Cut and Paint: Occlusion-Aware Subset Selection for Surface Processing

Mohamed Radwan
TU Wien, Austria

Stefan Ohrhallinger
TU Wien, Austria

Elmar Eisemann
TU Delft, Netherlands

Michael Wimmer
TU Wien, Austria

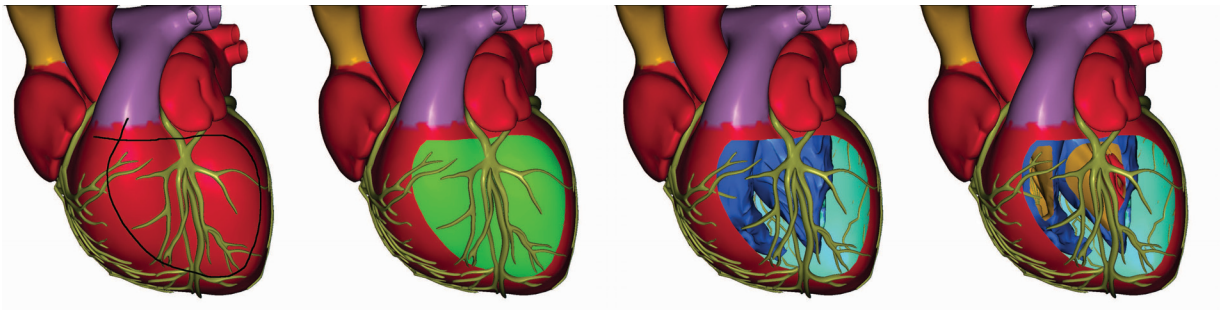


Figure 1: Occlusion-aware cutaway generation. From left to right: User-drawn curve, selected region (shaded green), cutout revealing interior, final illustration using consecutive cutaways.

ABSTRACT

Surface selection operations by a user are fundamental for many applications and a standard tool in mesh editing software. Unfortunately, defining a selection is only straightforward if the region is visible and on a convex model. Concave surfaces can exhibit self-occlusions, which require using multiple camera positions to obtain unobstructed views. The process thus becomes iterative and cumbersome. Our novel approach enables selections to lie under occlusions and even on the backside of objects and for arbitrary depth complexity at interactive rates. We rely on a user-drawn curve in screen space, which is projected onto the mesh and analyzed with respect to visibility to guarantee a continuous path on the surface. Our occlusion-aware surface-processing method enables a number of applications in an easy way. As examples, we show continuous painting on the surface, selecting regions for texturing, creating illustrative cutaways from nested models and animate them.

Index Terms: Computer Graphics [I.3.3]: Picture/Image Generation—Line and curve generation Computer Graphics [I.3.3]: Picture/Image Generation—Viewing algorithms

1 INTRODUCTION

Selecting regions of a model's surface is part of the workflow of many 3D designers. One application is to continuously *paint* on the surface of the model, without having to rotate it when occluding parts cover the brush path. Similarly, a *segmentation* of the surface is needed for texture assignments. Hiding surface selections is also useful to remove otherwise hidden elements, e.g., spurious primitives in the interior resulting from isosurface meshing. Finally, hiding surfaces is at the basis of *cutaway* illustrations, which are the standard for technical illustrations, but also have applications in the medical domain; from pre-operative planning to post-operative visualization.

The process of defining a region boundary on an object with moderate or high depth complexity is non-trivial. Tools in current 3D design software products, such as Maya or Blender, only permit simple strokes to select a region. Additionally, complex occlusions may make it impossible to find a viewpoint that maps a screen-space curve to the surface. Most existing methods for cutaways only work on manually pre-defined primitives [16] or take minutes of processing [6], while 3D artists require interactive speed.

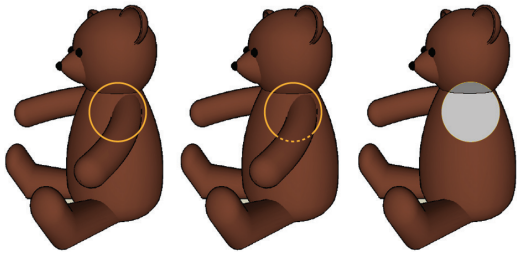


Figure 2: Occlusion-aware surface cut from 2D curve in screen space. A 2D curve is projected onto the surface in a closed curve. Thanks to occlusion-awareness the cut-out is performed on the body, removing the arm entirely, instead of a "cookie-cut" (simple removal of surface points mapping back inside/onto 2D curve).

In this paper, we propose a new method for occlusion-aware surface selections, inferred from a user-drawn curve, which aims at a plausible outcome. The input 2D curve is projected onto the surface, resulting in one or more potential selection-bounding curves. For high depth complexity, the selection can be ambiguous and we offer a control to specify the desired candidate boundary via a selection point on the screen-space curve. To provide an example, Figure 2 illustrates how to "amputate" a bear's arm without having to rotate the object into a view, where the arm does not occlude the body.

Our main technical contribution is the interactive inference process that determines interactively a plausible selection boundary on 3D meshes from a single screen-space input curve without requiring any pre-processing. The input mesh can have arbitrary depth complexity, self-intersections, and can even contain holes, as long as the connectivity between the triangles is manifold. The curve can be modified in screen space, or dragged on the surface of the mesh. Further, we support animation, which can be of interest for many illustrative purposes.

Our contributions are:

- Efficient mapping of a screen-space curve onto a mesh with

16-19 May, Edmonton, Alberta, Canada

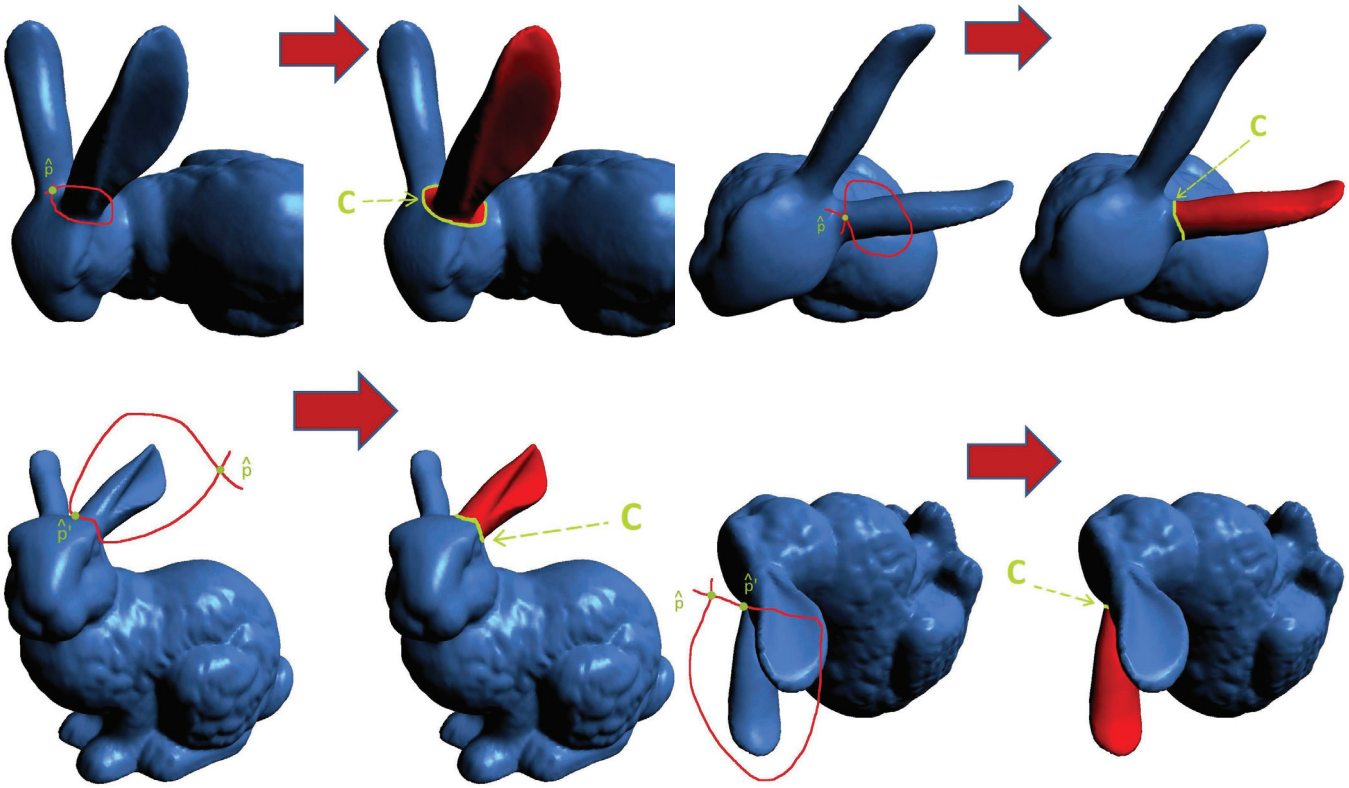


Figure 3: We explicitly want to select the left ear of the BUNNY from different views. In each pair of sub-figures, the boundary curve C is outlined green and the bounded region shaded red. In all cases, C is selected by the default position of the selection point. The selection point $\hat{p}$, colored green, is traced forward in drawing direction to $\hat{p}'$ in the two cases at bottom.

arbitrary depth complexity.

- Interpreting user input to derive a plausible selection boundary.
- Dynamically extending a selection boundary in an occlusion-aware manner.
- Sample applications: Illustrative cutaway, texturing, painting and animation.

We summarize and review previous and related approaches (Sec. 2), before explaining our surface selection algorithm (Sec. 3). We then cover the details of our discrete implementation (Sec. 4), and present applications to the approach (Sec. 5).

Finally, we discuss the time performance results (Sec. 5.4) and conclude (Sec. 6) with an outlook on future work.

2 Related Work

User-driven region selection. Our method can be categorized as a user-driven region-selection method, which is – unlike data-driven segmentation approaches [1, 17, 20] – based on user input with drawing tools. In the approach of Fan et al. [8], the user places a brush stroke inside the region to be selected and the final selection is based on a shape diameter function. Two strokes are used in the easy mesh cutting approach of Ji et al. [14], one for the selected region and one for its complement, which then derives the selection boundary via an optimization process. Alternatively, like our method, the user can define the boundary of the selection directly. This is more intuitive according to a recent comparative study [9]. The iCutter method [18] constructs a scalar field on the surface and finds the best isoline based on centerness and concavity. The intelligent-scissors method [11] defines a cost function over the mesh edges – based on concavity, length, and closeness to the stroke – and solves a constrained least-cost path problem to find the optimal curve. In semi-automatic mesh scissoring [15], the user-drawn contour on the visible side is completed by a weighted shortest path that is attracted to concave features. In the cross boundary brush [21] of Zheng and Tai, the user is asked to draw the stroke across the boundary, and selects an isoline extracted from a harmonic field. The dot scissor [22] asks the user to place a circle in a single click, and finds the best cut passing through the circle from a number of isolines of concavity-aware harmonic fields. All these approaches assume that the user wants to place the boundary along surface lines with concave features, based on the minima rule [5, 12], and the curve is optimized accordingly. Our method gives the 3D designers the freedom to draw the boundaries precisely as they want, allowing for arbitrary shapes. Furthermore, our stroke drawing does not have to stop at occlusions (from other objects or a self-occlusion). If desired, however, attracting curves to concavities as in the above approaches can complement our method.

Cutaways. Our method can be used to create manual cutaways without having to change the view, and it can even automatically generate cutaways to reveal objects of interest.

In the interactive cutaways method [16] of Li et at., it is assumed that the model is subdivided into basic shapes from a set of common

geometric shapes. They associate a parameterized cut to each of these shapes. Dynamic cutaways are generated by automatically cutting the parts which occlude the object of interest. Our approach can be used to create cutaways without any prior segmentation of the model. A method for adaptive cutaways [2] reveals objects of interest by generating a cutaway surface in a depth image, and excludes parts of the occluding objects during rendering at interactive rates. However, removing a part of an occluding object can lead to separate elements, which are typically avoided in illustrative cutaways. Our method can remove the entire region of the model bounded by a selected curve via a flood-filling process.

Occlusion-aware painting. The interactive texture-generation system Chameleon [13] features an intelligent brush that extends painted strokes to occluded regions in order to keep the continuity. The system starts with a visible polygon and then adds adjacent polygons intersecting with the stroke in screen space, but behaves like a 2D stencil – in case a connected part occludes the brush boundary, the region outside of it will not be painted. LayerPaint is a multi-layer painting tool presented by Fu et al. [10], that can draw long strokes across different depth layers. The method is based on the work of Eisemann et al. [6], in which the mesh is decomposed based on a graph analysis, whose nodes are view-dependent regions of constant visibility connected by “adjacent to" and “occludes" relations. However, the decomposition requires minutes of pre-processing time and painting is only possible inside a single visibility layer along a view ray. Our method is interactive and maps the brush boundary as a single loop onto the 3D model, so that it also paints protruding parts.

3 The Occlusion-Aware Selection Algorithm

Our method can be applied to arbitrary manifold meshes but is particularly beneficial for objects with high depth complexity. The surface can contain holes, and its triangles can even intersect each other, as long as mesh connectivity is not affected. The user draws a closed 2D curve in screen space, which we map to selection boundaries on the 3D mesh. Our goal is to find a curve which projects to the user-defined 2D curve and limits the desired region. In principle, there can be more than one correspondence; however, each point on the 2D curve implies a unique selection on the mesh, when enforcing the inclusion of the underlying visible surface point. This insight provides a simple interface to select the desired solution; the user can investigate alternative choices by moving the *selection point* along the previously-drawn curve, and choose the selection boundary they want. Figure 3 illustrates the ease of selecting the left bunny ear in different views.

Now we describe the theoretical concept, while the practical implementation in discrete screen-space is covered in Sec. 4.

3.1 The Mapping of Screen Space Curves to the 3D Model

The input to our algorithm is a two-dimensional surface S (possibly with boundary) embedded in $\mathbb{R}^3$ and a closed and simple non-self-intersecting 2D curve $\hat{C}$ drawn by the user in the view plane H. This closed curve $\hat{C}$ is defined by the boundary of the largest closed area defined by an input stroke drawn by the user, i.e., the user does not have to provide a point-accurate loop, but can draw a self-intersecting curve.

This closed manifold boundary curve in screen space $\hat{C}$ results in a cone (a 2-manifold) that when intersected with S (a 2-manifold) results in a set of curves $\mathbb{C}$ (1-manifold). A single point can then be used to identify the desired curve C, and the local differentiability of the 1-manifolds can be used to cut the surface by visiting the local neighborhood of faces. C will then separate S into two connected components, unless it cuts through a handle (then an additional curve cutting through that handle is required).

Let Π be the projection from 3D space to the view plane. We define the set of boundaries $\mathbb{C}$ as the set of all curves on S, whose

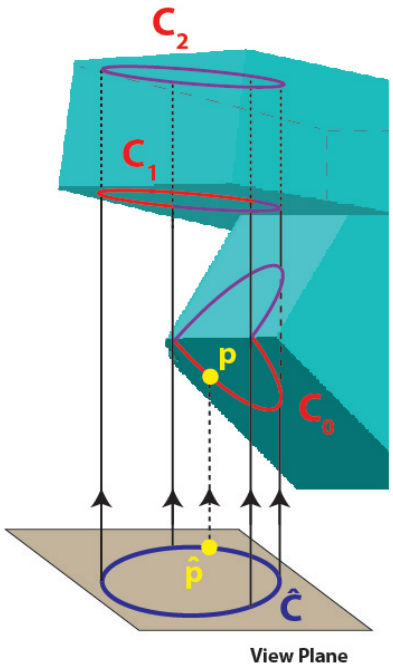


Figure 4: The blue curve $\hat{C}$ drawn on the view plane is projected onto the model surface as a number of curves $C_{0..2}$. The curve segments are colored red (visible), or magenta (occluded). $\mathbb{C} = \{C_0, C_1, C_2\}$, while the candidate set is $\mathbb{C}_{vis} = \{C_0, C_1\}$, as C_2 is entirely invisible . $\hat{p}$ on $\hat{C}$ is the *selection point*, and p is its closest projection onto the surface. In the figure, p lies on C_0, so $C = C_0$ is the boundary curve selected.

projection is contained in $\hat{C}$, i.e.:

$$\mathbb{C} := \{C \subseteq S | \Pi(C) \subseteq \hat{C} \wedge C \text{ is closed}\} \quad (1)$$

Note that the projection of a curve $\Pi(C)$ is a subset of $\hat{C}$, but not necessarily equal, e.g., when C is partially backfacing (e.g., C_0 in Figure 4). In order to extract the selected subset from the model surface S, we require a singly-connected curve (loop) $C \in \mathbb{C}$ forming the boundary of the selected region in S. We assume that while the user is drawing $\hat{C}$, they would not expect the boundary curve to be entirely occluded. So, $C \in \mathbb{C}_{vis}$, where $\mathbb{C}_{vis} \subseteq \mathbb{C}$ is the set of visible and partially visible curves. Any point $p \in \mathbb{C}$ uniquely determines a C_i, since the $C_i \in \mathbb{C}$ are disjoint. Therefore, to extract C, it is sufficient to provide a single such point. To this extent, the user provides a *selection point* $\hat{p} \in \hat{C}$ (see Figure 4), which is then associated to a visible point $p \in C$ on the surface that projects to $\hat{p}$. By moving $\hat{p}$ along $\hat{C}$, the user can select any $C \in \mathbb{C}_{vis}$.

By default, $\hat{p}$ is initialized with the first intersection point of the input stroke that resides on $\hat{C}$. In case $\hat{p}$ does not project on S, we displace $\hat{p}$ on $\hat{C}$ along the drawing direction of the input stroke until S is hit. Figure 5 shows how the user can instantly select a specific finger by positioning $\hat{p}$ while drawing the stroke. Other ways to set the initial location of $\hat{p}$ are possible, such as to position it on the curve with the longest visible subset. Nonetheless this is not crucial, as moving $\hat{p}$ to the desired curve is usually easy and quick.

Given C, a flood fill is performed on the surface from an arbitrary visible point on it, such that it spreads into its interior as determined by the current view (Figure 6).

In some cases, it may be useful to select additional boundary curves, which can be done by picking more selection points. In this case, the flood fill stops at any boundary of the selected set. This is required to handle cases such as the one cutting through a handle

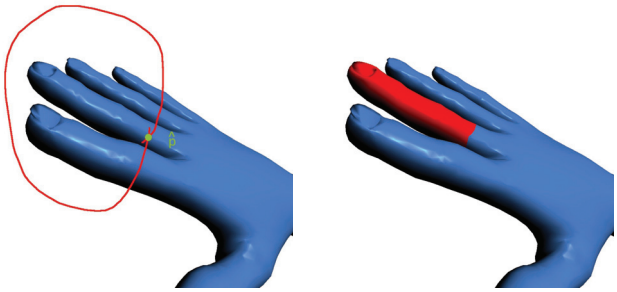


Figure 5: The user indicates the *selection point* $\hat{p}$ by the self-intersection of the stroke in order to select the intended finger.

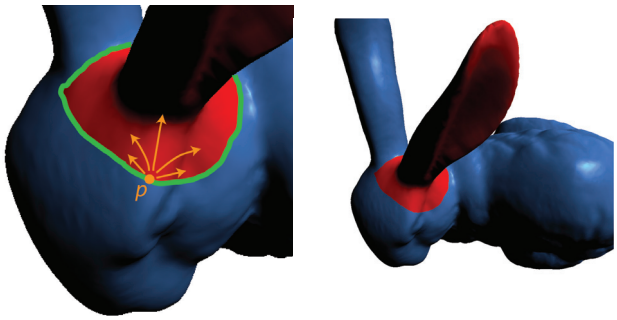


Figure 6: Flood-filling determines the “interior" of the selection. Left: The detail shows the boundary C on the surface shaded green, the marked subset of the surface shaded red. The orange arrows show the direction of flood-filling from the selection point p on the boundary towards the interior of the closed curve $\hat{C}$ in screen space. Right: The object viewed at large.

in Figure 7, where it is not possible (and is easily detected since the flood fill reaches back to the starting boundary from the other side) to bound the desired region with a single boundary. Further, it facilitates cutting out regions which break up a model into parts (see Figure 13).

4 Implementation in Discrete Screen Space

Our solution could be implemented by working directly on the geometry but it would require intersecting view rays with possibly complex geometry, which can be costly and might prohibit interactivity, especially as static structures would have to be rebuilt each time the geometry changes or is deformed. Instead, we use a hybrid approach, which traces the actual curves on the surface by following

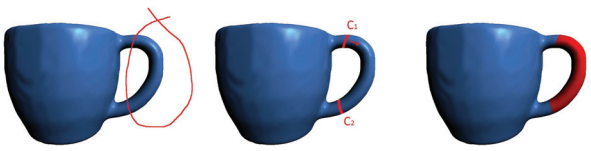


Figure 7: Intersecting handles: The user-drawn curve intersects a single handle, resulting in two partially visible curves C_1 and C_2. In order to select the handle region, the user has to choose both boundary curves C_1 and C_2. Flood-filling starts from C_1 to the interior direction and stops at C_2 to mark the red shaded region.

the mesh connectivity, but the starting positions of the curves are derived in image-space with a standard $O(N)$ rendering pass. Once the boundary curve is extracted, we can determine the region by a flood-filling process.

From the user-provided curve, we extract the largest connected area by a flood-fill process in screen space, and derive its 4-pixel connected boundary. The resulting pixel curve is then transformed into a polyline representing $\hat{C}$, that connects the centers of adjacent pixels. The first observation is that for each vertex p_i of the polyline $\hat{C}$, we can easily find the triangle t_i that contains its projection: as $\hat{C}$ aligns by construction with the pixel raster, we can draw the scene and output triangle indices instead of colors. The indices underneath $\hat{C}$ correspond to candidate starting triangles for the curve extraction. A single depth-buffered rendering is sufficient to extract the front-most triangle. This approach might miss subpixel curves, but given that they would also be invisible to the user, this does not represent an important limitation.

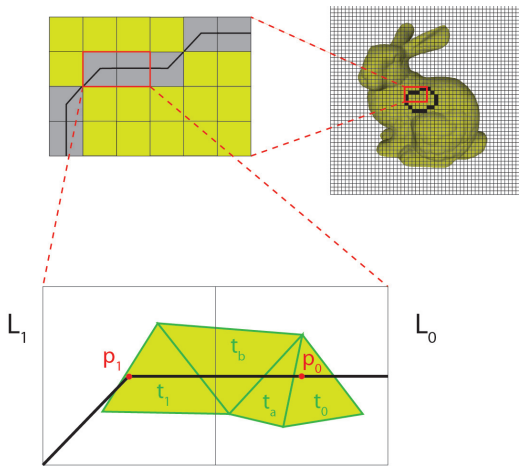


Figure 8: Finding curves on a mesh. Upper right: Discrete user input in screen space (thinned to an 4-connected curve) on the BUNNY model. Upper left: Detail of input (shaded grey). Bottom: Detail of two pixels on the curve. The triangle index of the visible fragment at the center of each pixel is stored (t_0 for L_0 and t_1 for L_1). Assume the tracing starts at pixel L_0 to find C. The projection of $[p_0, p_1]$ intersects triangles t_0, t_a, t_b, and t_2, and they are all added to C. The process continues with $[p_1, p_2]$ and subsequent line segments, until C is completed when t_0 is reached in L_0 again.

We will now determine a list of all triangles along the boundary selected by the user. These triangles will later be cut to adhere to the corresponding region boundary on the surface.

To extract C, we take the pixel of the selection point as the starting point. For illustration, let us assume the starting point is p_0. Let $[p_0, p_1]$ be the line segment between p_0 and p_1 in $\hat{C}$. When projecting $[p_0, p_1]$ onto t_0, which is the first triangle added to our list, we may intersect its boundary. In this case, we cross over to the adjacent intersected triangle and add it to our list as well. If this triangle is front-facing, we intersect it again with $[p_0, p_1]$. If it contains p_1, we continue with $[p_1, p_2]$, else we visit the next adjacent intersected triangle. If the adjacent triangle is backfacing, the traversal direction is flipped and the process continues with $[p_1, p_0]$. This also handles the case where the projection of $\hat{C}$ intersects a boundary of the mesh, as the orientation changes between the two adjacent triangles on the boundary. The process (see Figure 8 for an example) is repeated until the entire curve C has been processed. The algorithm is outlined in Algorithm 1,

In case the user selects multiple boundary curves (Figure 7), we simply execute the algorithm for each curve and its selection point.

Algorithm 1 Collecting triangles on the selected boundary

Input: p_{i_0} ▷ the center of the pixel that $\hat{p}$ projects on
Input: t_{i_0} ▷ the triangle which p_{i_0} lies inside
Output: C ▷ list of triangles that $\hat{C}$ passes through

$i \leftarrow i_0$ ▷ index of current point on $\hat{C}$. initialized.
$t \leftarrow t_{i_0}$ ▷ current triangle. initialized
$C \leftarrow [\,]$
Finish $\leftarrow$ false
while not Finish **do**
  $a \leftarrow i$
  if t is front facing **then**
    $b \leftarrow i+1$
  else
    $b \leftarrow i-1$ ▷ flip direction of tracing
  end if
  project segment $[p_a, p_b]$ on t
  if projected segment intersects an edge e of t_j **then**
    $C \leftarrow C+t$ ▷ add t to the list
    $t \leftarrow$ triangle sharing e with t
  else
    $i \leftarrow b$
  end if
  if $i = i_0$ And $t = t_{i_0}$ **then** ▷ first point and triangle re-visited
    Finish $\leftarrow$ true ▷ stop tracing
  end if
end while

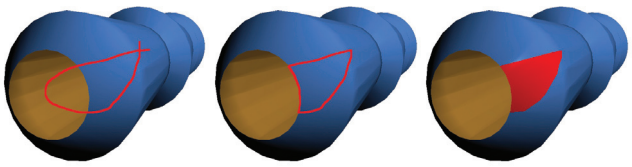


Figure 9: The user-drawn curve intersects a mesh boundary: The curve is traced along that boundary between the intersection points.

Because the traversal itself is performed on the mesh, we handle self-intersecting meshes successfully, as long as the connectivity between the triangles remains manifold. If we encounter a mesh boundary (Figure 9), we follow it until its next intersection with $\hat{C}$.

Given the curve represented by lists of triangles with barycentric coordinates and indices, we can split the mesh – by splitting the triangles along the curve. The triangles resulting from a split are marked as interior/exterior, depending on whether they project inside/outside of $\hat{C}$. To extract a selected region from the mesh, we perform a stack-based flood-fill on the mesh, which is initialized by the interior triangles adjacent to the selected starting point of the boundary. Propagation stops at triangles already marked as interior.

5 Applications and Results

We have tested our method on various data sets, including self-intersecting meshes. Figure 10 shows an example of selecting different parts of the Armadillo model, all from a single viewpoint.

In the following, we show three applications: applying textures, cutting for inspection and illustrations, and painting the surface with a brush. We classify these applications based on whether the selection is static or is transformed dynamically. Furthermore, we also analyze the performance of our approach.

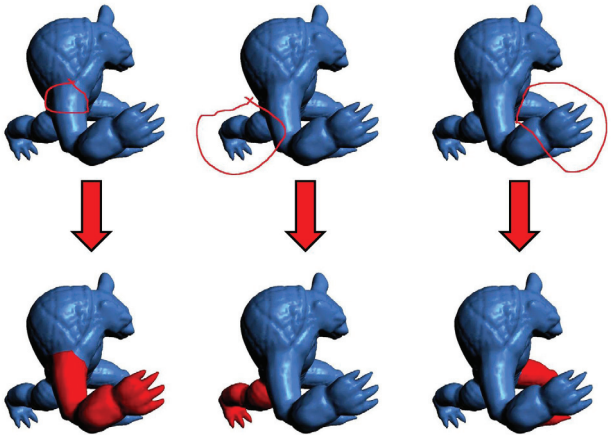
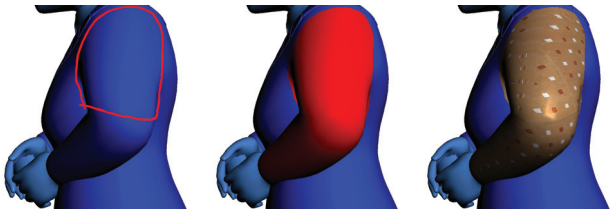


Figure 10: The selection tool permits selecting both an arm and any leg of the Armadillo from a single view.



(a) (b) (c)

Figure 11: Painting a texture on the surface: (a) The user draws a curve. (b) The selected region is marked. (c) A texture is associated with it. (e) The model with the textured arm (different viewpoint).

5.1 Static Selection

The user draws a curve once, which is mapped to bounding curves on the mesh, and a bounded region is selected by the user. To this selection, one of the following operations can be applied:

Texture Mapping. This is especially useful for surfaces of unsegmented models with texture coordinates already assigned. After selecting a region, the user can choose the texture image which is assigned to the segmented area, and further adjust the texture transformation, e.g., by dragging the texture inside the region. An example is shown in Figure 11.

Inspection with Cutting. Another application of our method is to explore models with significant depth complexity, such as in Figure 12. Many models are self-occluding, and some even contain numerous interleaving components (e.g., wires, tree branches, or blood vessels) that occlude the view of other parts from most or even all camera positions. Using our method, the user can quickly reveal the hidden areas by cutting away the occluding parts with simple curve sketching. Figure 13 shows an example where several candidate-for-selection regions are detected by drawing a single 2D curve. The user can move the selection point along the screen space curve to choose a region.

Quick Generation of Cutaways. Once experts have inspected a complex model, they may wish to show selected features to non-expert users. Cutaways (either as holes or just as transparencies) are a very good tool to expose and highlight details of complex objects. For centuries, drawings with cutaways have been used in medical instruction and biology. For example, The work of Burns et al. [3] shows that inner components can be revealed by manipulating the

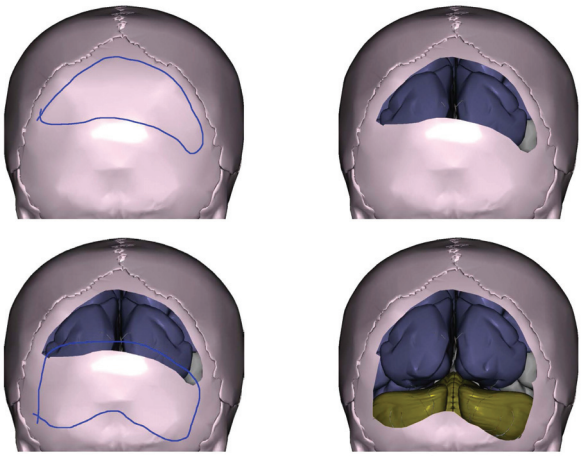


Figure 12: Inspecting the SKULL model. The user interactively widens the cut by drawing composite curves to better see the interior.

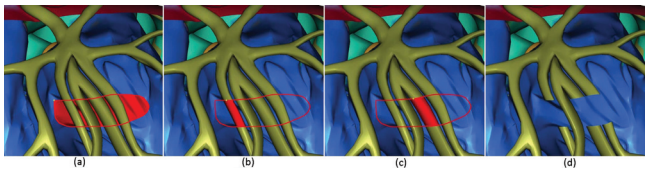


Figure 13: Inspecting the interior of the HEART. Different possible regions selected by the yellow selection point. Note that in (b) & (c) the figure shows the selection point on only one curve, but the user selected an additional curve to bound the red region. (d) three arteries cut.

visibility of the (occluding) outer layers. Our algorithm is ideally suited to quickly create such illustrations. Figure 1 shows a cutaway of the HEART model, and Figure 14 shows a transparent illustration, together with the steps used to create it. We were able to create each of these drawings in less than a minute, just from a single viewpoint. The key advantage of our method in this scenario is that thin structures might occlude only little, but add to understanding of shape. In consequence, it can be useful to leave them intact, like the thin blood vessels in the HEART model.

Automatic Cutaways. A cutaway shape can also be generated based on an (occluded) object of interest O that the user picks. To do so, we generate $\hat{C}$ automatically as the boundary of the projection of O onto the current view (e.g., using the convex hull of $\Pi(O)$), with some margin. In order to remove parts that extend outside the bounding curve in screen space, as explained in Sec. 2, each curve in $\mathbb{C}$ is treated as C, and a corresponding region is bounded by this curve only. The surface regions that occlude O are then marked and removed. Since regions bounded by entirely occluded curve boundaries are also be extracted and checked, we use depth peeling [7] to find occluded boundaries in $\mathbb{C}$ using the same algorithm.

5.2 Dynamic Selection

Since the outline and selected region is calculated in real time, our method also supports interactive region modification.

Continuous Painting. A perfect application for this mechanism is to paint on the surface by dragging a brush without being bothered by occlusions, as shown in Figure 15. Unlike the chameleon system [13] and the LayerPaint [10], our algorithm maps a screen-space curve to a surface boundary. Therefore, the brush does not have to cover the whole desired region. Figure 17 shows example cases:

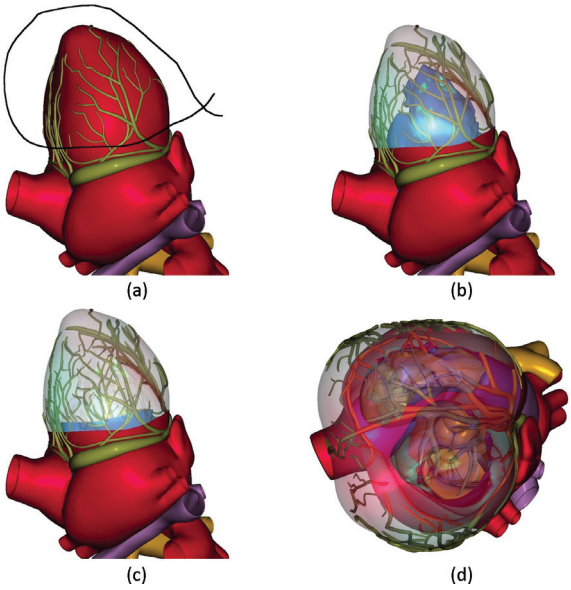


Figure 14: Generating a transparent illustration of the HEART in under a minute: (a) The user draws a curve, and the selected region is turned transparent (b). Tw other regions turned transparent to get the final illustration (c), also shown from another perspective (d).

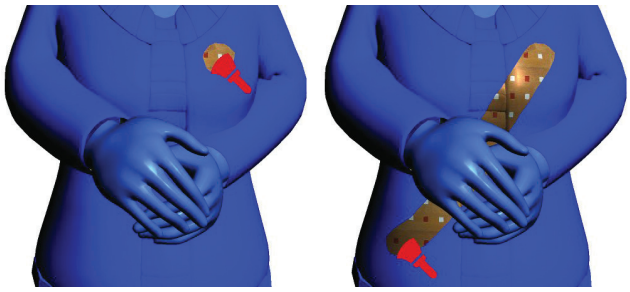


Figure 15: The user paints a texture with a brush onto the model.

The brush is moving along two layers at the same time (c, d), and the whole arm is textured because it is bounded by curves of $\mathbb{C}$ even if it is not entirely covered by the brush in screen space (e, f). Both previous methods do not handle such cases.

For this application, $\hat{C}$ is constructed and extended with the outline of the brush, usually a disk. We additionally need a representative point in the brush, which is responsible for selecting the painting layer in case the brush covers multiple layers, and for the starting frame 0 as with curve selection we require that point to lie on the curve $\hat{C}^0$ (the brush outline). When extending the painted region in screen space in consecutive frames i, $\hat{C}^i$ becomes the outer hull of the union of the brush region in frame i and the region bounded by $\hat{C}^{i-1}$ in screen space in order to avoid holes. We pick the new selection point p such that p is on the boundary curve of frame i-1 and $\Pi(p) \in \hat{C}^i \cap \hat{C}^{i-1}$. Figure 16 illustrates this with an example. Points P_0 and P_2' lie on both $\hat{C}^i$ and $\hat{C}^{i-1}$, and therefore either one of them can be the selection point.

5.3 Animating the Position of the Selection

Instead of extending the selected region as in painting, we can also move the boundary across the surface.

Animating Cutaways. An example of this are animated cutaways, where the cutaway shape is dragged along the surface, obliv-

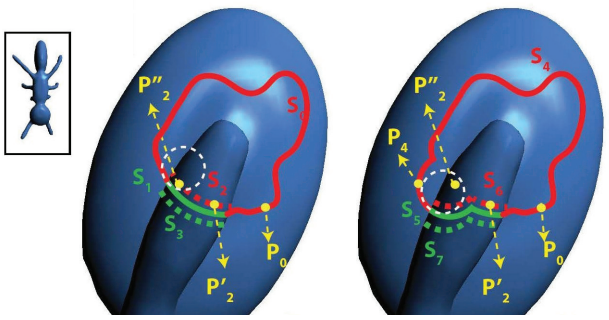


Figure 16: Occlusion-aware region extension by the example of brush painting. Left: Frame $i-1$: The brush is outlined in white. $\hat{C}^{i-1}$ projects to segments $S_{0..3}$ (only S_0 and S_1 are visible). Assume the selected region is bounded by $S_0 \cup S_2$. Right: Frame i: P_0 and P_2' project back to both $\hat{C}^i$ and $\hat{C}^{i-1}$, therefore tracing from either will get the intended region, which is bounded by $S_4 \cup S_6$. P_2'' and P_4, on the other hand, do not satisfy the condition.

ious to any occluding objects, as shown in Figure 18.

In each frame, the curve $\hat{C}$ is dragged by the user into a new position. For starting frame 0, the desired boundary C^0 is chosen using the selection point $\hat{p}^0$ on $\hat{C}^0$. Then, for subsequent frames i, the curves $\hat{C}^i$ are displaced by relative mouse input movement, and the selection points $\hat{p}^i$ move along with $\hat{C}^i$. At frame i, we project the selection point $\hat{p}^i$ to all corresponding p^i on the surface and choose the one geodesically closest to p^{i-1} to trace and find C^i. This ensures that the curve moves in an occlusion-aware fashion between these close "key frames".

5.4 Performance

curve radius		50 pixels			100 pixels		
model	# faces	trace	split	fill	trace	split	fill
BEAR	19K	3.2	0.9	0.9	10.1	4.7	3.3
BUNNY	70K	4.4	6.0	3.9	5.7	4.0	5.8
HEART	177K	9.2	7.9	10.2	11.5	5.7	10.1

Table 1: Runtimes (in ms) of our algorithm for 3 models and 2 curves covering varying area sizes (radii in pixels), are recorded for the main steps: tracing the candidate curves, splitting the triangles, and flood-filling. Projecting the curve is negligible (always <1 ms).

viewport	rad	trace	split	fill
400X400	25	2.7	1.4	52.6
800X800	50	5.4	1.5	52.8
1600X1600	100	13.5	1.9	53.4

Table 2: Runtime measures for the HAPPY model, with different viewport sizes (and different corresponding curve radii).

Table 1 shows the runtime of the algorithm on models of different sizes and varying curve sizes. The input is a 2D curve centered in the middle of a 512×512 pixel window, with radius either 50 or 100 pixels. The timings show that the tracing, triangle splitting, and marking are linear as expected. The recorded runtime of the projection on the GPU is always less than one millisecond in all experiments. However, CPU-GPU data transfers take between 15 to 30 ms. In a future implementation, the entire processing could be performed on the GPU, which would eliminate that bottleneck. Table 2 shows that the runtime of tracing the curves scales linearly with the viewport length, while the processing of the mesh (splitting and flood-filling) which takes most of the time, is mostly constant, as expected. The algorithm was implemented with C++ and OpenGL and tested on a Core2 Quad 2.4 GHz CPU with 4 GB RAM and GeForce GTX 680 GPU.

6 CONCLUSION

We proposed a novel selection method that enables users to perform operations on meshes without being blocked by, or having to remove occlusions of the mesh itself or from other objects. Our method even works for meshes which self-intersect and/or contain holes. We show that these surface subset selections can be created very easily, and the user can determine a preference between different choices of occlusion if they exist. These occlusion-aware selections can also be performed interactively, which enables artists to use them in state-of-the-art 3D design software. By adding our algorithm into such a product, many already included operations could be enhanced,such as extrusion, mesh simplification, mesh mixing, and many more. As the method is simple, it is especially convenient for implementing it as an add-on module in 3D modeling systems. Several different operations can be performed either on the selection boundary or on the surface subset marked by its interior: painting, cutting, modifying surface attributes (e.g., transparency), extrusion, deforming. We have demonstrated the first three as examples, as well as animating illustrative cut boundaries.

Limitations. Our implementation does currently not handle non-manifold meshes, where more then two triangles are connected by one edge, but it could be extended by performing the tracing, while maintaining a stack. This is not a severe limitation since the kind of meshes used as input usually are manifold. Currently, our implementation handles only triangular meshes, but it is trivial to extend it to handle any polygonal mesh.

Except in automatic cutaway generation and continuous painting applications, our algorithm also does not capture regions bounded by entirely occluded boundaries, as we assume it is not intuitive for the user. Those boundaries can be extracted using depth peeling [7] and then allow the user to select occluded curves as well. Depth peeling can be efficiently performed using the illustration buffer [4],

Another limitation is cutting through triangle soups; a commonly encountered artifact of isosurface meshing. Such triangles lack the structure of a connected manifold, which our method assumes. However, in case the triangle soup is just spurious and hidden, our cutting method can aid the user to remove it. Figure 19 shows an example, where the user temporarily cuts the (manifold) outer part of the mesh in order to reach and select the spurious primitives - with a tool capturing connected components enclosed in a rectangle. The selected components are removed and the cut is undone.

Future Work. We are currently extending our method to process other surface representations, such as point clouds with recovered connectivity, since that suffices for both curve tracing and bounded region marking. Using a discretized data structure for connectivity recovery, e.g., the one proposed by Radwan et al. [19], could permit those operations on implied connectivity of sensed data.

Aside from cutting and painting, other surface operations (e.g., extrusion, deformation, mesh simplification, etc.), as well as operations on surface subsets like mesh mixing, could be applied to regions selected with our tool. Converting the code into a plug-in of an existing 3D software (e.g., Blender or Maya) would allow this.

While one advantage of our method is that the final boundary is faithful to the curve sketched by the user, it is not unlikely that the user will want to smooth it out. Therefore, in order to enhance the usability of the tool, we will implement a postprocessing boundary smoothing phase on the mesh.

Our method could also be used for user-guided segmentation: Since the user-drawn 2D curve cuts the object, it could serve as initial 3D curve to fit with local optimization criteria, e.g., based on

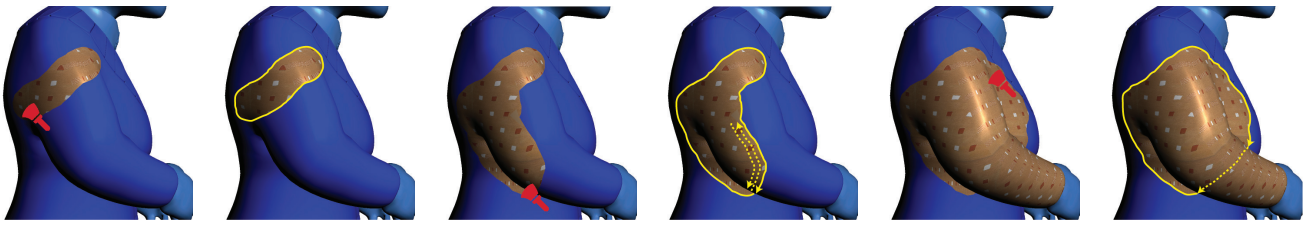


Figure 17: Several screen shots of a painting session, each followed by a yellow outline of the region boundaries (occluded segments dashed).

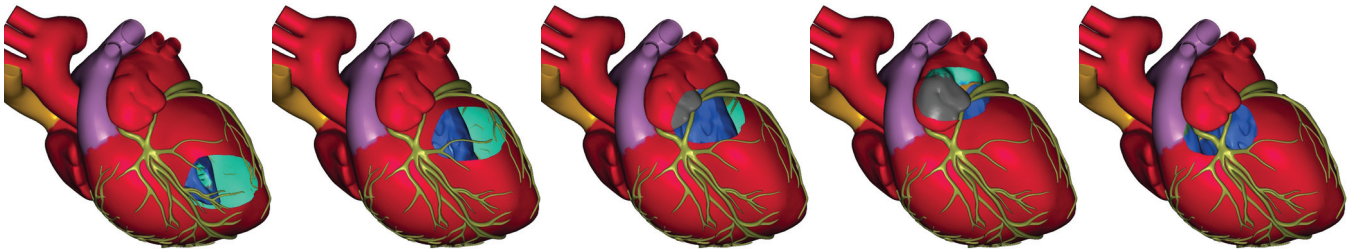


Figure 18: Animating the position of a cutaway: A few frames of a cut that is being dragged along the surface.

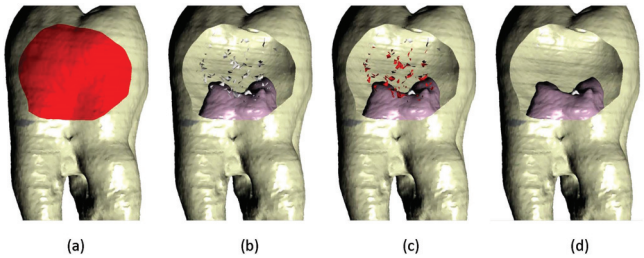


Figure 19: (a) A tooth mesh reconstructed from an isosurface of a CT scan, with the selected region shaded red, then cut (b), revealing spurious connected components. (c) Selected artifacts (with a different tool). (d) Cut undone after removal of artifacts.

concavity isolines as proposed by Au et al. [1].

References

[1] O. K.-C. Au, Y. Zheng, M. Chen, P. Xu, and C.-L. Tai. Mesh segmentation with concavity-aware fields. *Visualization and Comp. Graph., IEEE Trans. on*, 18(7):1125 – 1134, July 2011.

[2] M. Burns and A. Finkelstein. Adaptive cutaways for comprehensible rendering of polygonal scenes. *ACM Trans. Graph.*, 27(5):154:1–154:7, dec 2008. doi: 10.1145/1409060.1409107

[3] M. Burns, M. Haidacher, W. Wein, I. Viola, and E. Gröller. Feature emphasis and contextual cutaways for multimodal medical visualization. *Proceedings of Eurographics / IEEE VGTC Symposium on Visualization (EuroVis 2007)*, 2007.

[4] R. Carnecky, R. Fuchs, S. Mehl, and R. Peikert. Smart transparency for illustrative visualization of complex flow surfaces. *IEEE Trans. on Vis. and Comp. Graph.*, July 2012.

[5] H. D., R. W., P. A., and R. J. Parts of recognition. *Cognition 18*, 1984.

[6] E. Eisemann, S. Paris, and F. Durand. A visibility algorithm for converting 3d meshes into editable 2d vector graphics. *ACM Trans. Graph.*, 28(3):83:1–83:8, jul 2009. doi: 10.1145/1531326.1531389

[7] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA corporation, 2001.

[8] L. Fan, L. Lic, and K. Liu. Paint mesh cutting. *Computer Graphics Forum*, 30(2):603–612, Aprit 2011.

[9] L. Fan, M. Meng, and L. Liu. Sketch-based mesh cutting: A comparative study. *Graphical Models*, 2012.

[10] C.-W. Fu, C.-W. Fu, and Y. He. Layerpaint: A multi-layer interactive 3d painting interface. *CHI 2010*, 2010.

[11] T. Funkhouser, M. Kazhdan, P. Shilane, P. Min, W. Kiefer, A. Tal, S. Rusinkiewicz, and D. Dobkin. Modeling by example. *ACM SIGGRAPH 2004*, 2004.

[12] S. M. Hoffman D. D and. Salience of visual parts. *Cognition 63*, 1997.

[13] T. Igarashi and D. Cosgrove. Adaptive unwrapping for interactive texture painting. *Proceeding I3D '01 Proceedings of the 2001 symposium on Interactive 3D graphics*, pp. 209–216, 2001.

[14] Z. Ji, L. Liu, Z. Chen, , and G. Wang. Easy mesh cutting. *Computer Graphics Forum*, 2006.

[15] Y. Lee, S. Lee, A. Shamir, D. Cohen-Or, and H.-P. Seidel. Mesh scissoring with minima rule and part salience. *Computer Aided Graphics Design*, 2005.

[16] W. Li, L. Ritter, M. Agrawala, B. Curless, and D. Salesin. Interactive cutaway illustrations of complex 3d models. *ACM Trans. Graph.*, 26(3), jul 2007. doi: 10.1145/1276377.1276416

[17] J.-M. Lien and N. M. Amato. Approximate convex decomposition of polyhedra. *CAGD*, 25(7):503–522, 2008.

[18] M. Meng, L. Fan, and L. Liu. icutter: a direct cut-out tool for 3d shapes. *Comp. Anim. & Virt. Worlds*, 22(4), August 2011.

[19] M. Radwan, S. Ohrhallinger, and M. Wimmer. Efficient collision detection while rendering dynamic points. *Proc. of the 2014 Graphics Interface Conference*, pp. 25–33, May 2014.

[20] O. van Kaick, N. Fish, Y. Kleiman, S. Asafi, and D. Cohen-Or. Shape segmentation by approximate convexity analysis. *ACM Transactions on Graphics*, 34(1), November 2014.

[21] Y. Zheng and C.-L. Tai. Mesh decomposition with cross-boundary brushes. *Computer Graphics Forum (In Proc. of Eurographics 2010)*, 2010.

[22] Y. Zheng, C.-L. Tai, and O. K.-C. Au. Dot scissor: A single-click interface. *IEEE Transactions on Visualization and Computer Graphics*, 18(8):1304–1312, 2012.