CS 554 Final Project Report

Adaptive Local Remeshing for Smooth Silhouette on 3D Triangle Meshes

Rachel Xing

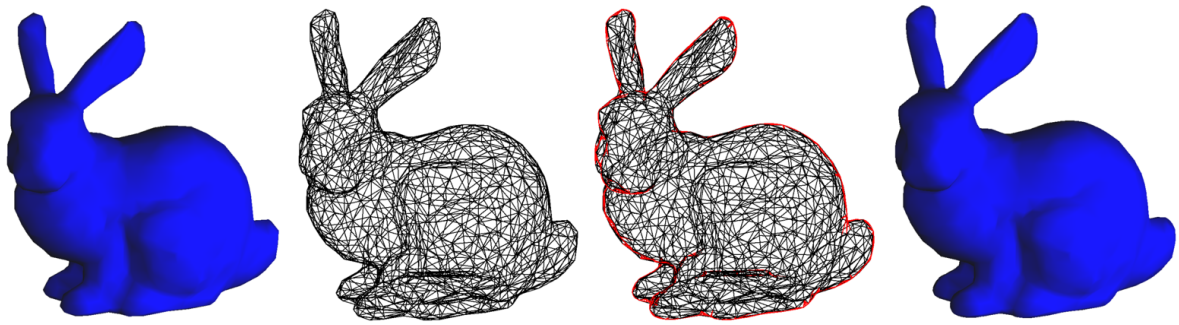

Fig. 1: Coarse Stanford Bunny model with 2,000 faces. From left to right: (1) Shaded mesh rendering, (2) Wireframe rendering, (3) Wireframe with newly added geometry (highlighted in red) from the adaptive silhouette smoothing pipeline, and (4) Shaded mesh rendering after remeshing by the adaptive silhouette smoothing pipeline.

1 Introduction

The smoothness of a model's silhouette is important for artistic applications of computer graphics such as pen-and-ink rendering. It always requires a high-resolution mesh so that there are enough details around the silhouette regions to appear smooth in real-time rendering. However, in the context of triangular meshes, the high-resolution versions of complex models can contain ten thousands to millions of triangles. Such abundant mesh data can significantly increase the computational load and hurt real-time rendering performance, especially on consumer-level hardware. To address this problem, the paper "**Silhouette Smoothing for Real-Time Rendering of Mesh Surfaces**" [6] proposes an efficient real-time local remeshing pipeline for smooth silhouette so that details are only added where they are needed. With this pipeline, we can achieve a smooth silhouette even with a coarse input mesh. The original pipeline also includes procedures for detecting and smoothing feature edges, but since the major focus of my final project is the visual silhouette, I skipped that part and focused only on the partial pipeline related to silhouette edges. Thus, in this project, I reimplemented the real-time local remeshing for the smooth silhouette described in the paper and successfully integrated it into the C++-based graphics interface learnply that we've used in this course. To evaluate my implementation, I compared the visual quality of the smoothed silhouettes on coarse meshes after local remeshing with those from high-resolution meshes, evaluated the smoothed silhouette in pen-and-ink rendering, and measured its real-time performance.

2 Background

Line drawing is one of the most basic forms of artistic expression, so it plays an essential role in non-photorealistic rendering. One of its most important components is the silhouette, which defines the basic shape of an object and ensures visual coherence from a global view. In pen-and-ink rendering, the smoothness of the silhouette directly determines the quality of rendering results. In Assignment #3, we implemented both edge-based and face-based silhouette extraction algorithms. We observed that even with the face-based algorithm that can interpolate to approximate the segments of the silhouette across the triangle faces, the extracted silhouette can still exhibit visual clutters like discontinuities and mismatches (Figure 2).

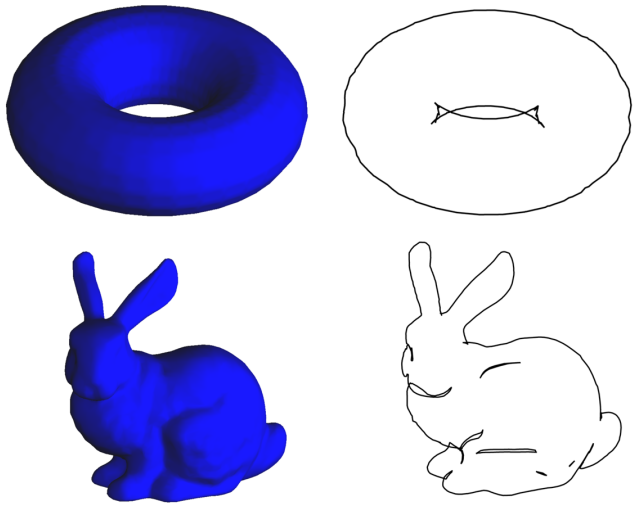

Fig. 2: Rendering of (left) shaded meshes and (right) extracted silhouettes using a face-based silhouette detection algorithm for the torus model (top, 4,608 triangles) and the Stanford bunny model (bottom, 10,000 triangles). With relatively coarse meshes, the extracted silhouettes exhibit obvious discontinuities and mismatches.

The most straightforward approach to improve the silhouette extraction accuracy is to use a finer mesh. However, since high-resolution versions of most complicated models usually contain ten thousands to millions of triangles, it can significantly increase computational cost and introduce lag during real-time rendering. Also, abundant mesh data has the risk of causing "out of memory" issues for a graphics program.

• *Rachel Xing is with Oregon State University. E-mail: xingru@oregonstate.edu*

Thus, we need a method that can balance silhouette visual coherence with rendering efficiency and computational load.

3 Previous Work

The paper I implemented is related to two major topics, adaptive level-of-detail (LOD) and silhouette smoothing. An adaptive LOD system is a technique that reduces the number of triangles in less important regions of a mesh while allocating more detail in more important regions like the silhouette. It helps achieve good visual quality for distant views with significantly fewer triangles than the original mesh to help real-time rendering efficiency while ensuring appropriate detail for close-up views. At the time of the paper's publication (2004), one of the most popular adaptive LOD systems was the view-dependent LOD system [7]. It can adjust mesh detail based on viewing factors such as viewing distance, lighting, and visibility. However, regardless of the type of adaptive LOD systems, most of them rely on progressive meshes and tend to have a fine mesh as input [2, 4]. This is because coarse meshes provide little room for simplification and carry a risk of oversimplification. Also, they require additional refinement processing that can bring a lot of memory and computational costs.

Besides, at the time of publication, most mesh smoothing techniques focused on global surface smoothing. Two prominent examples were Laplacian smoothing [3] and the Point-Normal (PN) triangle method [5]. Both could be applied to coarse meshes. Laplacian smoothing is the most straightforward way to perform mesh surface smoothing by moving each vertex to the average position of its neighbors. Since no additional geometries are created, it has a risk of blurring the original mesh and degrading visual quality. The PN triangle method tessellates each triangle face of the input mesh into a curved surface using normal-based Bezier patches to approximate smooth surface curves. While it could achieve better quality by adding new geometries for smoothing compared to Laplacian smoothing, since all triangles receive the same level of subdivision, the silhouette could still appear polygonal without sufficiently high subdivision levels [6]. Also, with high subdivision levels, it may introduce abundant unnecessary triangles that could hurt real-time rendering performance, especially if further processing is needed.

The local remeshing pipeline in this paper [6] intends to serve as an alternative to view-dependent LOD systems. Unlike previous approaches that require either fine input meshes or global surface reconstruction, this method performs local mesh refinement specifically targeted at silhouette regions. By focusing computational resources only on visually critical areas, it can achieve smooth silhouettes while maintaining a good real-time rendering efficiency.

4 Methods

The adaptive local remeshing pipeline for smooth silhouette proposed in the paper [6] consists of the following four major stages:

1. **Silhouette detection**: Perform a visibility test on each triangle and mark the silhouette triangles.
2. **Computing silhouette bridges**: For each silhouette triangle, identify the two edges intersected by the silhouette and compute the corresponding silhouette bridges using Hermite interpolation.
3. **Compute silhouette segments**: Compute silhouette points based on silhouette bridges and use Hermite interpolation to construct a smooth silhouette segment connecting each pair of silhouette points.
4. **Remeshing**: Along the silhouette segment for each silhouette triangle, adaptively determine the number of sampled points based on the curvature and its projected length. Then, perform local remeshing by forming a new triangulation between the sampled points and the original triangle.

To integrate this pipeline into our learnply graphics interface, I added a new struct `RemeshData` to the `Polyhedron` class for the input mesh to store temporary mesh data resulted from the pipeline for the current frame. Whenever the camera direction changes, any previously generated geometry is deleted from the struct, and the pipeline is re-executed to update it.

4.1 Detection of Silhouette Triangles

This local remeshing pipeline begins by iterating through each triangle in the original mesh to identify silhouette triangles, which are triangles located at the view-dependent silhouette regions of the mesh that contain both front-facing and back-facing vertices. If the current triangle is not a silhouette triangle, the pipeline leaves it unchanged. Otherwise, it continues with the subsequent processing steps. The silhouette triangle is determined by testing the visibility of each of its vertices and checking whether they all share the same visibility. The visibility test for each vertex V is defined by computing $N \cdot (V - E)$, where N is the vertex normal vector, E is the eye position vector, and their difference $V - E$ is the eye vector. Vertex V is considered to be front-facing if its visibility value is greater than or equal to 0; otherwise, it is back-facing.

4.2 Computation for Silhouette Bridges and Silhouette Segments

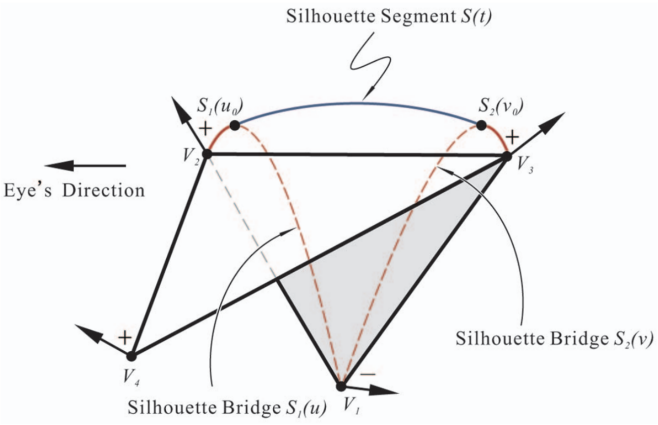

Fig. 3: Silhouette bridges $S_1(u)$, $S_2(v)$ and silhouette segment $S(t)$ for a silhouette triangle [6]. Points $S_1(u_0)$ and $S_2(v_0)$ are silhouette points with a visibility value of 0 on the two silhouette bridges, which serve as the endpoints of the silhouette segment. Only the portion of each silhouette bridge that connects the silhouette segment and the base edge V_2V_3 is used in later local remeshing steps (marked by solid lines), and other parts are virtual (marked by dashed lines).

For each silhouette triangles, we target the two silhouette edges that have the vertices of different visibility. In the example silhouette triangle (Figure 3), these two edges are V_1V_2 and V_1V_3. Then, perform the Hermite interpolation on each edge to approximate the local surface protrusion. The resulting curves are called silhouette bridges, which are $S_1(u)$ for edge V_1V_2 and $S_2(v)$ for edge V_1V_3 in Figure 3. These bridges serve as virtual scaffolds for the subsequent approximation of the true silhouette segment. The resulting silhouette bridge $S(u)$ that connects V_1 and V_2 is given by the following cubic Hermite curve function:

$$S(u) = (2V_1 - 2V_2 + T_1 + T_2)u^3 - (3V_1 - 3V_2 + 2T_1 + T_2)u^2 + T_1u + V_1, \ u \in [0,1]$$

where T_1 and T_2 are tangent vectors at V_1 and V_2 computed by:

$$T_1 = \hat{T}_1 * |T_1|, \qquad T_2 = \hat{T}_2 * |T_2|$$

Here, $\hat{T}_1$ and $\hat{T}_2$ are unit tangent vectors at two vertices computed by projecting the vector $\overrightarrow{V_1V_2}$ on the tangent plane at V_1 and V_2 using their respective normal vectors N_1 and N_2 as reference:

$$\hat{T}_1 = (V_2 - V_1) - [(V_2 - V_1) \cdot N_1] \cdot N_1$$

$$\hat{T}_2 = (V_2 - V_1) - [(V_2 - V_1) \cdot N_2] \cdot N_2$$

The tangent vector magnitudes at V_1 and V_2 can be computed using various methods, and the method used in this paper extracts the tangent lengths from a circular arc approximation:

$$L_1 = \frac{2|V_2 - V_1|}{1 + \cos\theta_1}, \qquad L_2 = \frac{2|V_2 - V_1|}{1 + \cos\theta_2}$$

Once the cubic Hermite curve functions for silhouette bridges are established, we compute the silhouette points that are points on each bridge where the surface normal is perpendicular to the viewing direction and a part of the smoothed silhouette. To find the silhouette point on a given silhouette bridge, we first interpolate the surface normal along the silhouette bridge using linear interpolation. Assume there is a silhouette bridge $S(u)$ that connects V_1 and V_2, the interpolated normal along $S(u)$ is:

$$\tilde{N}(u) = (1-u)\cdot N_1 + u \cdot N_2, \;\; u \in [0,1]$$

We apply the same linear interpolation to the eye vectors D_1 and D_2 at vertices V_1 and V_2:

$$\hat{D}(u) = (1-u)\cdot D_1 + u \cdot D_2, \;\; u \in [0,1]$$

The silhouette point $S(u_0)$ is then found by solving the visibility equation:

$$h(u) = \hat{D}(u) \cdot \tilde{N}(u) = 0$$

This equation must have a unique solution u_0 in the range of $[0,1]$ because $h(u)$ has different signs at $u = 0$ and $u = 1$. This is guaranteed since the silhouette bridge connects vertices with different visibility, as described in section 4.1. Then, the resulting $S(u_0)$ is the silhouette point on the silhouette bridge $S(u)$.

Once the silhouette points on both silhouette bridges of a silhouette triangle have been determined, along with their corresponding normal vectors, we perform Hermite interpolation between these silhouette points to construct the silhouette segment. The resulting silhouette segment is a part of the smoothed silhouette.

4.3 Adaptive Local Remeshing

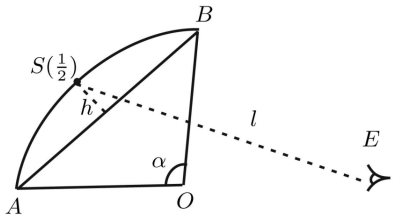

Fig. 4: Adaptive sampling of points along the silhouette segment based on local curvature and projected length [6].

To perform adaptive local remeshing based on the computed silhouette components, we first determine the number of sample points needed to construct each silhouette segment. Since the silhouette segment approximates a circular arc, this number is calculated based on the local curvature and projected length of the segment (Figure 4): For a silhouette segment $S(u), u \in [0,1]$ with endpoints A and B, we first calculate the midpoint $S(\frac{1}{2})$ and use it to approximate the height h of the silhouette segment projected onto the plane perpendicular to line AB. We then compute the curvature indicator $r = \sin(\alpha/2) = \sqrt{1-\cos(\alpha)/2}$, where α is the center angle of $S(u)$, and $cos(\alpha)$ is approximated by the dot product of the normal vectors at vertices A and B. A larger r indicates a greater angle between the normals, meaning the surface bends more sharply between A and B, and more sampled points are needed for smoothness. The final number of sample points for the silhouette segment $S(u)$ is computed by:

$$n_s = max(\left\lceil \frac{1}{2} \cdot \sqrt{h \cdot r / e} \right\rceil, 4)$$

where e is the allowed tolerance in pixels. The original paper suggests setting it in the range $[0.5, 2]$. In my implementation, it is set to 0.5 to ensure optimal silhouette smoothness during rendering.

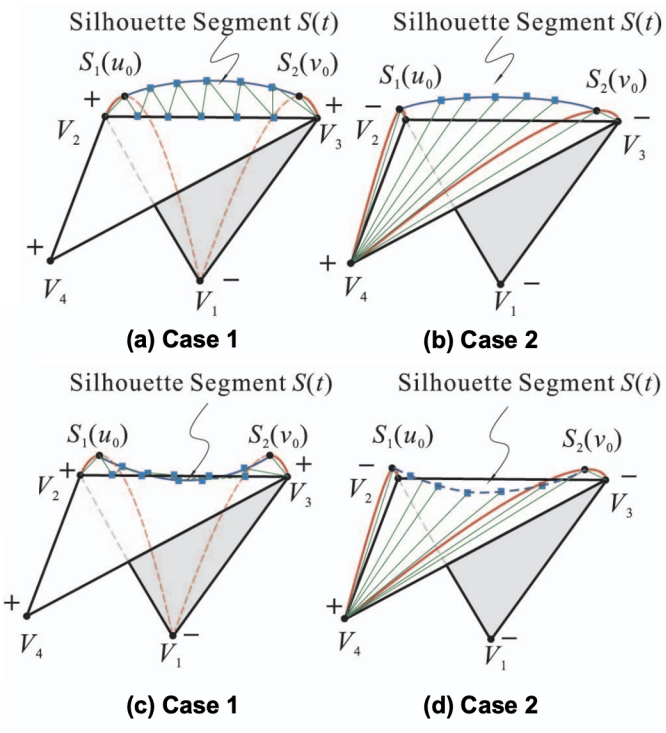

Fig. 5: Four scenarios of local remeshing on a silhouette triangle: convex silhouette segments in (a) and (b), and concave silhouette segments in (c) and (d) [6]. The choice of two remeshing strategies depends on whether the silhouette segment lies in front of the base edge V_2V_3.

Finally, after obtaining all sampled points along the silhouette segment, we create a new triangulation between the silhouette segment and the original silhouette triangle. As shown in Figure 5, there are two cases depending on the relative position of the silhouette segment with respect to the base edge (i.e., the edge connecting the two vertices with the same visibility in the silhouette triangle). If the silhouette segment lies behind the base edge, we form triangulation between the sampled points along the silhouette segment and sampled points along the base edge, as shown in Figure 5 (a) and (c). If the silhouette segment is in front of the base edge, we form triangulation by connecting the sampled points along the silhouette segment to the third vertex of the silhouette triangle, as depicted in Figure 5 (b) and (d).

4.4 Evaluation

There are three major steps for validating my implementation:

1. Compare the smoothness of silhouettes detected by the face-based silhouette algorithm on the model with silhouettes extracted by the pipeline (i.e., the set of silhouette segments) on the same model after local remeshing.
2. Compare the visual appearance of the silhouette between coarse models with silhouette smoothing and fine models without silhouette smoothing in shaded mesh rendering.
3. Evaluate real-time rendering efficiency in frames per second (FPS) between coarse models with silhouette smoothing and finer models without silhouette smoothing.

All test models were selected from those used in this course: Torus, Stanford Bunny, Happy Buddha, Feline, and Dragon. The coarse and fine versions of these models were generated using MeshLab [1]. For the coarse versions, models were processed using quadric edge collapse decimation with default settings. Their face counts were reduced to the

point where it was coarse enough, but no critical visual details were lost from the original meshes. For the fine versions, models were processed using Loop's subdivision surface algorithm with 3 iterations. Detailed information about the original meshes and their coarse and fine versions is provided in the table 1.

Table 1: Models with their coarse and fine versions used in this project

	Number of Triangles		
Models	Original Mesh	Coarse Mesh	Fine Mesh
Torus	9,216	1,000	56,160
Stanford Bunny	10,000	800	60,272
Feline	10,000	3,000	50,386
Dragon	20,000	4,000	74,744
Happy Budda	20,000	5,000	88,366

5 Results

Due to page limitations, the resulting images are included in the appendix on the last two pages. Figure 6 demonstrates that this adaptive local remeshing pipeline can effectively generate smooth silhouettes by approximating silhouette segments for each silhouette triangle on coarse models. Compared to silhouettes directly extracted using the face-based silhouette detection algorithm, the pipeline-generated silhouettes exhibit fewer zigzag patterns and improved overall coherence. This improvement is especially obvious in areas with significant protrusions, such as the ears and tails of the bunny and feline models, and the horns of the feline and dragon models.

Figure 7 presents how this adaptive local remeshing pipeline smooths visual silhouettes in mesh rendering. The remeshing results on coarse models provide visually comparable smooth silhouettes to those inherent in fine models. That means we can achieve similar visual coherence for silhouettes using models with thousands of faces through this pipeline without the need for a fine input mesh of ten times or more faces or additional efforts for global reconstruction.

The pipeline was tested on my own laptop with an Intel Core Ultra 9 185H CPU, NVIDIA GeForce RTX 4060 GPU, and 32GB DDR5 memory. The performance results shown in Table 2 show that coarse meshes with silhouette smoothing can achieve significant improvements in real-time rendering efficiency compared to direct rendering on fine meshes. Based on my subjective experience, rendering coarse meshes with silhouette smoothing performs smoothly overall. It only has minor lag when rendering more complex models like the dragon and happy Buddha models. In contrast, rendering fine meshes exhibits significant lag in runtime regardless of geometric complexity.

Table 2: Performance comparisons of rendering with fine meshes and coarse meshes with Silhouette Smoothing

	Real-time Rendering Efficiency in FPS	
Models	Fine Mesh	Coarse Mesh with Silhouette Smoothing
Torus	14∼15	57∼60
Stanford Bunny	15∼16	22∼25
Feline	15∼16	21∼22
Dragon	10∼11	13∼15
Happy Budda	6∼7	10∼13

6 Conclusion

In this project, I successfully implemented the adaptive local remeshing pipeline described in "Silhouette Smoothing for Real-Time Rendering of Mesh Surfaces" [6] and integrated it into our learnply graphics interface. The results show that this pipeline can effectively extract smoother and more visually coherent silhouettes from models that exhibit polygonal artifacts. Also, the remeshing pipeline allows us to achieve silhouette smoothness with a coarse input mesh comparable to a fine mesh. It can help significantly improve real-time rendering performance by eliminating the need for either fine input meshes or global surface reconstruction.

However, there is still a lot of room for improvement in my implementation. First, I found that the current visibility testing cannot detect the full visual silhouette. Although the smoothness of the extracted silhouette has been improved by the pipeline, it still does not perfectly align with the actual visual silhouette. This is because silhouettes are not just defined by local depth discontinuities, so the visibility test can also identify edges that are not part of the true silhouette (e.g., sharp corners or any significant internal geometric features). Therefore, additional metrics may be needed to ensure accurate extraction of the actual visual silhouette boundaries. Then, rendering efficiency could be further improved by leveraging GPU computing power. In the original paper, the authors allocated the part for silhouette smoothing on the CPU and the part for rendering on the GPU using shaders to achieve additional speedup. Also, since the pipeline involves extensive vector computations, it may benefit from parallel programming and SIMD. Besides, the adaptive sampling step for points along silhouette segments might be improved by adopting the idea of view-dependent LOD. Currently, the sampling is based solely on local geometric information, but for a further viewing distance, fewer samples on the silhouette segment may be sufficient even in areas with significant curvature.

References

[1] P. Cignoni, M. Callieri, M. Corsini, M. Dellepiane, F. Ganovelli, and G. Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In V. Scarano, R. D. Chiara, and U. Erra, eds., *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008. doi: 10.2312/LocalChapterEvents/ItalChap/ItalianChapConf2008/129-136 3

[2] T. K. Heok and D. Daman. A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.*, pp. 70–75, 2004. doi: 10.1109/CGIV.2004.1323963 2

[3] L. R. Herrmann. Laplacian-isoparametric grid generation scheme. *Journal of Engineering Mechanics-asce*, 102:749–907, 1976. doi: 10.1061/JMCEA3.0002158 2

[4] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, p. 99–108. Association for Computing Machinery, New York, NY, USA, 1996. doi: 10.1145/237170.237216 2

[5] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell. Curved pn triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, I3D '01, p. 159–166. Association for Computing Machinery, New York, NY, USA, 2001. doi: 10.1145/364338.364387 2

[6] L. Wang, C. Tu, W. Wang, X. Meng, B. Chan, and D. Yan. Silhouette smoothing for real-time rendering of mesh surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):640–652, 2008. doi: 10.1109/TVCG.2008.8 1, 2, 3, 4

[7] J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *Proceedings of Seventh Annual IEEE Visualization '96*, pp. 327–334, 1996. doi: 10.1109/VISUAL.1996.568126 2

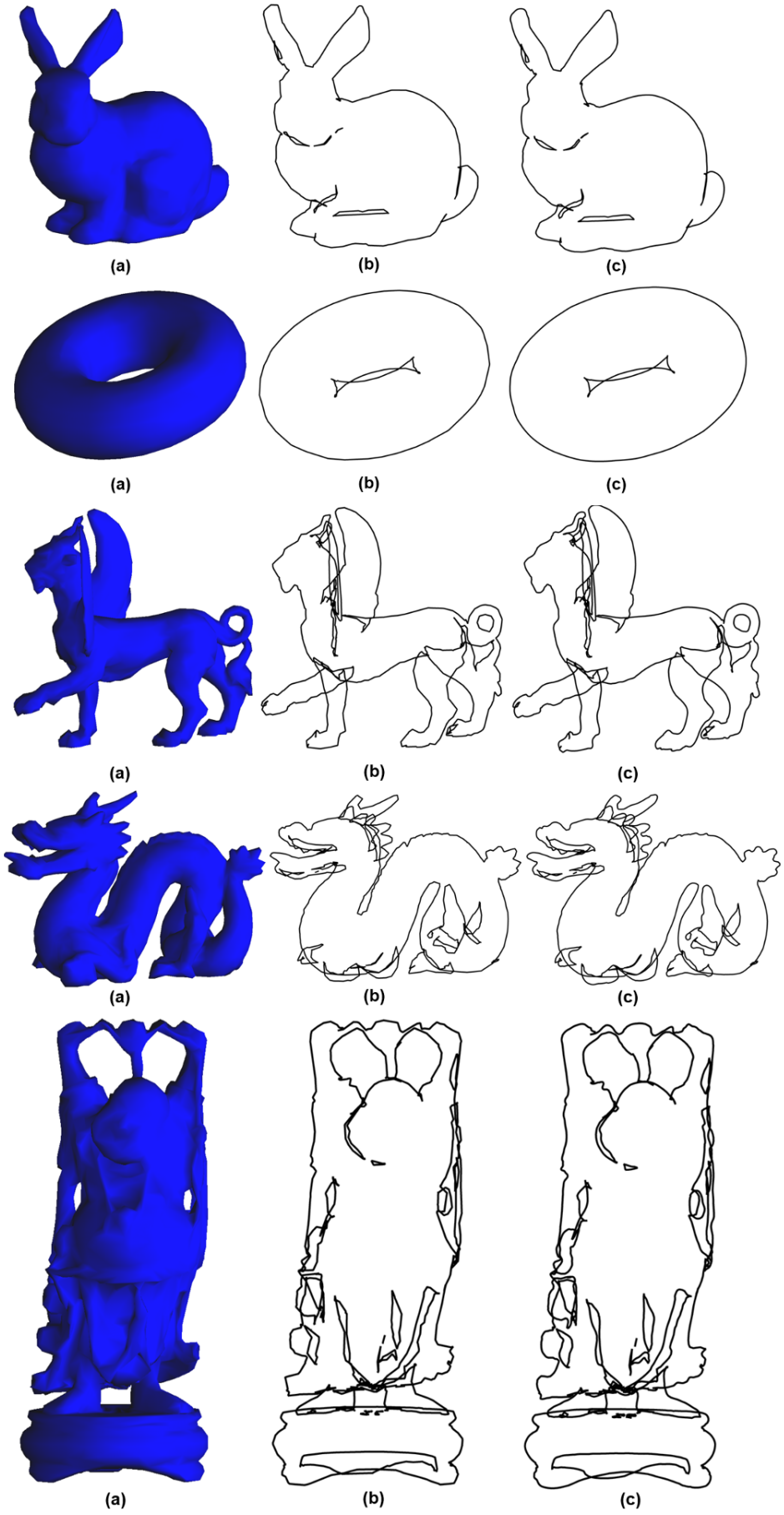

Fig. 6: Comparison of silhouette extraction by different approaches on coarse models: (a) Shaded mesh rendering, (b) Silhouette extracted using the face-based silhouette detection algorithm, and (c) Smoothed silhouette extracted by the implemented adaptive remeshing pipeline.

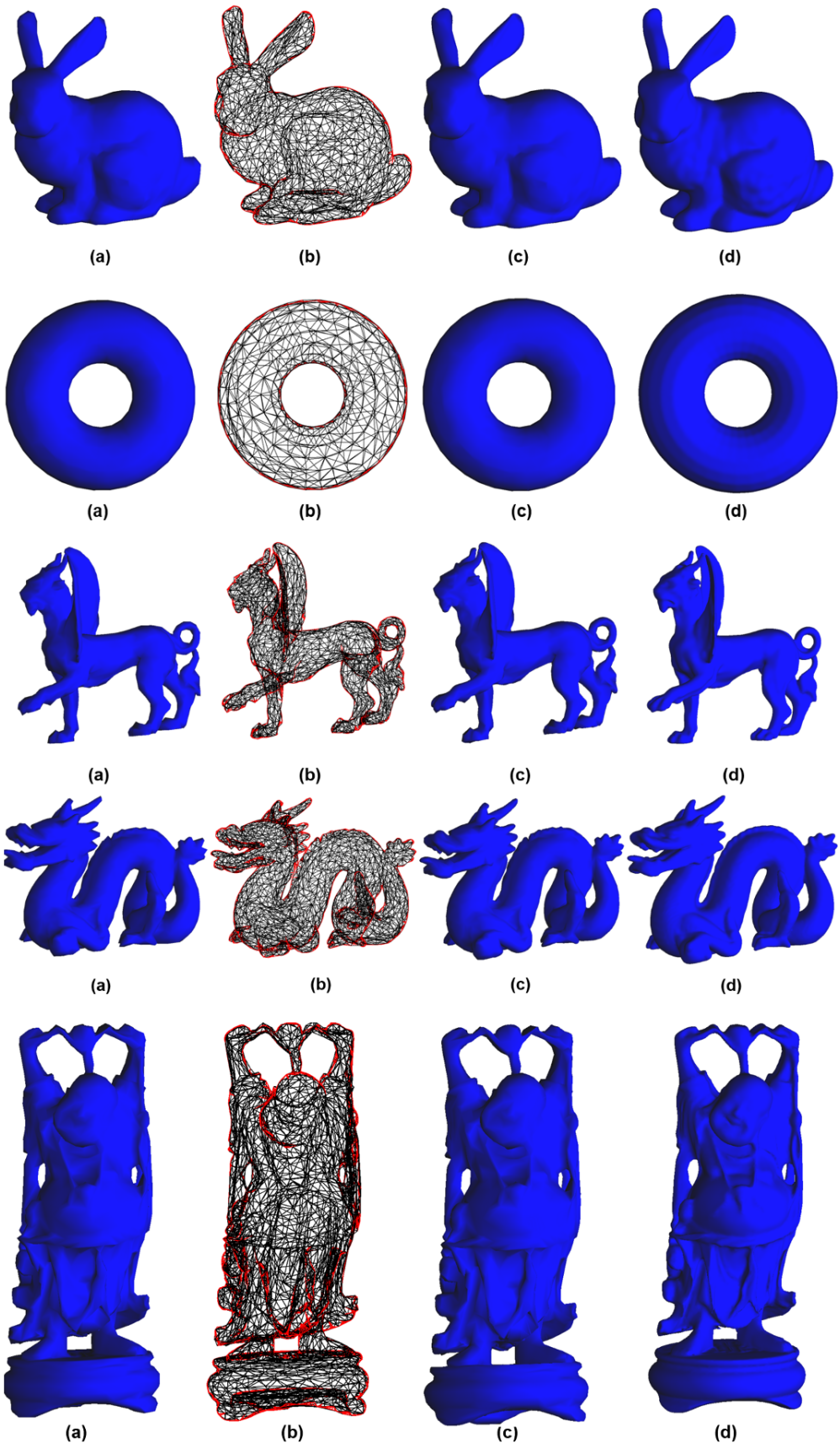

Fig. 7: (a) Shaded mesh rendering of coarse models, (b) Wireframe rendering after silhouette smoothing with newly added geometry highlighted in red, (c) Shaded rendering of coarse models with smoothed silhouettes, and (d) Shaded rendering of fine models.