PushPull++

Markus Lipp [*]
Esri R&D Center Zurich

Peter Wonka [†]
KAUST

Pascal Müller [‡]
Esri R&D Center Zurich

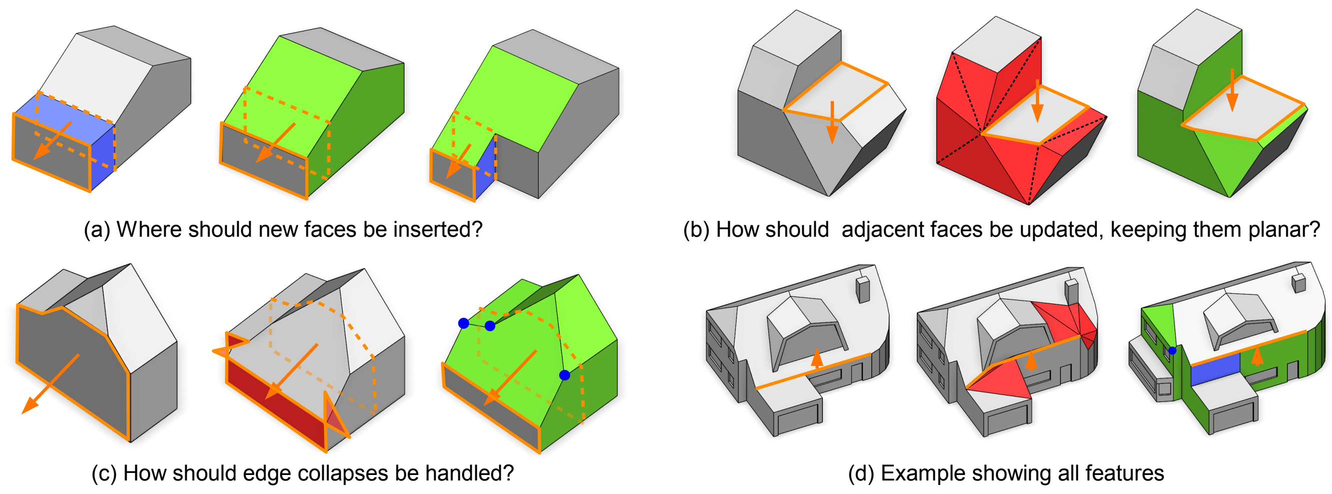

(a) Where should new faces be inserted?
(b) How should adjacent faces be updated, keeping them planar?
(c) How should edge collapses be handled?
(d) Example showing all features

Figure 1: *There are multiple challenges when a PushPull operation is performed on a face or edge. Case (a): New faces can either be inserted for all edges (left) or not at all by adjusting adjacent faces (middle). In addition, our solution can adaptively add new faces where needed (right). New faces are blue and modified adjacent faces are green. In (b-d), the left figure is the input, the middle is the degenerate result by previous approaches, and the right is our result. Non-planar or self-intersecting faces are red and edge collapses are blue dots.*

Abstract

PushPull tools are implemented in most commercial 3D modeling suites. Their purpose is to intuitively transform a face, edge, or vertex, and then to adapt the polygonal mesh locally. However, previous approaches have limitations: Some allow adjustments only when adjacent faces are orthogonal; others support slanted surfaces but never create new details. Moreover, self-intersections and edge-collapses during editing are either ignored or work only partially for solid geometry. To overcome these limitations, we introduce the PushPull++ tool for rapid polygonal modeling. In our solution, we contribute novel methods for adaptive face insertion, adjacent face updates, edge collapse handling, and an intuitive user interface that automatically proposes useful drag directions. We show that PushPull++ reduces the complexity of common modeling tasks by up to an order of magnitude when compared with existing tools.

Keywords: Polygonal modeling, local mesh editing

Links: DL PDF

[*]mlipp@esri.com
[†]pwonka@gmail.com
[‡]pascal.mueller@esri.com

1 Introduction

3D models of plane-dominant man-made objects are widely used in areas such as architecture, engineering, and design. Therefore, tools enabling intuitive creation and editing of such models are vital. One popular and widespread class of tools is what we call the PushPull techniques. Variants of these are implemented in most commercial modeling suites such as AutoCad [Autodesk 2014a] or Sketchup [Trimble 2013]. Their idea is for the user to transform a face or edge and then automatically adapt the model locally, possibly inserting new faces and modifying adjacent ones. We identified three main challenges for mesh adaption, as explained below.

The first challenge is to decide when new faces should be inserted. On the one hand, inserting new faces is important to add details. On the other hand, the user might want to adjust the existing adjacent faces without inserting new ones. However, sometimes adjusting is not possible, for example when the adjacent face is parallel to the transformed one. Therefore it is tricky to decide when to add details and when to adjust, especially with slanted surfaces as shown in Figure 1(a). The second challenge is how vertices and adjacent faces should be updated. Moving one face affects all the neighboring faces. Care must be taken to keep them planar. This is particularly challenging when the valence of a moved vertex is greater than three, which requires calculating and adding new vertices to neighboring faces, as shown in Figure 1(b). The third challenge is how edge or face collapses should be handled. Moving a face can cause adjacent edges or faces to collapse. Such collapses must be handled to maintain a valid mesh, as seen in Figure 1(c).

PushPull++ is a novel tool for the rapid modeling of polygonal objects. It is the first method that provides solutions for all three main challenges. Essentially it is a generalization of common PushPull practices, supporting both existing and new editing operations. This is achieved using the following contributions:

- An *Adaptive Face Insertion* method that decides where to insert new faces based on angular thresholds between planes (Section 3.1).
- A *Generalized Mesh Update Algorithm* to calculate new vertex positions, handle adjacent faces, and efficiently update the local mesh topology (Section 3.2).
- A *Stepwise Modification* approach to handle edge collapses without resorting to constructive solid geometry (Section 3.3).
- The *PushPull++ User Interface* that automatically proposes useful drag directions for an intuitive user experience and rapid modeling operations (Section 4).

The PushPull++ tool requires up to an order of magnitude fewer mouse clicks compared with state-of-the-art modeling suites, as shown in our results (Section 5).

2 Related Work

In our review, we focus on modeling tools for polygonal mesh modeling of man-made objects. We refer the reader to the excellent work by Botsch et al. [2007] for a general overview of polygonal mesh processing.

PushPull Techniques One of the simplest approaches to handle a transformed face is *extrusion* [Baumgart 1974], by inserting a new face for every edge. Extensions include scaling, direction and offset parameters [Havemann and Fellner 2005] as implemented in, e.g., Maya [Autodesk 2014b], and sketched extrusion paths [Zeleznik et al. 1996]. SketchUp [Schell et al. 2003; Trimble 2013] modifies faces orthogonal to the move direction instead of inserting new ones. This enables dimension adjustments of orthogonal meshes, but it does not support slanted ones. Kelly and Wonka [2011] introduce procedural extrusions based on weighted straight skeletons, essentially controlling the offsets of the extrusion along a path.

AutoCAD [Kripac 2005; Autodesk 2014a] implements a *constrained move*: Instead of inserting new faces, the vertices of a moved face are constrained to lie on the adjacent faces, ensuring planarity. This enables the adjustment of slanted surfaces, but no new details are added because no new faces are created. By employing boolean operations on solid objects [Shapiro 2002], AutoCAD supports valence changes and edge collapses to some degree: It performs a boolean subtraction of the volume between the original and the transformed face from the model. However, this only works for water-tight meshes and can fail when the transformed face has self-intersections, as seen in the middle of Figure 1(c), for example.

We consider the editing operations in AutoCAD, Maya and SketchUp as the closest related work and we give extensive comparisons in the results section.

Global Approaches There has been a lot of recent work in constraint-based modeling for rapid polygonal shape manipulation: Cabral et al. [2009] introduce a method to modify lengths of edges, while constraining angles. Kraevoy et al. [2008] show how global deformations can be distributed non-homogeneously by protecting vulnerable regions. Support for nonlinear constraints was introduced by Habbecke and Kobbelt [2012]. An internal structure of the model called iWIRES was used by Gal et al. [2009] for rapid structure-preserving deformations. Zheng et al. [2011] used shape analysis to find a hierarchy of controllers that also enabled structure-preserving editing. Bouaziz et al. [2012] used a single energy formulation for interactive shape exploration. Methods exploiting patterns in the shape were proposed by Bokeloh et al. [2011; 2012]. They detected repeating patterns and removed or inserted elements during deformation. Multiple methods employing global variational optimization on the surface were introduced for mesh deformation [Botsch and Sorkine 2008]. These methods are best suited for organic objects. Multi-resolution editing approaches on a coarse to fine-grained scale are also applicable [Zorin et al. 1997; Kobbelt et al. 1998]. Adding holes to non-solid meshes is possible by using global collision detection [Bernstein and Wojtan 2013]. Sasaki et al. [2013] used unconnected planes as the model representation, and employed global heuristics to determine connectivity and bounded faces. As a result, even a slight change of planes can cause a large difference in the output mesh.

Planar Quad (PQ) Meshes Recent work shows how to explore shape spaces [Yang et al. 2011] of PQ meshes, and how to model them using affine maps [Vaxman 2012] or with varying levels of locality [Deng et al. 2013]. Those methods are restricted to quad or circular meshes, in contrast to our method in which we use arbitrary planar polygons.

3 Plane-Driven Face Modification

In this section, we introduce the mesh modification method that underlies our novel modeling tool, PushPull++ . This robust method operates on arbitrary polygonal meshes; i.e., in contrast to previous work, no watertight meshes nor constructive solid geometry representations with boolean operations are required. In the following, we first explain the core concept and how faces can be adaptively inserted. Second, we present the local mesh update algorithm that computes the new geometry resulting from a push- or pull-operation. Third, we show how self-intersections are prevented, and finally, we describe how the method can be extended to work with simultaneous face modifications.

Our method applies the most widely used polygonal mesh representation, the so-called face-vertex meshes, consisting of a simple list of vertices and a set of faces that point to their vertices [Foley 1996]. Hence, our input mesh M is represented by vertices $v \in V$, and planar faces $f \in F$ where a face f is defined as a list of counterclockwise oriented vertices $\{v_i, v_j, \ldots, v_k\}$, and a face normal $f.\hat{n}$. For each two connected vertices in f, we define an edge e as the pair $\{v_i, v_j\}$. The mesh can be of arbitrary topology, e.g., it can contain boundaries or non-manifold edges.

3.1 Adaptive Face Insertion

The core concept of our face modification method is to transform a selected face f_m onto a user-defined target plane p_m as depicted in Figure 2. The first constraint is that all vertices of f_m have to lie on p_m. The second constraint is that these vertices need to be coplanar to the corresponding faces that are edge-adjacent to f_m. Thus, in the trivial case, a new vertex position is computed by interesecting three planes, i.e., p_m with the two adjacent face planes.

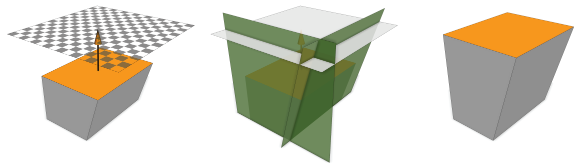


Figure 2: *The core concept of our mesh modification method. Left: The user selects a face f_m and defines a target plane p_m. Middle: New vertex positions are computed by intersecting the adjacent face planes with p_m. Right: The resulting mesh with modified face f_m.*

As stated above, the face planes that are edge-adjacent to f_m are used to compute the new vertex positions. However, this might lead to degenerate cases, e.g., when a face plane is parallel to p_m, the planes cannot be intersected. To solve this problem, we present a novel method that adaptively determines - using an angle threshold θ - if the planes of the adjacent faces can be used (green faces in Figure 3) or if new faces have to be inserted (blue faces). In the latter case, the user-defined direction vector $\vec{d}$ specifies the orientation of the new face. Note that only zero-area faces are inserted at this stage, and the computation of the new vertices is described in Section 3.2.

Algorithm 1 explains the face insertion method in detail. In Line 1 and following, we loop over each edge of f_m to get its adjacent face f_{adj} and then check if a new face needs to be inserted. The latter is the case if no adjacent face exists (mesh boundary) or if f_{adj} is approximately parallel (within the threshold of $90^\circ - \theta$) to p_m.

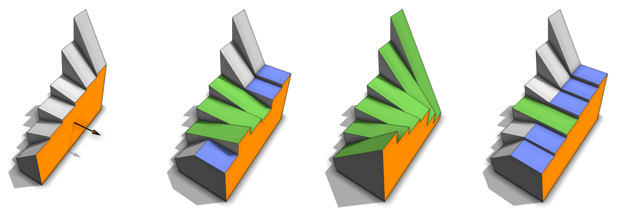
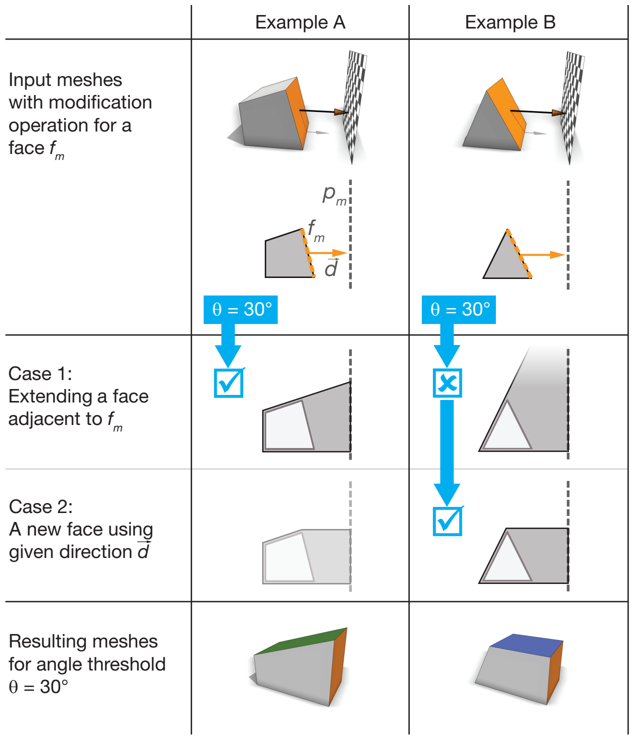


Figure 3: *Left: An initial mesh with face modification. Middle left: The resulting mesh with 30° as the angle threshold θ, which determines if existing planes can be used (green) or if new faces have to be inserted (blue). Middle right: The resulting mesh with $\theta = 70^\circ$. Right: In case θ is 0°, a traditional extrude operation results.*


Example A
Example B
Input meshes with modification operation for a face f_m
p_m
f_m
$\vec{d}$
θ = 30°
θ = 30°
Case 1: Extending a face adjacent to f_m
Case 2: A new face using given direction $\vec{d}$
Resulting meshes for angle threshold θ = 30°
Figure 4: *Two example meshes with their face modification operation depicted in the top row. The middle part presents the face insertion strategies, i.e., Example B requires the insertion of a new face. In the bottom row, the resulting meshes are shown.*

Example B of Figure 4 shows such a case where the angle between face plane and p_m is too small. Note that (1) the function $\angle(\vec{v_1}, \vec{v_2})$ always returns values smaller than or equal to 90°; i.e., if $\vec{v_1} \cdot \vec{v_2}$ is negative, then the complementary angle is returned; (2) planes are specified in the Hessian normal form, i.e., $\hat{n}$ is the normal of a plane and d is its distance to the origin; and (3) in the case of a non-manifold mesh, we might have to handle ambiguities when determining f_{adj}. Hence, to choose one of the multiple possibilities, we pick the face most orthogonal to f_m.

In Lines 4 and 5, we insert the new face consisting of the two vertices of edge e only (later in Section 3.2, more vertices are added) and set its normal to the cross product of the edge's direction and the user-defined direction vector $\vec{d}$.

3.2 Mesh Update Algorithm

In the following, we introduce an efficient algorithm to compute the new geometry resulting from a face modification. The mesh is updated locally by computing new vertex positions for f_m and updating the topology in its 1-neighborhood, i.e., its edge- and vertex-adjacent faces. The novel method is presented in Algorithm 2 and Examples C-G in Figure 5 illustrate it.

Example C shows the simple modification case where the new position for a vertex v is computed by intersecting p_m with its two adjacent faces. In more complex cases such as Example D, v has to be replaced with multiple vertices. Therefore we create the set F_{fan} which contains the faces around v (sorted counterclockwise and without f_m). The new vertices can then be computed by intersecting p_m with the face planes of each subsequent tuple in F_{fan}.

Algorithm 1 insertFaces($f_m, p_m, \vec{d}, \theta$)

```
1:  for each edge $e$ in $f_m$ do
2:      $f_{adj}$ = get face adjacent to $f_m$ on $e$
3:      if $\nexists f_{adj}$ or $\angle(p_m.\hat{n}, f_{adj}.\hat{n}) < 90^\circ - \theta$ then
4:          insert zero face $f_{new}$ at $e$ in $F$
5:          $f_{new}.\hat{n} = normalize(\vec{e} \times \vec{d})$
```

Algorithm 2 updateMesh(f_m, p_m)

```
1:  for each vertex $v$ in $f_m$ do
2:      # determine affected faces
3:      $F_{fan}$ = get fan of faces ccw around $v$ (without $f_m$)
4:      $F_{aff} = \emptyset$
5:      for each face $f_i$ in $F_{fan}$ do
6:          if $i$=0 || $i$=last || $f_i$ intersects $p_m$ then add $f_i$ to $F_{aff}$
7:      # compute new vertices
8:      $V_{new} = \emptyset$
9:      for each except last face $f_i$ in $F_{aff}$ do
10:         add $intersect(p_m, p(f_i), p(f_{i+1}))$ to $V$ and $V_{new}$
11:     # update vertex indices of faces
12:     replace $v$ with $V_{new}$ in $f_m$
13:     for each face $f_i$ in $F_{aff}$ do
14:         if $i$=0 then $V_{sel}$ = first vertex of $V_{new}$
15:         else if $i$=last then $V_{sel}$ = last vertex of $V_{new}$
16:         else $V_{sel}$ = vertices $v_{i-1}$ and $v_i$ of $V_{new}$
17:         $f_n$ = get next face to $f_i$ in $F_{fan}$ (modulo)
18:         $f_p$ = get previous face to $f_i$ in $F_{fan}$ (modulo)
19:         if $f_n \notin F_{aff}$ then insert $V_{sel}$ after $v$ in $f_i$
20:         else if $f_p \notin F_{aff}$ then insert $V_{sel}$ before $v$ in $f_i$
21:         else replace $v$ with $V_{sel}$ in $f_i$
```

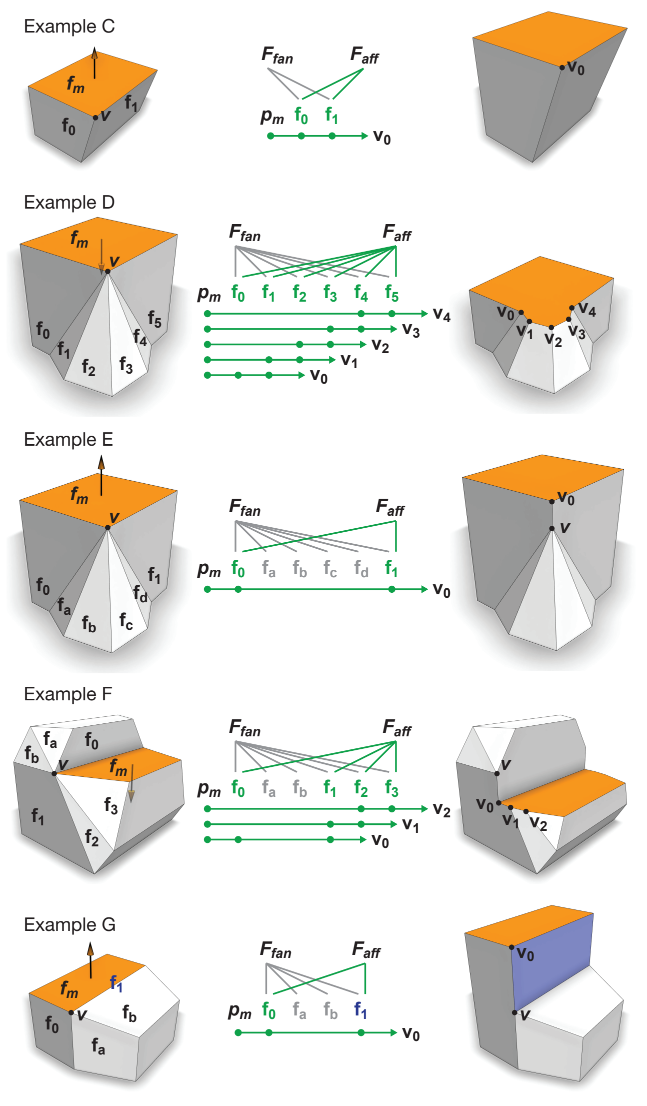

Example C
f_m
f_0
f_1
v
F_{fan}
F_{aff}
p_m f_0 f_1
v_0
Example D
p_m f_0 f_1 f_2 f_3 f_4 f_5
v_0 v_1 v_2 v_3 v_4
Example E
p_m f_0 f_a f_b f_c f_d f_1
Example F
p_m f_0 f_a f_b f_1 f_2 f_3
Example G
p_m f_0 f_a f_b f_1

Figure 5: *On the left, the meshes before the modification operation are shown. In the middle, the faces around v are listed and the dots highlight which face planes are used for the intersection to compute the new vertices v_i; i.e., an arrow illustrates how three planes create one vertex. On the right, the resulting meshes are shown. Note that f_1 in Example G is a zero-area face inserted by Algorithm 1.*

However, as shown in Examples E-G, not all faces in F_{fan} might be affected by the modification. Thus, to compute the new vertices, we only use the subset $F_{aff} \subseteq F_{fan}$ consisting only of the two faces edge-adjacent to f_m and faces that are intersected by p_m. The edge-adjacent faces are always affected because they share vertices with the modified face.

The new vertices V_{new} are computed in Line 10 of Algorithm 2 where the intersection of p_m with two face planes takes place. Note that (1) the function $p(f)$ returns the plane coplanar with face f, and (2) the function $intersect()$ also handles coplanar input planes by replacing one of them with an orthogonal plane through v.

The modified face f_m is then updated by replacing v with all new vertices in V_{new} (Line 12). Next, the affected faces around v are updated. Therefore, in Lines 14-16, we first determine which of the new vertices belong to an affected face and store them in V_{sel}. As shown in Example D, the faces that are edge-adjacent to f_m (i.e., the first and last face in F_{aff}) need to update only one vertex while the other affected faces require two new vertices.

In simple cases such as Examples C or D, the old vertex v in an affected face can now be replaced with the vertices in V_{sel} (Line 21). However, if not all faces in F_{fan} are affected, then the old vertex v cannot be removed because it is still used by the non-affected faces. Thus, we need to detect the border between affected and non-affected faces and ensure that the border edges are preserved. The face f_0 in Example E shows such a case where its next neighbor f_a is not in F_{aff}. Hence, to update f_0, we do not replace v with the new vertex, but instead keep v and insert the new vertex *after* v in the face's counterclockwise ordered list of vertices (Line 19). Similarly, f_1 needs to keep v since it is adjacent to the non-affected face f_d. But because it is on the other side of the border, the new vertex needs to be inserted *before* v (Line 20). The same case is shown in face f_1 of Example F: the previous face f_b is non-affected and therefore f_1 keeps v and inserts its new vertices V_{sel} before v (V_{sel} consists of two vertices since f_1 is not edge-adjacent to f_m; see previous paragraph). Note that this approach works also in the case of zero-faces inserted by Algorithm 1, i.e., because a new vertex is added before/after both existing vertices, the resulting face has valence 4 (see Example G).

Texture coordinates are updated as follows: Every vertex v in every modified face f is projected onto the original plane of f, resulting in v_p. Then, we use mean-value coordinates of v_p to extrapolate the texture coordinate. For new faces, the coordinates of the adjacent face are copied and scaled to the same relative size.

3.3 Intersection Handling with Stepwise Modification

A face modification can lead to self-intersecting meshes as shown in the second row of Figure 6. To prevent these local self-intersections, we introduce a plane sweep approach in which we iteratively identify the next conflict event (Algorithm 3) and consecutively apply the modification with corresponding intermediate planes until the user-given target plane p_m is reached (Algorithm 4). The approach is illustrated in the third row of Figure 6.

The first type of conflict event is edges connected with f_m that shrink to zero length (Example H of Figure 6). The event positions correspond to the vertices neighboring f_m (Algorithm 3, Line 2). The second type are collapsing edges on f_m itself, as shown in Example I. Their positions can be found by intersecting planes of three consecutive faces adjacent to f_m (Line 3). Afterwards, in Line

Algorithm 3 getNextStep(f_m, p_m)

```
1:  **for** each face $f_i$ edge-adjacent to $f_m$ **do**
2:      add vertices in $f_i$ neighboring $f_m$ to $V_{ev}$
3:      add $intersect(p(f_{i\ominus 1}), p(f_i), p(f_{i\oplus 1}))$ to $V_{ev}$
4:  $V_{ev}$ = get all vertices in $V_{ev}$ between $p(f_m)$ and $p_m$
5:  **if** $V_{ev} = \emptyset$ **then**
6:      $p = p_m$
7:  **else if** $p_m$ parallel to $f_m$ **then**
8:      $v$ = get vertex in $V_{ev}$ nearest to $p(f_m)$
9:      $p$ = plane through $v$ and parallel to $p_m$
10: **else**
11:     $l = intersect(p(f_m), p_m)$
12:     $P$ = get all planes through line $l$ and each $v_{ev}$ in $V_{ev}$
13:     $p$ = plane in $P$ most parallel with $f_m$
14: **return** $p$
```

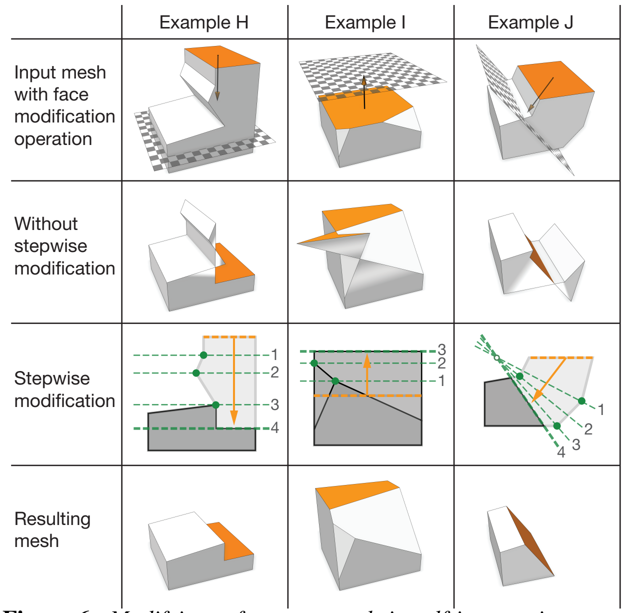

Example H
Example I
Example J
Input mesh with face modification operation
Without stepwise modification
Stepwise modification
Resulting mesh
1
2
3
4

Figure 6: *Modifying a face can result in self intersections as exemplified in the second row. To solve this, we identify the conflict events (green dots in the third row) and apply a step-by-step modification with corresponding intermediate planes.*

4, the resulting set of events V_{ev} is reduced to only the ones located between the start and end plane, e.g., in case of a push operation, the vertices above $p(f_m)$ and below p_m are removed.

If any events are remaining, the intermediate plane is determined by Lines 7-13. In case p_m is parallel to f_m, the plane through the event nearest to $p(f_m)$ is returned. Otherwise, as shown in Example J, we have to interpolate planes. Thus, a set P with planes through the intersection line l is constructed and the plane with the smallest angle to f_m is returned. Note that the first event in Example J is not the nearest one but its plane has the smallest angle.

Algorithm 4 describes the stepwise modification that continues until the applied target plane corresponds to the user input (or f_m collapses). In each loop, the face insertion needs to be applied anew (e.g., note how a new face is inserted in event 3 of Example H) and a local mesh cleanup within the 1-neighborhood of f_m is performed, e.g., to remove zero edges, zero faces or unused vertices.

Algorithm 4 mainSingle($f_m, p_m, \vec{d}, \theta$)

```
1: while $p_{tmp} \neq p_m$ and $f_m \in F$ do
2:     insertFaces($f_m, p_m, \vec{d}, \theta$)
3:     $p_{tmp}$ = getNextStep($f_m, p_m$)
4:     updateMesh($f_m, p_{tmp}$)
5:     cleanupMesh($f_m$)
```

3.4 Simultaneous Modification of Multiple Faces

So far, the modification operation has been applied to one face only. But in our PushPull++ tool, we also want to enable push- and pull-operations on an edge or a vertex. This requires the simultaneous modification of multiple neighboring faces. In the following, we explain the challenges with simultaneous modification and their solution.

Figure 7 shows examples where two faces are modified at once. The second row demonstrates that a simultaneous modification cannot

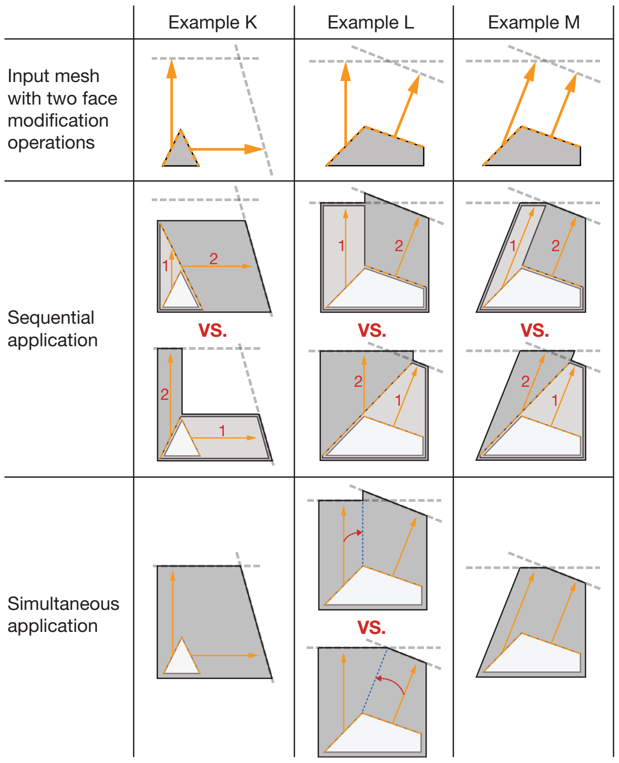


Figure 7: *Comparison of sequential and simultaneous application in case multiple faces are modified at once. Example K shows a situation where no new face needs to be inserted, in contrast to Examples L and M. The latter also has identical directions. In the sequential case, the result depends on the application order and is therefore ambiguous. In the simultaneous case, the result is only ambiguous in Example L. Thus, we allow simultaneous modifications with only one global direction.*

be serialized into two independent modifications unambiguously. Depending on the order of application, two different results emerge; e.g., in Example K, inconsistent face insertions are performed or a modification results in a face collapse. The third row depicts the simultaneous modification, which is ambiguous only when a face insertion is performed with different $\vec{d}$ directions. This is shown in Example L where which $\vec{d}$ should construct the new face is undefined. As a consequence, a simultaneous modification is limited to one global direction only. Note that even with this limitation the sequential application is still ambiguous (see Example M).

To facilitate an unambiguous simultaneous modification, we introduce simple changes in Algorithms 1 to 3. First, we add in each algorithm an outer loop that iterates over each tuple $\langle f_m, p_m \rangle$ in sets F_m and P_m, which contain all modification faces and their corresponding target planes. Second, in Line 3 of Algorithm 1, we do not check the angle of p_m against f_{adj} but against its target plane in P_m (if $f_{adj} \in F_m$). This ensures that the face insertion decision is consistent regardless of the order in F_m, i.e., only the angle between the two target planes determines if a face needs to be inserted on their edge. Third, we alter Algorithm 3 so that it returns a set of planes; i.e., for each p_m in P_m, the algorithm returns its next plane.

Finally, we replace Algorithm 4 with Algorithm 5. In contrast to Algorithm 4, the main loop in Algorithm 5 iterates until the modification face set F_m is empty (decremented by Lines 6 - 8) and the subroutine calls operate on sets F_m and P_m instead of a single face with its target plane only.

Algorithm 5 mainMultiple($F_m, P_m, \vec{d}, \theta$)

```
1: while $F_m \neq \emptyset$ do
2:     insertFaces($F_m, P_m, \vec{d}, \theta$)
3:     $P_{tmp}$ = getNextStep($F_m, P_m$)
4:     updateMesh($F_m, P_{tmp}$)
5:     cleanupMesh($F_m$)
6:     for each $\langle f_m, p_m, p_{tmp} \rangle$ in $\langle F_m, P_m, P_{tmp} \rangle$ do
7:         if $p_{tmp}$=$p_m$ or $f_m \notin F$ then
8:             remove $f_m$ in $F_m$ and $p_m$ in $P_m$
```

4 The PushPull++ Tool

One way to create a straightforward user-interface for our method is to map the parameters $f_m, p_m, \vec{d}$, and θ directly to user-interface elements. We have implemented this in the direct plane modification tool (Figure 8) as follows: When the user clicks on a face f_m, a 3D gizmo is shown. This gizmo has three orthogonal arrows. Dragging them defines the direction $\vec{d}$. It also has three circles that allow the user to specify the orientation of the target plane p_m.

The threshold angle θ is specified with a settings slider. The choice of θ mainly depends on the modeling task. To model the new geometry, a threshold below 30° is preferable, as this often leads to newly created faces. When deforming the existing geometry, a high threshold of about 60° is better suited, because existing faces are often used instead of creating new ones.

While this tool allows full control, it can be difficult get a desired result, because the many degrees of freedom and the face plane snapping behavior do not intuitively map to the output. For example, if no new faces need to be inserted, changes in $\vec{d}$ might have no impact, i.e., a drag along the target plane does not impact the resulting model. Therefore, we introduce methods to automatically find good parameters and propose useful directions.

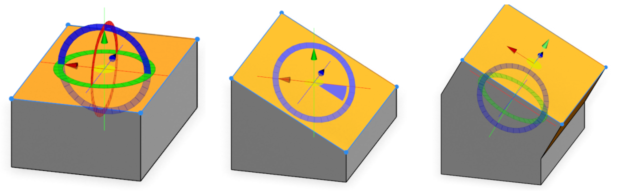


Figure 8: *In direct plane modification tool 3D gizmos are mapped to method parameters p_m and $\vec{d}$. θ is adjusted with a slider.*

4.1 Finding Useful Parameters

Instead of defining the planes unintuitively with a three axis rotation gizmo, we calculate them from component drag operations. This means, that when a user performs a push- or pull-operation on a face, the target plane is simply offset by the drag distance. To rotate planes, the user can drag edges or vertices. When an edge is dragged, the adjacent faces are tilted.

Directions Depending on the modeling task and component type, different directions can be useful. Possible directions for faces include, in order of priority: the face normal $\hat{n}$, the vector $\vec{n}_{proj}$ which is $\hat{n}$ projected on the horizontal plane, the world coordinate axis $\vec{a}_y$, and the set D_{adj} of directions along adjacent faces (including averages between them). Directions in D_{adj} are calculated by the cross product of the adjacent face normal with the edge direction. When moving an edge, we use $\vec{a}_y$, $\vec{a}_x$, $\vec{a}_z$ and its set of adjacent directions D_{adj}. For vertices, we use $\vec{a}_y$, $\vec{a}_x$, $\vec{a}_z$ as well as D_{adj} containing the directions of the adjacent edges.

The system automatically maps threshold values θ to the different directions, suitable for most modeling tasks. In our implementation, we use 60° for all directions in D_{adj}, because they are mainly used to adjust meshes, and 15° for the other directions. No additional setup by the user is required, and the user simply chooses between them interactively by hovering over corresponding arrows. If the user wants full control, he/she can override the automatic value with the angle threshold settings slider.

Figure 9(left) shows an example with proposed directions for a face. As seen, showing all those directions creates a cluttered result. Moreover, some of them actually create the same mesh. Hence, to present only the useful directions to the user, we have to filter them.

First, we analyze the potential outcome of Algorithm 1 for each possible direction. In case mutliple directions create the same output, i.e., no new faces are inserted or zero faces with identical normals are generated, we use only the direction with the highest priority. Note that the priority is defined by the ordering of directions in the last paragraphs. Second, as there can be a lot of directions in D_{adj}, we use k-means clustering with a dot product distance function to filter them.

Finally, starting with the highest priority, the directions are added as arrows displayed to the user, provided no direction is already in this set with an angle below a certain threshold (we use 10°), or the maximum number of directions is reached (here we use 5). When there are similar directions with different thresholds during filtering, we keep them both and display them next to each other. A reduced set is shown in Figure 9(right).

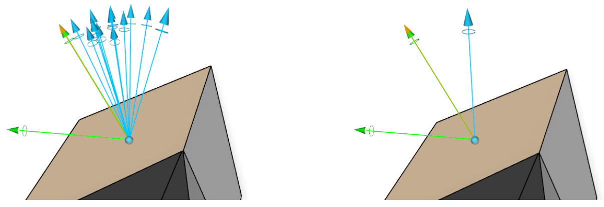


Figure 9: *Unfiltered and filtered choices for the direction $\vec{d}$. The normal is orange, adjacent directions D_{adj} are blue, global axes and the horizontally projected normal are green.*

Plane Equations When an edge or a vertex is dragged, multiple faces are modified simultaneously. To determine the target plane p_m for every adjacent face f_{adj}, we constrain the vertex v_f in f_{adj} furthest away from the edge not to move. This lets us calculate the new plane equations: If $\vec{e}$ is the edge direction, and if v_e is a vertex of e, then the new plane is given by $p_n = normalize(normalize((v_e + \vec{d}) - v_f) \times \vec{e})$ with distance $d = p_n \cdot v_f$, as illustrated in Figure 10 . For vertex moves, all adjacent faces are tilted in a similar fashion to the edge move tool. For an adjacent face f, given the vertex position v, the new plane is calculated with $p_n = normalize((v + \vec{d}) - v_f)) \times (normalize(v - v_f) \times f.\hat{n})$.

Using a hotkey, the user can also restrict the modification to only one of the adjacent faces, allowing quick modeling of dormer windows. Alternatively to the vertex furthest away, the vertex with the highest height coordinate can be used as v_f.

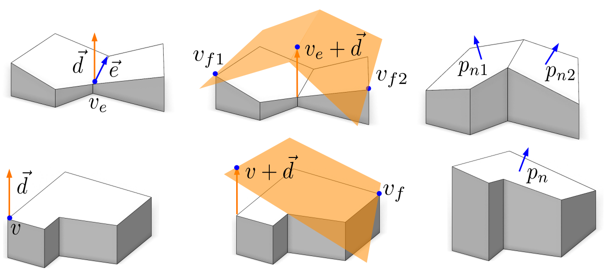


Figure 10: *Examples of the edge drag (top) and vertex drag (bottom) operations. Left: The initial meshes with drag directions $\vec{d}$. Middle: Resulting target planes with vertices v_f that are furthest away. Right: Final mesh with calculated plane normals p_n.*

4.2 PushPull++ User Interface

In this subsection, we introduce the user interface of PushPull++ . It is based on the useful parameters found in the previous section, combined with polyline drawing and polygon splitting, enabling comprehensive mesh modeling with a single tool.

The interface is shown in Figure 11: While hovering over a mesh, a small sphere handle is shown to highlight the mesh component located under the cursor. When the user drags the sphere, this component is selected, defining f_m. Useful directions d are calculated as explained previously and shown as arrows. While dragging, the arrow nearest to the mouse is selected and highlighted. For improved controllability, switching between arrows is disabled when the move amount is very small or in the negative direction.

Clicking on the mesh outside of a sphere handle inserts a vertex and starts a polyline drawing operation: every click adds a new vertex to the polyline, snapping guides are shown to increase precision, and existing faces are automatically split when intersected by the polyline using a face loop finding algorithm. The operation stops either after a split or when the polyline is closed to form a polygon. For convenience, a rectangle drawing mode is also available with a hotkey. To allow starting a polyline under a sphere handle, handles disappear when the mouse remains stationary for longer than three seconds.

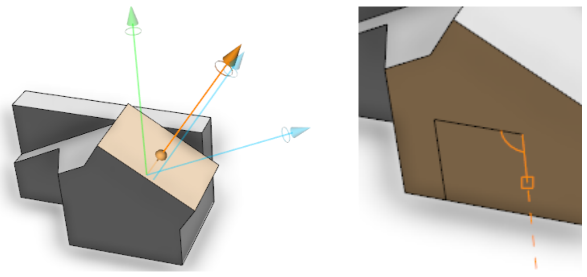


Figure 11: *PushPull++ tool: A sphere can be dragged along arrows to define the parameters (left). Additionally, polyline drawing can be used to split and create polygons (right).*

5 Evaluation

Our method and the PushPull++ tool were implemented in Java and combined with standard 3D editor features like camera control, basic transformations, and texture assignment tools. Examples created using the PushPull++ tool are shown in Figure 12.

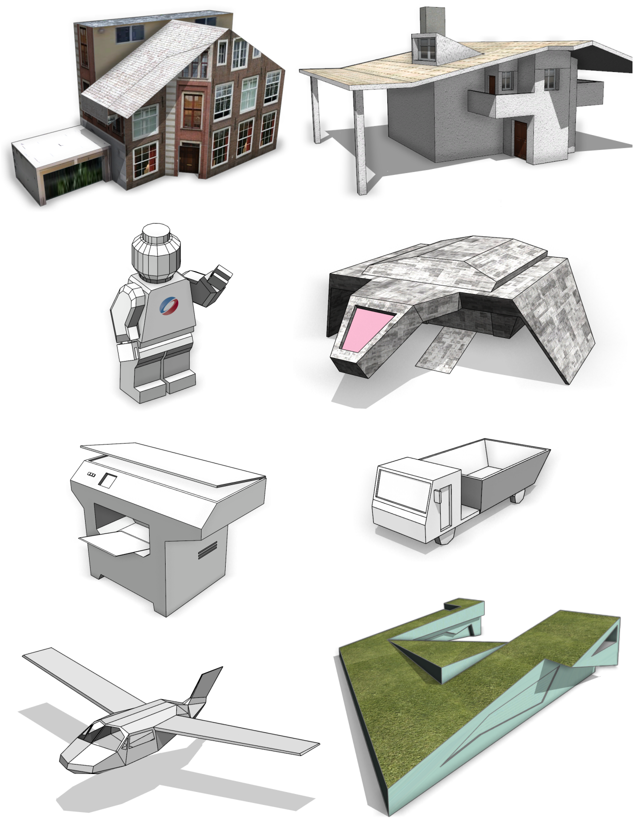


Figure 12: *Results created in 5 to 15 minutes using our PushPull++ tool combined with texturing and standard transformations.*

Comparison We compared our method with AutoCAD [Autodesk 2014a], SketchUp [Trimble 2013], and Maya [Autodesk 2014b] for every modeling step shown in Figure 13. Two metrics were employed: The first was the minimum amount of mouse and keyboard interactions (clicks or drags) required by expert users for those modeling steps. Camera controls were ignored. This metric compares the efficiency of each entire software package. For the second metric, we attempted to eliminate the influence of the user interface as much as possible and just measure the effect of the modeling operations. Therefore, we only counted interactions directly causing a change in the geometry or drawing planes. Tool changes, menu clicks, selections and camera controls were ignored.

To determine those interaction counts, one expert user per software package performed the modeling steps shown in Figure 13. The user had unrestricted time to find a modeling approach with the minimum amount of interactions and was allowed to use all tools provided by the software. The best effort was recorded and all interactions were counted. The results are shown in Table 1.

Discussion One important observation from Table 1 is that our method allows all adjustment operations (a-h) steps with just one mouse drag, enabling direct interactive adjustments. AutoCAD requires additional clicks for selection and setup, although it also requires only one geometry-changing interaction. This is because these operations essentially degenerate to a constrained move, always following existing faces. The other tools require much higher click counts, because they do not have a mechanism to maintain planarity. Essentially the user has to fix non-planar faces manually with low-level transform tools and snapping.

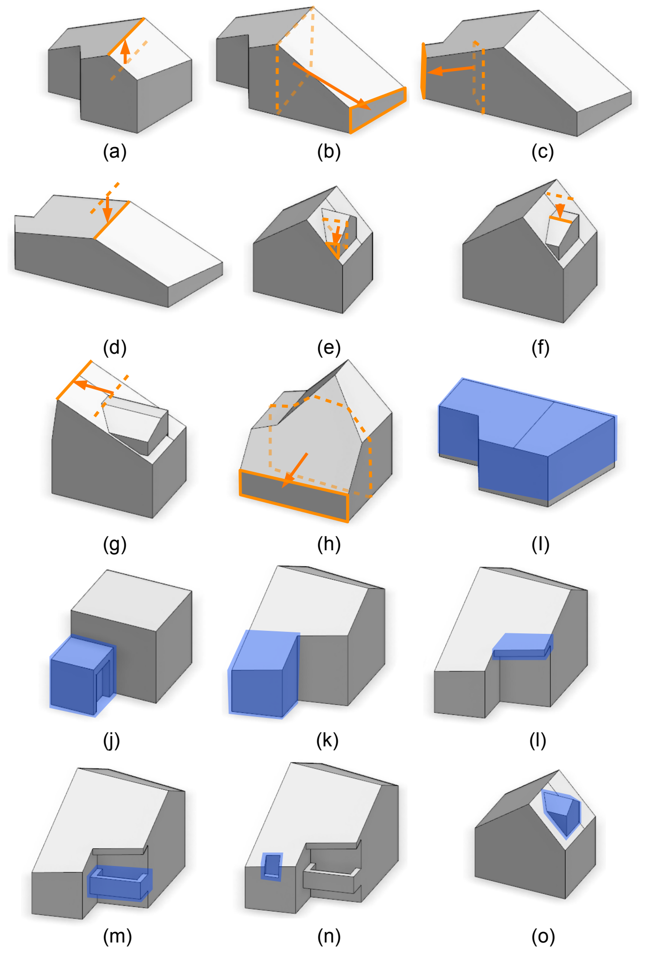


Figure 13: *Editing examples used for evaluation. The orange arrow indicates moved components; the blue indicates added details.*

Op.	Ours		AutoCAD		SketchUp		Maya	
a	1	1	5	1	17	8	17	4
b	1	1	5	1	14	6	36	5
c	1	1	5	1	15	7	14	5
d	1	1	5	1	24	10	26	7
e	1	1	7	1	12	5	15	6
f	1	1	5	1	10	3	10	3
g	1	1	4	1	11	3	8	3
h	1	1	14	2	24	13	26	8
i	3	2	20	3	5	2	6	2
j	7	4	16	4	10	4	48	14
k	4	1	17	3	27	7	26	8
l	3	1	22	6	33	10	9	4
m	9	4	43	9	31	11	54	18
n	9	4	28	7	24	7	66	21
o	8	5	33	7	28	8	21	7
rel.	1	1	5.5	1.7	10.5	5.3	12.4	4.6

Table 1: *Minimum number total clicks (left columns) and geometry-changing interactions (right columns) required by expert users to complete the steps in Figure 13. rel.: relative to ours, averaged.*

Our low counts in examples (i) and (j) are mainly because we allow extrusions and splitting without tool change. SketchUp holds up well in (j), because it has a specific polygon splitting tool. Such splits are more involved in AutoCAD; due to the solid modeling nature, they always require an imprint step. Maya only has a line-split, and thus requires many more steps in (j).

In the complex modeling steps (k-o), our tool significantly outperforms the other tools. This is mainly caused by the adaptive face insertion, as it allows efficiently adding details to meshes with slanted surfaces. Such steps have to be emulated in AutoCAD with boolean operations and global plane splits, in Sketchup with manual inference-based constructions and in Maya with manual vertex movements.

The last row in the table shows the averaged metrics relative to our tool. AutoCAD holds up quite well for the second metric (1.7), but falls significantly behind for the total clicks (5.5). SketchUp and Maya perform five to ten times worse than our approach.

Performance All examples in this paper can be edited interactively with at least 30 updates per second on mainstream hardware (e.g., Intel Core i5). The method scales linearly with the number of affected faces. Depending on the used data structure, finding the adjacent faces may require an initial setup cost. For a vertex-edge-face structure, this requires just constant time [Foley 1996]. The outer loop in Algorithm 5 depends on the number of edge collapse events, which has an upper bound in the number of edges.

6 Conclusion

Limitations and Future Work Currently, only local self-intersections are prevented. It would be interesting to extend the algorithms for global intersection handling. While intersection detection can be done with existing methods, crafting an intuitive response itself is an interesting avenue for future work. Extending our approach for curved surfaces would also be interesting.

Conclusion We have introduced the PushPull++ tool for rapid modeling of plane-dominant objects. We have shown that it significantly outperforms commercial state-of-the-art tools, especially when objects contain slanted surfaces. PushPull++ employs a new local and efficient method with three novel components: First, face insertion is adaptive based on threshold values and directions. Second, adjacent faces are updated to keep them planar, possibly changing vertex valences. Third, edge collapses are handled with an efficient stepwise method, without requiring boolean operations. Our method is a generalization of previous PushPull approaches, supporting both existing and new editing operations.

Acknowledgments

We thank Tom Kelly for his valuable inputs on the methodology, Matthias Buehler and all other testers for helping with the evaluation and creating examples, Virginia Unkefer for proof reading, Tina Smith for video voice-over, and the anonymous reviewers for their constructive comments on improving this paper. Furthermore, we thank Scott Morehouse and Jack Dangermond for making this publication possible.

This project is supported in part by the European Commission under contract number FP7 608775 (INDICATE).

References

AUTODESK, 2014. Autocad, http://www.autodesk.com/autocad.

AUTODESK, 2014. Maya, http://www.autodesk.com/maya.

BAUMGART, B. G. 1974. *Geometric modeling for computer vision.* PhD thesis, Stanford, CA, USA.

BERNSTEIN, G., AND WOJTAN, C. 2013. Putting holes in holey geometry: Topology change for arbitrary surfaces. *ACM TOG 32*, 4, 34:1–34:12.

BOKELOH, M., WAND, M., KOLTUN, V., AND SEIDEL, H.-P. 2011. Pattern-aware shape deformation using sliding dockers. *ACM TOG 30*, 6, 123:1–123:10.

BOKELOH, M., WAND, M., SEIDEL, H.-P., AND KOLTUN, V. 2012. An algebraic model for parameterized shape editing. *ACM TOG 31*, 4, 78:1–78:10.

BOTSCH, M., AND SORKINE, O. 2008. On linear variational surface deformation methods. *IEEE Trans. on Visualization and Computer Graphics*, 1, 213–230.

BOTSCH, M., PAULY, M., KOBBELT, L., ALLIEZ, P., LÉVY, B., BISCHOFF, S., AND RÖSSL, C. 2007. Geometric modeling based on polygonal meshes. In *ACM SIGGRAPH courses*.

BOUAZIZ, S., DEUSS, M., SCHWARTZBURG, Y., T. WEISE, T., AND PAULY, M. 2012. Shape-up: Shaping discrete geometry with projections. *Comp. Graph. Forum 31*, 5, 1657–1667.

CABRAL, M., LEFEBVRE, S., DACHSBACHER, C., AND DRETTAKIS, G. 2009. Structure preserving reshape for textured architectural scenes. *Comp. Graph. Forum 28*, 469–480.

DENG, B., BOUAZIZ, S., DEUSS, M., ZHANG, J., SCHWARTZBURG, Y., AND PAULY, M. 2013. Exploring local modifications for constrained meshes. *Comp. Graph. Forum 32*, 2, 11–20.

FOLEY, J. 1996. *Computer graphics: principles and practice*. Addison-Wesley Professional.

GAL, R., SORKINE, O., MITRA, N., AND COHEN-OR, D. 2009. iWIRES: An analyze-and-edit approach to shape manipulation. *ACM TOG 28*, 3, 33:1–33:10.

HABBECKE, M., AND KOBBELT, L. 2012. Linear analysis of nonlinear constraints for interactive geometric modeling. In *Comp. Graph. Forum*, vol. 31, 641–650.

HAVEMANN, S., AND FELLNER, D. 2005. *Generative mesh modeling*. PhD thesis, Technische Universität Braunschweig.

KELLY, T., AND WONKA, P. 2011. Interactive architectural modeling with procedural extrusions. *ACM TOG 30*, 2, 14:1–14:15.

KOBBELT, L., CAMPAGNA, S., VORSATZ, J., AND SEIDEL, H. 1998. Interactive multi-resolution modeling on arbitrary meshes. In *Proceedings of Computer graphics and interactive techniques*, 105–114.

KRAEVOY, V., SHEFFER, A., SHAMIR, A., AND COHEN-OR, D. 2008. Non-homogeneous resizing of complex models. *ACM TOG 27*, 5, 111:1–111:9.

KRIPAC, J., 2005. Controlled face dragging in solid models. US Patent 6,867,771.

SASAKI, N., CHEN, H., SAKAMOTO, D., AND IGARASHI, T. 2013. Facetons: face primitives with adaptive bounds for building 3d architectural models in virtual environment. In *Proc. of the 19th ACM Sym. on VR Software and Technology*, 77–82.

SCHELL, B., ESCH, J., AND ULMER, J., 2003. System and method for three-dimensional modeling. US Patent 6,628,279.

SHAPIRO, V. 2002. Solid modeling. *Handbook of computer aided geometric design 20*, 473–518.

TRIMBLE, 2013. Sketchup, www.sketchup.com.

VAXMAN, A. 2012. Modeling polyhedral meshes with affine maps. *Comp. Graph. Forum 31*, 5, 1647–1656.

YANG, Y., YANG, Y., POTTMANN, H., AND MITRA, N. 2011. Shape space exploration of constrained meshes. *ACM TOG 30*, 6, 124:1–124:12.

ZELEZNIK, R., HERNDON, K., AND HUGHES, J. 1996. Sketch: An interface for sketching 3d scenes. In *Proc. of SIGGRAPH 96*, 163–170.

ZHENG, Y., FU, H., COHEN-OR, D., AU, O., AND TAI, C. 2011. Component-wise controllers for structure-preserving shape manipulation. In *Comp. Graph. Forum*, vol. 30, 563–572.

ZORIN, D., SCHRÖDER, P., AND SWELDENS, W. 1997. Interactive multiresolution mesh editing. In *Proceedings of Computer graphics and interactive techniques*, 259–268.