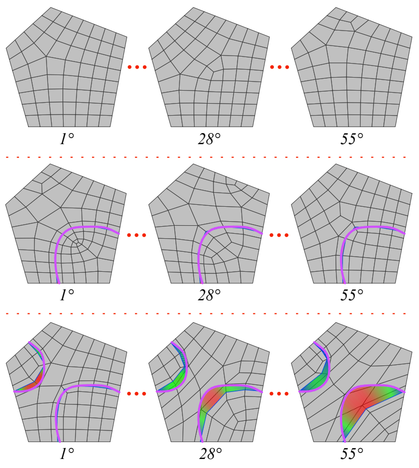

Figure 17: *Different sorting criteria for solutions with the same boundary constraint: the top row is sorted by number of extraordinary vertices and pattern frequency; the middle row is sorted by using an additional edge-flow; and the bottom row is sorted by using two edge-flows.*

quadrangulation patterns from existing quad meshes. We demonstrated that our data-driven approach provides a richer set of solutions compared to previous methods, reducing the amount of work needed to retopologize complex models.

Our current database has 477,567 patterns learned from 40 meshes. While being sufficient to demonstrate the effectiveness of our approach, it is not large enough to cover all the possible configurations for patches with more than six sides. In this sense, our approach does not warrant a valid solution for any possible boundary constraint, even on the limited range of polygon sides that we store in our database. In particular, our database contains many convex patterns, but a relatively small number of patterns with concavities, hence a query made with a boundary constraint with many reflex vertices might easily not find an answer. However, in our interactive

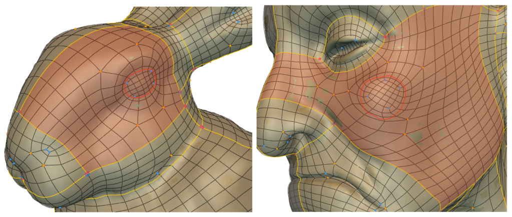

Figure 18: *Circular sketches are used to find patches with loop edge-flows in their interior, like for the eye of the bunny to the left, or the cheekbone of the old man to the right.*

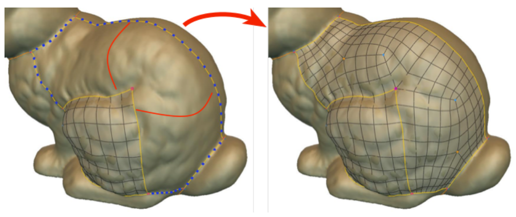

Figure 19: *A concave patch is used to cover a large patch on the back of the Stanford bunny.*

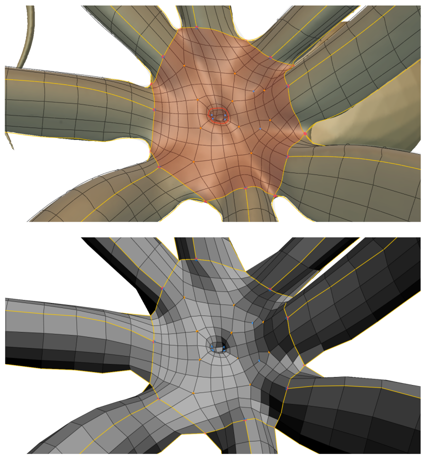

Figure 20: *An example of successful tessellation of patches with 8 corners at the bottom of the* Octopus *model.*

sessions we hardly found situations in which our database fails to provide a satisfactory answer. We also tested our system with 10,000 random queries for convex patches with 2 to 20 sides: the tool has failed only in two cases. In these cases, the user can try splitting a complicate patch into a set of simpler patches, thus generating a finer quad layout; alternatively, the system could be easily complemented with traditional tools for manual design. It would also be interesting to allow the system to dynamically learn new patterns as an artist uses the tool. In fact, in spite of using known patterns inside patches, new patterns may arise, which span across different patches of the quad layout.

We will release the source code of our application in the hope that users would share the patch layouts extracted from their meshes with us, helping in our efforts to create a large patch database, which we plan to release publicly. Having access to a database with millions of patches will open many new opportunities, improving the effectiveness of our system. Caching strategies and advanced querying system will likely be necessary to interactively explore the immense solution space. More sophisticated algorithms for connectivity compression, like the extended version of Edgebreaker [King et al. 2000] for quad meshes, may be used to reduce the overall memory occupancy keeping the efficiency of the querying system.

7 Acknowledgements

This work was supported in part by the ERC grant iModel (StG-2012-306877), by EU FP7 project ICT FET Harvest4D (www.harvest4d.org, G.A. 323567) and a gift from Adobe.

References

ACHTERBERG, T. 2009. SCIP: Solving constraint integer programs. *Mathematical Programming Computation 1*, 1, 1–41.

AUTODESK, 2007. Mudbox. `http://www.autodesk.com`.

BLENDER FOUND., 2008. Big buck bunny. `http://peach.blender.org`.

BLENDER FOUND., 2010. Sintel. `http://durian.blender.org`.

BOMMES, D., ZIMMER, H., AND KOBBELT, L. 2009. Mixed-integer quadrangulation. *ACM Trans. Graph. 28*, 3.

BOMMES, D., LEMPFER, T., AND KOBBELT, L. 2011. Global structure optimization of quadrilateral meshes. *Comput. Graph. Forum 30*, 2.

BOMMES, D., CAMPEN, M., EBKE, H.-C., ALLIEZ, P., AND KOBBELT, L. 2013. Integer-grid maps for reliable quad meshing. *ACM Trans. Graph. 32*, 4.

BOMMES, D., LEVY, B., PIETRONI, N., PUPPO, E., SILVA, C., TARINI, M., AND ZORIN, D. 2013. Quad-mesh generation and processing: A survey. *Comput. Graph. Forum 32*, 6.

CAMPEN, M., AND KOBBELT, L. 2014. Dual strip weaving: Interactive design of quad layouts using elastica strips. *ACM Trans. Graph. 33*, 6, 183:1–183:10.

CAMPEN, M., BOMMES, D., AND KOBBELT, L. 2012. Dual loops meshing: quality quad layouts on manifolds. *ACM Trans. Graph. 31*, 4.

KING, D., ROSSIGNAC, J., AND SZYMCZAK, A. 2000. Connectivity compression for irregular quadrilateral meshes. *CoRR cs.GR/0005005*.

MARCIAS, G., PIETRONI, N., PANOZZO, D., PUPPO, E., AND SORKINE, O. 2013. Animation-aware quadrangulation. *Computer Graphics Forum SGP 2013*.

NASRI, A., SABIN, M., AND YASSEEN, Z. 2009. Filling N-sided regions by quad meshes for subdivision surfaces. *Comput. Graph. Forum 28*, 6.

PENG, C.-H., ZHANG, E., KOBAYASHI, Y., AND WONKA, P. 2011. Connectivity editing for quadrilateral meshes. *ACM Trans. Graph. 30*, 6.

PENG, C.-H., BARTON, M., JIANG, C., AND WONKA, P. 2014. Exploring quadrangulations. *ACM Trans. Graph. 33*, 1.

PILGWAY, 2013. 3D-Coat 3.0. `http://3d-coat.com/`.

PIXOLOGIC, 2013. ZBrush 4.4. `http://pixologic.com`.

ROSSIGNAC, J. 1999. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Trans. Vis. Comput. Graph. 5*, 1, 47–61.

SCHAEFER, S., WARREN, J., AND ZORIN, D. 2004. Lofting curve networks using subdivision surfaces. In *Proc. SGP*.

SORKINE, O., COHEN-OR, D., LIPMAN, Y., ALEXA, M., RÖSSL, C., AND SEIDEL, H.-P. 2004. Laplacian surface editing. In *Proc. Eurographics Symposium on Geometry Processing*, 179–188.

TAKAYAMA, K., PANOZZO, D., SORKINE-HORNUNG, A., AND SORKINE-HORNUNG, O. 2013. Sketch-based generation and editing of quad meshes. *ACM Trans. Graph. 32*, 4, 97:1–97:8.

TAKAYAMA, K., PANOZZO, D., AND SORKINE-HORNUNG, O. 2014. Pattern-based quadrangulation for N-sided patches. *Computer Graphics Forum 33*, 5, 177–184.

TARINI, M., PUPPO, E., PANOZZO, D., PIETRONI, N., AND CIGNONI, P. 2011. Simple quad domains for field aligned mesh parametrization. *ACM Trans. Graph. 30*, 6.

TIERNY, J., DANIELS, II, J., NONATO, L. G., PASCUCCI, V., AND SILVA, C. T. 2011. Inspired quadrangulation. *Computer Aided Design 43*, 11.

YASSEEN, Z., NASRI, A., BOUKARAM, W., VOLINO, P., AND MAGNENAT-THALMANN, N. 2013. Sketch-based garment design with quad meshes. *Computer Aided Design 45*, 2.