Eurographics Symposium on Rendering (2006)
Tomas Akenine-Möller and Wolfgang Heidrich (Editors)

Silhouette Texture

Hongzhi Wu[1,2] Li-Yi Wei[1] Xi Wang[1] Baining Guo[1]

[1]Microsoft Research Asia [2]Fudan University

Abstract

Using coarse meshes with textures and/or normal maps to represent detailed meshes often results in poor visual quality along silhouettes. To tackle this problem, we introduce silhouette texture, a new data structure for capturing and reconstructing the silhouettes of detailed meshes. In addition to the recording of color and normal fields in traditional methods, we sample information that represents the original silhouettes and pack it into a three dimensional texture. In the rendering stage, our algorithm extracts relevant information from the texture to rebuild the silhouettes for any perspective view. Unlike previous work, our approach is based on GPU and could achieve high rendering performance. Moreover, both exterior and interior silhouettes are processed for better approximation quality. In addition to rendering acceleration, our algorithm also enables detailed silhouette visualization with minimum geometric complexity.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture: Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Texture;

Keywords: silhouette, visibility, occlusion culling, graphics hardware, real-time rendering, image-based rendering

1. Introduction

Silhouette is arguably one of the most important visual cues for conveying object shapes. However, rendering silhouettes accurately can be expensive, especially for complex geometric models such as scanned sculptures [LPC*00] or synthetic gaming assets [UT04]. As a result, many interactive applications today such as games still utilize crude polygonal representation of objects (e.g. see screen shots of *World of Warcraft* or *Grand Theft Auto*), resulting in jagged edges around silhouettes. This phenomenon is quite prevalent, even given the computation power of today's commodity graphics hardware, capable of rendering models with pixel-sized triangles in real time [LH04].

We present a technique to render complex polygonal models with accurate silhouettes with a fraction of cost for rendering the original models. Our core idea is a new data structure for capturing and later reconstructing the silhouettes of detailed meshes, called *silhouette texture*. In addition to color and normal fields, our silhouette texture incorporates information that represents the silhouettes of the original mesh as a special *visibility function*, defined over every point on a coarse outer hull around the original mesh. In the rendering stage, we extract this visibility information from our silhouette textures to rebuild the original silhouette according to desired viewing parameters.

Our technique is inspired by silhouette clipping [SGG*00], but we utilize a completely different data structure to allow a GPU-friendly implementation. In particular, our visibility function representation is similar to the one in horizon mapping [Max88, SC00], but instead of self-shadowing, we re-create and apply the basic idea in a novel way for storing visibility information. In addition, our technique handles interior silhouettes which is not considered in [SGG*00].

An immediate application of our algorithm is fast rendering of complex polygonal models, while preserving visual faithfulness of both silhouettes and interior shading. The speed improvement of our algorithm comes from the following simple observation. Most commodity GPUs today have higher pixel than vertex processing power (with a performance ratio about 10). The major reason behind this ratio is that commercial GPUs are targeted for games and benchmarks, which usually have $\sim$10:1 ratios for rendered scene pixels to vertices. Unfortunately, this ratio is sub-optimal for high-resolution polygonal meshes which usually have more rendered vertices than pixels. Our approach bridges this gap

outer hull + normal map
1000 vertices

original mesh
40002 vertices 308 fps

our result
1000 vertices 725 fps

Figure 1: Silhouette texture rendering. Our silhouette texture significantly speeds up rendering speed while faithfully reproduce both interior and exterior silouettes. All frame-rates reported in this paper are measured on a NVIDIA Geforce-7800-GT chip with a viewport size 512×512.

by reducing the number of rendered vertices at the expense of more complex pixel shading, resulting in a more balanced work load between vertex and pixel processing units.

In addition, our explicit knowledge of both interior and exterior silhouettes enables additional applications, such as silhouette-based visualizations. The major advantage of our approach over previous visualization methods is that we are able to draw high quality silhouettes from a coarse hull, instead of requiring a detailed input mesh.

The contributions of this paper include:

- Reconstructing high quality, smooth silhouettes from a silhouette texture for any perspective view, incorporating both interior and exterior silhouettes.
- Compression from original 4D silhouette information into 3D via a novel single-lobe observation, with the additional benefit of allowing native hardware filtering and anti-aliasing.
- The performance tuning ability of our silhouette texture for load balancing between vertex and fragment units, with high performance achieved via a GPU-based per-pixel shading algorithm.
- Additional silhouette-based applications enabled by our technique, such as contour visualization.

2. Previous Work

Silhouettes for perception and visualization Silhouettes have long been recognized as one of the most important visual cues for shape perception [Koe84]. Mathematically, silhouettes are points $\{p \mid \vec{n}(p) \cdot \vec{v}(p) = 0\}$ on a 3D object where $\vec{n}(p)$ is the normal and $\vec{v}(p)$ the vector from p to the eye point. [DFRS03] recommended using suggestive contours for shape illustration, incorporating points $\{p\}$ where $\vec{n}(p) \cdot \vec{v}(p)$ is a positive local minimum instead of being zero. This concept is further accelerated for real-time rendering on graphics hardware [DFR04]. Our technique differs fundamentally from [DFRS03, DFR04] in that these techniques, along with the majority of visualization work [HZ00, Dur02], concentrate mainly on illustration quality rather than rendering speed; in particular, they mostly rely on detailed original geometry whereas our technique utilizes a much simplified geometric representation.

Rendering acceleration via model simplification Due to the advance of authoring and scanning technology [LPC*00], large and detailed geometric meshes have becoming common, requiring proper LOD techniques for maintaining performance and anti-aliasing [LWC*02]. In particular, rendering large meshes on CPU require sophisticated memory management [RL00, CMRS03], and the problem is more challenging for GPU with even less storage. To make this problem tractable, we have to concentrate on preserving the most important visual cues; due to the importance of silhouettes, a coarse visual hull approximating the original object plus the necessary silhouette/visibility information is enough to obtain faithful rendering reproduction, as demonstrated in [SGG*00, MPN*02]. [SHSG01] further attests the feasibility of such approach via anti-aliasing silhouette edges.

[DDSD03] utilizes billboards for extreme model simplification. The technique is able to preserve silhouettes only for geometry well-aligned with billboard planes; in contrast, our technique preserves silhouettes uniformly over the entire model similar to [SGG*00].

Texture-based representation of geometry details We are certainly not the first to propose the idea of trading geometric complexity with extra texture storage + shader compu-

tation on GPU, as this has been explored in various forms including VDM [WWT*03], GDM [WTL*04], and relief mapping [POC05]. However, to render accurate silhouettes, these techniques would require a complex on-line procedure to combine multiple prisms or tiles covering the 3D object surface. Since our technique is designed primarily for silhouettes, our per-pixel computation is simpler and more efficient, at the expense of less accurate interior self-shadowing and occlusion. In addition, the explicit knowledge of silhouettes enables us to perform silhouette-based visualization which is not possible via [WWT*03, WTL*04, POC05].

In particular, our basic data structure (as well storage and computation requirement) is much more similar to horizon mapping [Max88, SC00, HDKS00], but we apply this basic representation in an entirely novel way for rendering silhouettes/visibility instead of self-shadowing.

Occlusion/visibility information encoding There exists a variety of previous work for encoding visibility or occlusion information [AAM03, ZHL*05, KL05]. A core part of our silhouette texture is a visibility function, which is inspired by these previous methods. A common problem for such visibility information is the huge storage requirement, requiring compression. A major novelty of our approach is that, instead of a generic data-driven compression technique like PCA, wavelet [NRH04], or spherical harmonics [SKS02], our technique first reduces the visibility function from 4D to 3D via a single-lobe observation, before applying another step of PCA compression. Our single-lobe assumption is obtained via empirical observations and works very well in practice; to our knowledge, this has not yet been done before. In addition, this single-lobe assumption allows high sampling precision of object silhouettes without consuming excessive storage.

3. Silhouette Texture

Our silhouette texture framework consists of two stages: the pre-processing stage and the rendering stage.

In the preprocessing stage, we first construct a coarse outer hull entirely enclosing the original mesh by applying offset surface generation [PKZ04] to the original mesh and then using mesh simplification [GH97] to simplify the offset mesh. (Although this process requires iterations to determine the optimal offset value for a given hull resolution, we found the resulting outer hull is typically better in approximating quality compared with the result of the modified progressive mesh construction algorithm described in [SGG*00].) Next, we parameterize the original mesh over this coarse outer hull by sampling color, normal, as well the visibility information from the original mesh, storing these sampled information as texture maps. The outer hull and the sampled texture maps serve as the only approximation of the original model, as we discard the original model after pre-processing.

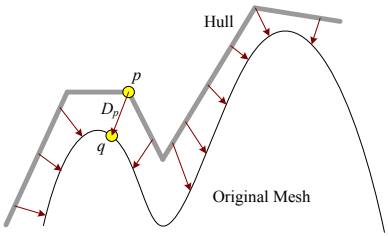


Figure 2: Correspondence between the outer hull and the original mesh. p is a point on the outer hull and q is p's corresponding point on the original mesh. D_p is the direction pointing from p to q.

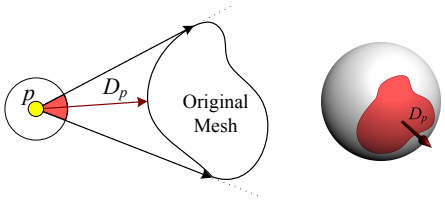


Figure 3: Visibility function (a) A 2D VF and (b) a 3D VF. Red region indicates the set of directions whose VF values are true.

In the rendering stage, we rasterize the outer hull, shaded with the corresponding color and normal texture maps. We employ a per-pixel algorithm to determine whether a screen-space pixel would lie within the original object by querying the visibility information recorded in the pre-processing stage. We implement our per-pixel algorithm as a fragment program in commodity graphics processing units (GPU), as detailed in Appendix A.

3.1. Color and Normal Information

We associate with the coarse outer hull two texture maps for storing the color and normal information of the original mesh. This is standard practice for color/normal texture map polygonal objects; the only minor difference in our technique is that we require the coarse hull to completely enclose the original mesh, whereas traditional color/normal maps do not have this restriction.

For each point p on the outer hull, we define the principal projection direction D_p as the vector from p to the point q on the original surface where the color/normal information at p is sampled (Figure 2). Specifically, the location of q can be derived during the color/normal maps construction process (e.g. [SGG*00, COM98]). We will need D_p for defining our visibility function below.

3.2. Visibility Function

Basically, our algorithm can be thought of carving out "excessive" portions of the outer hull which lies outside the sil-

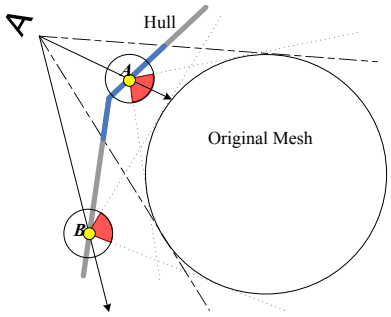


Figure 4: Illustration of our algorithm. The hull is used to approximate the original mesh. Blue region is the portion of the hull actually rendered. Point *A* is rendered and point *B* is discarded according to query results into their respective VFs.

houettes of the original mesh. A visibility function is employed to determine whether a point on the outer hull should appear as if viewing the original mesh under the same viewing parameters.

Exterior Silhouettes Only

First, we focus on using visibility function to preserve exterior silhouettes only. Our *Visibility Function* (VF) is a spherical boolean function defined over an arbitrary point p on the outer hull, with respect to an object M.

$$VF_p(\vec{v}) = \begin{cases} true & \text{M is visible to p in direction } \vec{v} \\ false & \text{M is invisible to p in direction } \vec{v} \end{cases} \quad (1)$$

where $\vec{v}$ indicates a viewing direction originated from p. (see Figure 3 for VF examples).

Given the visibility functions of all points on the outer hull, the original exterior silhouettes are reconstructed by selectively rendering the outer hull. We cast a ray from the eye point towards each point on the outer hull and then use the direction of the ray as variable to lookup the visibility function. We continue to shade the point if its visibility function evaluates true at this direction. Otherwise, the point is discarded for rendering (via fragment kill in shader program). Figure 4 illustrates this process.

Interior Silhouettes

Our visibility function defined above correctly renders exterior silouettes; unfortunately, it is not sufficient for interior silouettes, as illustrated below. As shown in Figure 5 (a), point p_1 on the outer hull has color/normal associated with point q_1 on the smaller bump of the original mesh (determined during color/normal map construction). However,

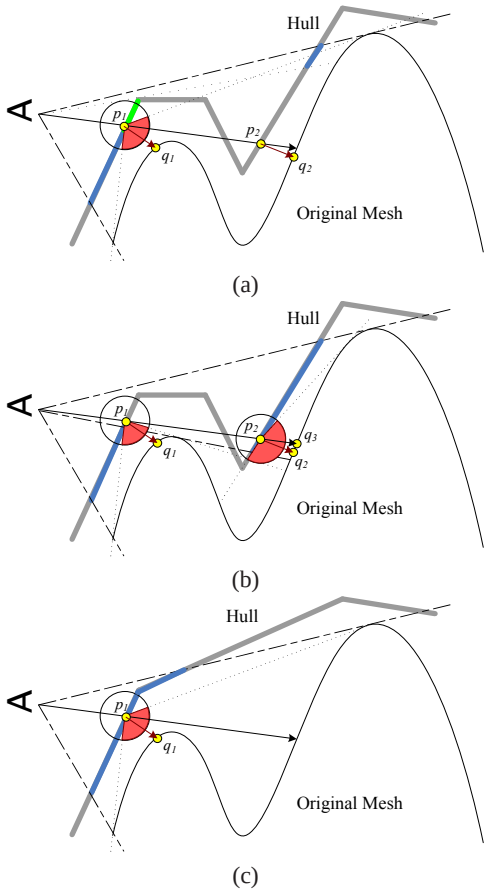


Figure 5: Processing interior silhouettes. (a) The preliminary definition of VF does not preserve interior silhouettes. (b) Modified VF reconstructs correct interior silhouettes. Note the green portion in (a) is now carved out. (c) A case where interior silhouettes cannot be processed by our framework.

under case (a), our definition in Equation 1 would render the color of p_1 incorrectly; specifically, the color/normal information should come from point q_2 on the bigger bump, instead of the stored color/normal information at q_1. This phenomenon is often termed incorrect self-occlusion in the vision literature.

To resolve this problem, we need to modify our visibility function definition. Figure 5 (b) illustrates the basic idea. Note that for $VF_{p_1}(p_2 - p_1)$, it should really be classified as invisible, since it missed the smaller bump. In particular, if we declare $VF_{p_1}(p_2 - p_1)$ as invisible, then we could obtain correct color/normal information for this ray by querying $VF_{p_2}(p_2 - p_1)$. This improvement in our visibility function can be formalized as follows:

$$VF_p(\vec{v}) = \begin{cases} true & d_M(\vec{v}) < \infty \text{ and } d_M(\vec{v}) < d_H(\vec{v}) \\ false & \text{Otherwise} \end{cases} \quad (2)$$

where $d_H(\vec{v})$ denotes the distance from p to the nearest outer hull polygon (not including the one containing p itself) along direction $\vec{v}$, and $d_M(\vec{v})$ represents the distance from p to M (If no hit occurs along the direction, we define $d = \infty$.). Essentially, what this equation says is that we classify $VF_p(\vec{v})$ as visible only if the ray $\vec{v}$ (1) intersects the original mesh (hence the term $d_M(\vec{v}) < \infty$) and (2) does not intersect any portion of the outer hull prior to intersecting the original mesh (hence the term $d_M(\vec{v}) < d_H(\vec{v})$).

However, the correctness of our visibility function formulation in Equation 2 depends on the closeness between outer hull and the original mesh. As illustrated in Figure 5 (c), when the outer hull is too coarse to follow the original geometry, we will not be able to render the self-occusion effects correctly as in case (b), where the outer hull has enough resolution. This is a tradeoff between quality and storage, and is an inherent limitation of any similar silouette clipping algorithms based on color/normal texture maps [SGG*00].

Limitations As shown in Figure 5 (b), even though the disocclusion effect is correct for p_2, it is color/normal information should really come from q_3, not q_2. This information cannot be obtained by our data structure since we only associate one pair of color/normal information per outer hull point. However, this is not unique to our technique, as any color/normal mapped polygonal models will have similar problems. In fact, if a normal-mapped model is constructed in exactly the same method as our outer hull, we would obtain identical shading results for pixels lying within the exterior silhouettes; the only difference is that our approach would correctly carve out the silhouettes while the normal-mapped model would not, as demonstrated in Figure 6.

Another limitation of our approach is that since our technique produces z values of the outer hull instead of the original geometry, we will not be able to correctly render any other object intersecting the outer hull but not the original geometry. But this is a relatively rare case.

3.3. Sampling

We have described the high level mathematical definition of our visibility function as summarized in Equation 2. We now present how to actually sample and represent VF for practical implementation. A naive sampling method would require a uniform dense sampling across the sphere around each p on the outer hull, resulting in a 2D array for a single VF. We propose a sampling method allowing us to represent each VF as a 1D array. This not only significantly reduces storage, but also allows us to store the entire VF field as a 3D texture and utilize native hardware filtering/anti-aliasing. In addition, our sampling representation incurs negligible quality degradation in reconstructed silhouettes.

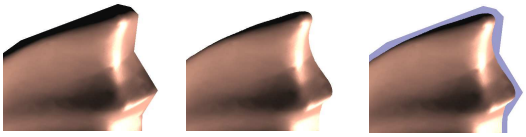


Figure 6: Comparison of normal-map with our technique. With an identical normal-mapped mesh and our outer hull, these two techniques produce indentical results within object silhouettes; the only difference is that our technique correctly renders the silhouettes. The right-most image indicates differences in gray; as shown, all differences lie on the silhouettes.

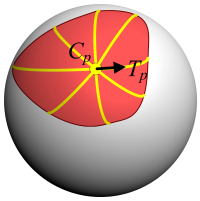


Figure 7: Sampling a VF by recording the shape of its *true* region. Angular distances from $\vec{C_p}$ to the boundary of *true* region are recorded at different azimuth angles (eight angles in this figure) in counter-clockwise order. The first azimuth angle is the same direction as $\vec{T_p}$.

Our VF sampling representation is based on one crucial experimental observation: the majority of VF contains a single connected region of *true*, as illustrated in Figure 3(b). This peculiar property allows us to describe each VF as a 1D array by the circumference of the *true* region, as follows (see Figure 7 for an illustration). First, we select a direction $\vec{C_p}$ which roughly sits in the middle of the entire *true* region by taking the arithmetic mean of $\{\vec{v} \mid VF_p(\vec{v}) = true\}$. We then record in counter-clockwise order the angular distance from $\vec{C_p}$ to the boundary of the *true* region at discrete azimuth angles, the first of which lies in the same direction with $\vec{T_p}$, a vector tangent to $\vec{C_p}$, and store these distances into a 1D array.

For all VFs over the entire outer hull, we sample them using the above method and pack the result into a 3D texture, which is denoted as a *Silhouette Texture* (Figure 8). First, we parameterize the hull into a 2D texture atlas via standard techniques (e.g. UVAtlas in the D3DX library [ZSGS04]). We then store the 1D array of angular distances corresponding to each VF in the angular dimension of the 3D texture. One nice property of this organization is that our VF samples can be linearly interpolated in all three dimensions us-

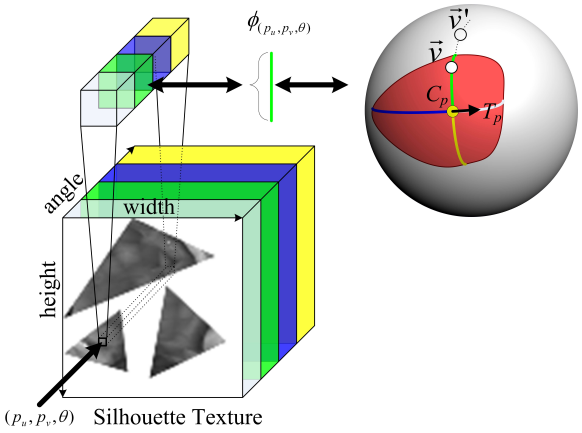


Figure 8: Silhouette texture encoding and decoding. Our silhouette texture is encoded as a 3D texture on GPU, with spatial texture atlas in the $1^{st}/2^{nd}$ dimensions and angular resolution in the 3^{rd} dimension. For illustration purpose, we use a small angular resolution of 4. To encode a single VF_p, we store angular distance samples along various azimuth angles, with the 1^{st} angular sample aligned with T_p. For decoding, we use the combination of spatial coordinates (p_u, p_v) and azimuth angle θ of viewing direction at p to index the silhouette texture. In this example, we determine $VF_p(\vec{v}) = true$ and $VF_p(\vec{v'}) = false$ by comparing their fetched texture values with their angular distances from C_p, respectively.

ing native hardware, producing smooth silhouettes with high rendering performance.

In addition to angular distance samples, $\vec{C_p}$ and $\vec{T_p}$ over the hull also need to be stored explicitly in order to aid decoding of silhouette texture during rendering. In our implementation, we store them as per-vertex attributes of the outer hull and perform interpolation during rendering. (This is correct for decoding when every interpolated $\vec{C_p}$ lies in the *true* region of its corresponding VF, as are most cases in our experiments. However, if this condition cannot be met due to the low density of vertices of the outer hull which is not sufficient to capture the drastically changing VFs across the surface, one may choose to store $\vec{C}$ and $\vec{T}$ as separate high resolution texture maps.) Since $\vec{C_p}$ is solely determined by the hull and the original mesh, we only need to compute corresponding $\vec{T_p}$, using some standard method; in our implementation, we utilize D3DXComputeTangentFrameEx function call in D3DX library.

The major limitation of our sampling method is that it can only handle one lobe, as described above. Several typical cases where our one-lobe assumption would fail are shown in Figure 9. Nevertheless, this limitation can be alleviated by constructing an outer hull which reflects major features of the original mesh. Actually, the definition of VF roughly records the shape of the nearest "bump" which is generally far less complicated than the entire original mesh.

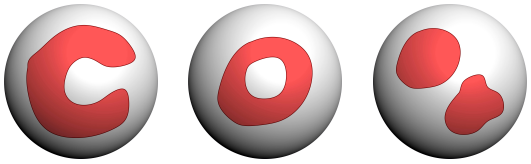


Figure 9: Examples of VFs which cannot be properly sampled by our method.

In our experiments with many different real-world meshes, our single-lobe assumption has hold pretty well and so far we have encountered very few such pathological cases as illustrated in Figure 9; we have observed the appearances of such failures only when the outer hull is very coarse, such as the 22 vertex case in Figure 13. To quantify the error incurred by our single-lobe assumption, we utilize the following equation as error metric:

$$\begin{aligned} e(a,b) &= \begin{cases} 1 & a \neq b \\ 0 & a = b \end{cases} \\ E(VF_1,VF_2) &= \frac{1}{4\pi} \int_{\vec{v}\in R^3, \|\vec{v}\|=1} e(VF_1(\vec{v}),VF_2(\vec{v}))d\vec{v} \end{aligned} \tag{3}$$

where VF_1 and VF_2 indicate the original and our approximated VF at the same point p. We have measured the average errors across all hull points of the meshes we utilized in this paper, as detailed in Table 1.

3.4. Rendering

We now describe how we render each pixel covered by a silhouette-textured object, as implemented in a pixel shader program (see Appendix A). For every screen space pixel p, we use its world space position $\vec{O_p}$, the sampling center $\vec{C_p}$ (not surface normal), and tangent vector $\vec{T_p}$ to establish a local tangent space. We calculate the viewing vector $\vec{v}$ at p as the normalized vector pointing from eye position to O_p, expressed in p's local tangent frame. We then lookup $\vec{v}$ in the silhouette texture to determine whether $VF_p(\vec{v})$ equals *true* (Figure 8). Specifically, we compute the azimuth angle θ of $\vec{v}$ with respect to $\vec{T_p}$, and use θ along with the spatial texture coordinates (p_u, p_v) of p to index into the silhouette texture, returning a value ϕ representing the angular distance to $\vec{C_p}$. Note that we allow ϕ to be linearly filtered by the texturing hardware, as discussed above. Finally, we compare ϕ against the angular distance between $\vec{v}$ and $\vec{C_p}$ to determine the visibility of p, as formalized in Equation 4. If p is not visible, we simply discard it via fragment kill.

$$VF_p(\vec{v}) = acos(\vec{v}\cdot\vec{C_p}) \leq \phi \tag{4}$$

Note that Equation 4 renders only hard silhouettes without any alpha-antialiasing. This feature could be easily added to

our algorithm, but it would involve the tricky issue of transparency ordering. In our experience, hard silhouettes rendered by Equation 4 is perceptually smooth, especially after full-screen anti-aliasing enabled by commodity graphics chips.

3.5. Compression

Throughout this paper, unless explicitly specified, all experimental models are rendered using silhouette textures with a spatial resolution of 512×512 and an angular resolution of 64. As we use 8-bit to store each angular distance sample, the total size for one such silhouette texture is 16MB. Although this is not a huge amount considering the average video memory size of today's graphics cards, the high correlation of distance samples in silhouette texture opens up possibility for further compression without sacrificing too much rendering quality.

To perform such a compression, we employ singular value decomposition (SVD) to process the silhouette texture. First, we reorganize the silhouette texture as a 262144(=512×512)×64 matrix A, one row of which representing all 64 angular distance samples for a particular VF. SVD decomposition is then applied to A to get $A = U\lambda V^T = WV^T$, where V contains the eigen functions of A and $W = U\lambda$ contains weights of the eigen functions. Exploiting the correlation among samples of VF, we can keep only a few eigen-functions of the greatest eigenvalues and omit the rest. In our experiments, 16 eigen functions are enough to maintain a rendering quality close to that using the original data (Figure 10(b)). Thus, the size of the silhouette texture is reduced from 16MB to 512×512×16 + 16×64 ≈ 4MB.

However, rendering from compressed VF texture is about 5 to 8 times slower than rendering from uncompressed VF due to the need to perform custom bilinear/trilinear filtering via fragment program; in contrast, as discussed earlier, for uncompressed VF our algorithm exploits the graphics hardware for native filtering. All timing reported in this paper are measured from rendering with uncompressed VF texture.

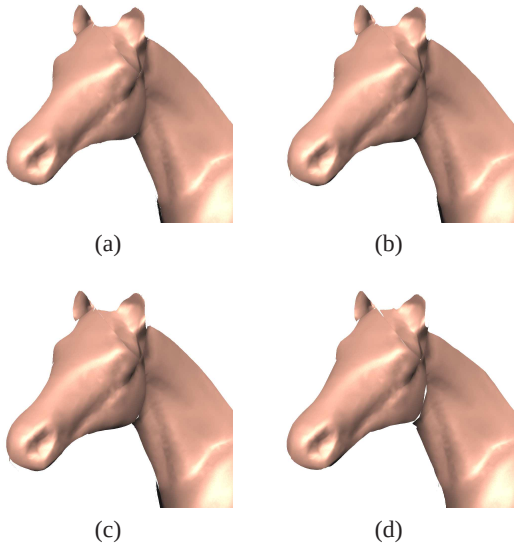


Figure 10: Compression quality. (a) uncompressed data (b) 16 term, PSNR = 45.99db (c) 8 term, PSNR = 39.52db (d) 4 term, PSNR = 30.91db

4. Results and Applications

Quality, speed, and anti-aliasing

Figure 1 and Figure 16 demonstrate the rendering results of our algorithm; note that even though our technique utilizes a very coarse mesh, we are still able to reproduce detailed silhouettes of the original object. This quality is achieved with a higher frame rate than rendering the detailed original model, due to proper load balancing between vertex and fragment processors enabled by our technique. See Table 1 for detailed timing and mesh statistics. (For fair comparison, we have performed all renderings in Figure 1 and Figure 16 so that no geometry aliasing occurs, i.e. the projected vertex density roughly equals to the pixel grid spacing.)

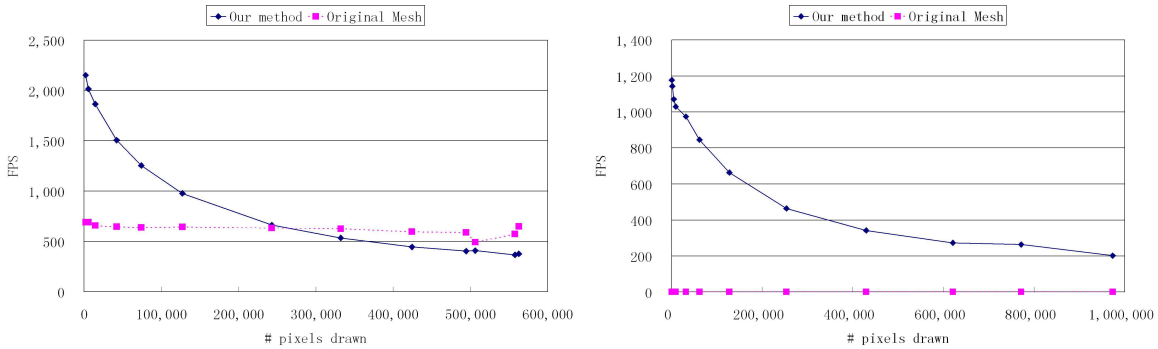
Figure 11 provides a more detailed performance analysis, comparing frame rates of the original model versus our algorithm under different number of projected pixels. For complex models such as Lucy, the curve for the original model remains flat (indicating a geometry-bound workload), whereas our technique runs significant faster, especially when the projection area is small. This indicates an important advantage of our technique, as it automatically performs geometry LOD as the model distances away from the camera. However, for simple models such as Bunny, our technique might run slower than rendering the original model when the projection area is big enough.

To emphasize the quality of reconstructed silhouettes, we have utilized a simple shading model via normal maps; since the shading model is orthogonal to silhouette/visibility determination, a more complex shader can be easily swapped in for higher rendering quality.

In addition, our technique creates smoothly interpolated silhouettes by taking advantage of native 3D hardware filtering during VF look up (discussed in Section 3.3). As demonstrated in Figure 12, we do not need a very high resolution sampling to reconstruct smooth silhouettes; when the screen-projected silhouette-texture sample spacing is large (e.g. 10 pixels), our method can still produce faithful and smooth silhouettes via proper interpolation of existing VF samples.

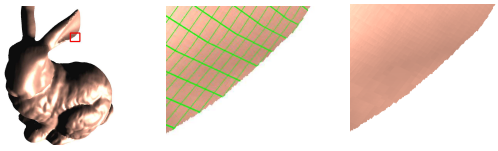
mesh name	# original vertices	# outer hull vertices	viewport size	fps (original mesh)	fps (siltex)	average VF sampling error
Armadillo	40002	1000	512×512	308	725	0.426%
Bunny	34834	500	512×512	411	638	0.402%
Gargoyle	30002	500	512×512	460	762	0.309%
Dragon	40525	970	512×512	523	674	0.409%
Lucy	1000000	10000	1024×1024	1.5	273	0.805%

Table 1: Performance timing and mesh statistics. Please refer to Figure 1 and Figure 16 for quality comparison. All reported frame-rates are measured on a NVIDIA Geforce-7800-GT chip.



Bunny Lucy

Figure 11: Performance comparison of rendering original model and our technique with different number of projected pixels (by varying the distance between the camera and the object). The frame-rate for rendering the original Lucy model is about 1.5 fps. The viewport size is 1024×1024 for all cases.



entire mesh VF texture our rendering

Figure 12: Smooth interpolation during silhouette reconstruction. Left: a mesh with a local region marked by a red rectangle. Middle: VF texture in that region with texels visualized in green grid. Right: Our rendering result. Note that our technique performs smooth interpolation even within a coarse silhouette texture.

In our current implementation we utilize only one mipmap level; even though in theory a full mipmap texture ought to be adapted for storing our silhouette texture, we have not found it necessary. In our experiments, we have found that our single mipmap approach produces smooth and continuous silhouettes even when the object is viewed far away. We conjecture that this is caused by the fact that under perspective projection, the set of viewing rays are more coherent when the object is zoomed out, making mipmap unnecessary.

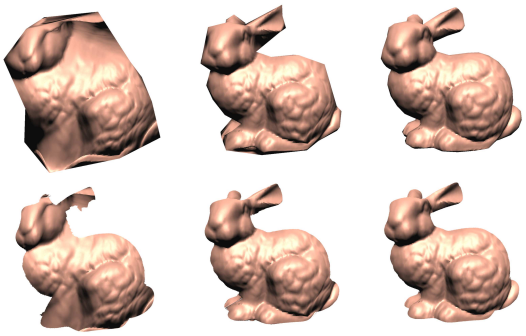
Another related issue is cache coherence; when the object is viewed from far away, our single resolution approach might suffer from incoherence cache access due to the spreading-out of texture footprints at adjacent pixels. Fortunately, experimentally we have not found this to be a major issue.

Parameters

The quality, performance, and storage of our algorithm depend on a number of parameters, including the size of the outer hull as well the 3D silhouette texture resolution.

Figure 13 illustrates the impact of outer hull on our rendering quality. As expected, if the outer hull is too coarse, it will fail to follow the silhouettes properly, resulting in obvious visibility errors (such as holes) at our rendering. As the outer hull becomes finer, so improves our rendering result. This figure also demonstrates the major limitation of our approach: we rely on the closeness of the outer hull with respect to the original mesh in order to faithfully reproduce interior silhouettes (as explained in Figure 5). This makes our algorithm unsuitable for highly self-occluded objects, such as trees or bushes.

Figure 14 demonstrates the effect of angular + spatial resolution of our 3D silhouette texture on rendering quality.



22 vertices 102 vertices 500 vertices

Figure 13: The effect of outer hull quality on rendering. Top row: outer hull + normal map. Bottom row: our rendering.

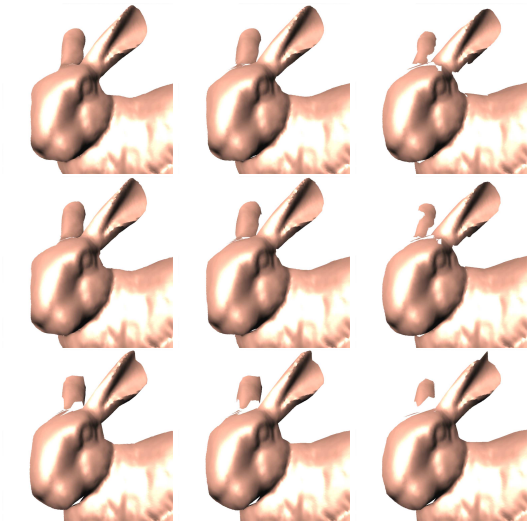
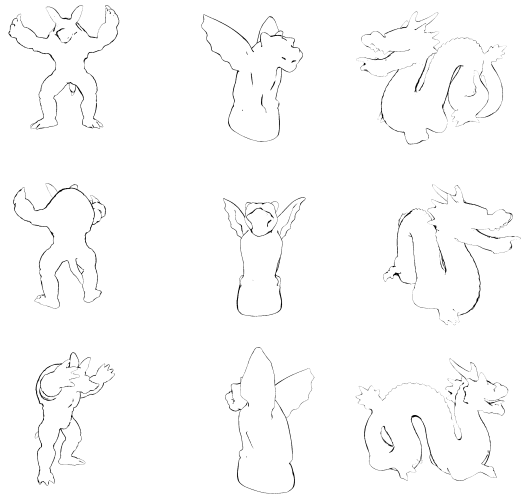


Figure 14: The effect of VF texture resolution on rendering. From top to bottom: spatial resolutions in 512^2, 128^2, and 32^2. From left to right: angular resolutions in 64, 16, and 4.

As expected, the higher the resolution, the better quality; at extremely low spatial or angular resolutions, our rendering would exhibit holes around silhouettes due to grossly inaccurate visibility sampling. Since current graphics hardware imposes a maximum resolution (512^3) on 3D textures, this might not be sufficient to capture VF for more complex models such as the Lucy model in Figure 16. (Notice the holes around her ear and hair similar to the artifacts in Figure 14.) We believe this restriction can be alleviated by using multiple 3D textures, but we have not explored this yet.

In addition to sampling VF, sufficient resolution for the normal and texture maps is also important for maintaining rendering quality. Insufficient sampling of normal maps can cause artifacts as shown in the Lucy model (around her nose and mouth) in Figure 16.



787 fps 802 fps 753 fps

Figure 15: Silhouette visualization by our technique. Each column shows the same model in three different views.

Silhouette Visualization

Due to its ability to recover silhouette information, our algorithm has additional applications beyond just rendering speed-up.

One such possible applications is silhouette visualization, as shown in Figure 15. Unlike previous methods which reply on additional processing to render silhouettes, our algorithm can sketch out the silhouettes directly by a simple modification of our fragment program. Instead of knocking out all pixels with $VF_p(\vec{v}) == false$ as evaluated in Equation 4, we determine the intensity $I_p(\vec{v})$ at each pixel p with viewing direction $\vec{v}$ by the closeness of $acos(\vec{v} \cdot \vec{C_p})$ and ϕ (see Equation 4 for the meaning of these symbols):

$$I_p(\vec{v}) = \begin{cases} true & | \, acos(\vec{v} \cdot \vec{C_p}) - \phi \, | \leq \sigma * Z_p \\ false & \text{otherwise} \end{cases} \quad (5)$$

where σ is a user threshold parameter deciding the thickness of the lines and Z_p denotes the depth value at point p. (We need to scale σ by Z_p to take perspective projection into account, in order to generate roughly uniform thick lines.)

Despite the simplicity of our visualization algorithm, we are able to effectively visualize both interior and exterior silhouettes in real-time, as demonstrated in Figure 15. Our current algorithm does not take into account suggestive contours [DFRS03, DFR04]. We believe this is can be achieved by slight perturbation of view points via a multi-pass a-

buffer like algorithm; we intend to explore this possibility as a future work.

5. Conclusions and Future Work

We present *silhouette texture*, a new data representation that allows real-time rendering of high quality interior and exterior silhouettes on graphics hardware. Our algorithm achieves faster rendering speed for detailed polygonal meshes due to its load balancing ability between vertex and fragment processors. We also utilize native texture hardware filtering for proper anti-aliasing. We have also presented an additional application of our technique, silhouette visualization, beyond high speed and high quality rendering.

For future work, we plan to extend our technique to handle dynamic, articulated objects. In principle, we could precompute silhouette textures for each individual body parts (head, torso, and limbs) and combine them in a novel way during rendering. The challenges include how to perform proper visibility sorting and how to incorporate vertex skinning. We are also interested in exploring additional applications that require silhouette information, which is naturally provided by our silhouette texture representation.

Acknowledgement We would like to thank JianWei Han for his help on performance measurement and anonymous reviewers for their comments. We originally planned to utilize several complex models from Digital Michelangelo project but potential legal ramifications prevented us from doing so; we would like to thank Professor Marc Levoy for his help on this issue. Polygonal models are courtesy of Stanford University, University of Washington, and Georgia Institute of Technology.

Appendix A: Pixel Shader

To facilitate reproduction of our technique, we have attached our pixel shader in Table 2.

```
#define PI            3.1415926536

texture silTex;
sampler texSampler = sampler_state
{
   texture = <silTex>;
   AddressU  = WRAP;    AddressV  = WRAP;    AddressW  = WRAP;
   MIPFILTER = NONE;    MINFILTER = LINEAR;    MAGFILTER = LINEAR;
};

float LookUpSilTex(
   sampler siltexSampler,
   float3 E, float3 N, float3 T, float2 tex)
{
   float3 B;
   float  fx, fy, fangle;

   // Calculate binormal from normal and tangent
   B = cross(T, N);
   // Transform E into local frame
   fx = dot(T, E); fy = dot(B, E);
   //Convert to azimuth angle
   fangle = atan2(fy, fx) / (2*PI);

   // fetch the VF sample
   return    cos(PI * tex3D( siltexSampler, float3(tex, fangle)).a);
}

float4 PS_SilTex(
   in float3 normal    : NORMAL,     //Cp
   in float3 tangent   : TANGENT,    //Tp
   in float2 tex       : TEXCOORD0,
   in float3 v         : TEXCOORD1  //Viewing direction
): COLOR
{
   float3    N, T;

   // Normalize the interpolated normal
   N = normalize(normal);
   // Orthogonize the tangent
   T = normalize(tangent - dot(tangent, N) * N);

   // Visibility Function(E) = true ?
   if ( dot(N, v) >= LookUpSilTex(texSampler, v, N, T, tex) )
   {
      return Shade(......); // a user-defined function for shading
   } else {
      //kill current fragment
      clip(-1);
      return -1;
   }
}
```

Table 2: Our pixel shader program written in HLSL.

References

[AAM03] ASSARSSON U., AKENINE-MOLLER T.: A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Trans. Graph. (SIGGRAPH 2003) 22*, 3 (2003), 511–520.

[CMRS03] CIGNONI P., MONTANI C., ROCCHINI C., SCOPIGNO R.: External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics 9*, 4 (2003), 525–537.

[COM98] COHEN J., OLANO M., MANOCHA D.: Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), pp. 115–122.

[DDSD03] DECORET X., DURAND F., SILLION F. X., DORSEY J.: Billboard clouds for extreme model simplification. *ACM Trans. Graph. (SIGGRAPH 2003) 22*, 3 (2003), 689–696.

[DFR04] DECARLO D., FINKELSTEIN A., RUSINKIEWICZ S.: Interactive rendering of suggestive contours with temporal coherence. In *NPAR '04: Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering* (2004), pp. 15–145.

[DFRS03] DECARLO D., FINKELSTEIN A., RUSINKIEWICZ S., SANTELLA A.: Suggestive contours for conveying shape. *ACM Trans. Graph. (SIGGRAPH 2003) 22*, 3 (2003), 848–855.

[Dur02] DURAND F.: An invitation to discuss computer depiction. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering* (Annecy, France, June 3-5 2002).

[GH97] GARLAND M., HECKBERT P. S.: Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), pp. 209–216.

[HDKS00] HEIDRICH W., DAUBERT K., KAUTZ J., SEIDEL H.-P.: Illuminating micro geometry based on precomputed visibility. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 455–464.

[HZ00] HERTZMANN A., ZORIN D.: Illustrating smooth surfaces. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 517–526.

[KL05] KONTKANEN J., LAINE S.: Ambient occlusion fields. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005), pp. 41–48.

[Koe84] KOENDERINK J.: What does the occluding contour tell us about solid shape? *Perception 13* (1984), 321–330.

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph. (SIGGRAPH 2004) 23*, 3 (2004), 769–776.

[LPC*00] LEVOY M., PULLI K., CURLESS B., RUSINKIEWICZ S., KOLLER D., PEREIRA L., GINZTON M., ANDERSON S., DAVIS J., GINSBERG J., SHADE J., FULK D.: The digital michelangelo project: 3d scanning of large statues. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 131–144.

[LWC*02] LUEBKE D., WATSON B., COHEN J. D., REDDY M., VARSHNEY A.: *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[Max88] MAX N. L.: Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer 4*, 2 (1988), 109–117.

[MPN*02] MATUSIK W., PFISTER H., NGAN A., BEARDSLEY P., ZIEGLER R., MCMILLAN L.: Image-based 3d photography using opacity hulls. *ACM Trans. Graph. (SIGGRAPH 2002) 21*, 3 (2002), 427–437.

[NRH04] NG R., RAMAMOORTHI R., HANRAHAN P.: Triple product wavelet integrals for all-frequency relighting. *ACM Trans. Graph. (SIGGRAPH 2004) 23*, 3 (2004), 477–487.

[PKZ04] PENG J., KRISTJANSSON D., ZORIN D.: Interactive modeling of topologically complex geometric detail. *ACM Trans. Graph. (SIGGRAPH 2004) 23*, 3 (2004), 635–643.

[POC05] POLICARPO F., OLIVEIRA M. M., COMBA J. L. D.: Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005), pp. 155–162.

[RL00] RUSINKIEWICZ S., LEVOY M.: Qsplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 343–352.

[SC00] SLOAN P.-P. J., COHEN M. F.: Interactive horizon mapping. In *Rendering Techniques 2000: Proceedings of the 2000 Eurographics Workshop on Rendering* (2000), pp. 281–286.

[SGG*00] SANDER P. V., GU X., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 327–334.

[SHSG01] SANDER P. V., HOPPE H., SNYDER J., GORTLER S. J.: Discontinuity edge overdraw. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), pp. 167–174.

[SKS02] SLOAN P.-P., KAUTZ J., SNYDER J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph. (SIGGRAPH 2002) 21*, 3 (2002), 527–536.

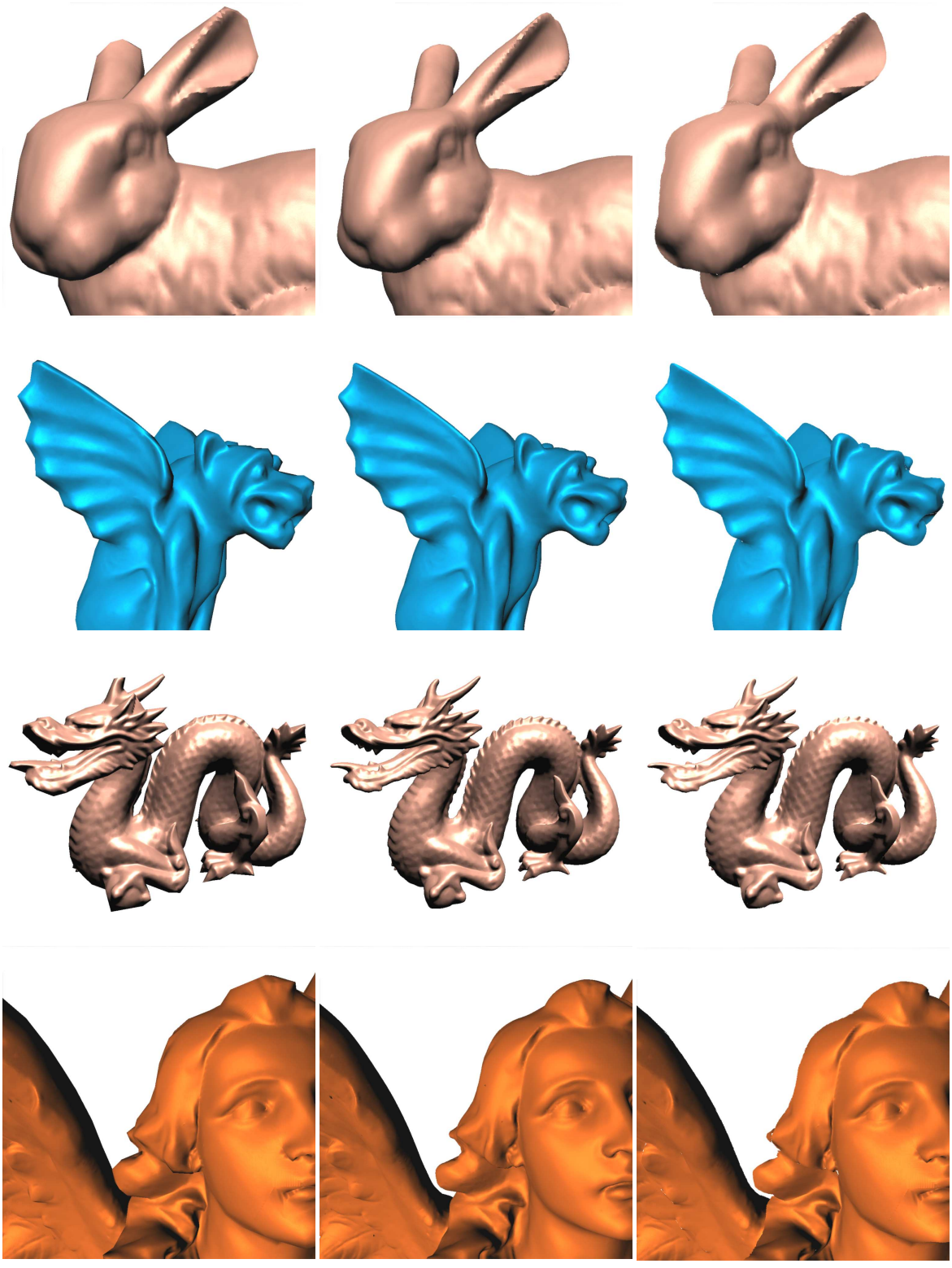
[UT04] UNREAL-TECHNOLOGY: Unreal engine 3, 2004. `http://www.unrealtechnology.com/html/technology/ue30.shtml`.

[WTL*04] WANG X., TONG X., LIN S., HU S.-M., GUO B., SHUM H.-Y.: Generalized displacement maps. In *Rendering Techniques 2004: Proceedings of the 2004 Eurographics Symposium on Rendering* (2004), pp. 227–234.

[WWT*03] WANG L., WANG X., TONG X., LIN S., HU S., GUO B., SHUM H.-Y.: View-dependent displacement mapping. *ACM Trans. Graph. (SIGGRAPH 2003) 22*, 3 (2003), 334–339.

[ZHL*05] ZHOU K., HU Y., LIN S., GUO B., SHUM H.-Y.: Precomputed shadow fields for dynamic scenes. *ACM Trans. Graph. (SIGGRAPH 2005) 24*, 3 (2005), 1196–1201.

[ZSGS04] ZHOU K., SYNDER J., GUO B., SHUM H.-Y.: Iso-charts: stretch-driven mesh parameterization using spectral analysis. In *SGP '04: Proceedings of the 2004 Eurographics/ACM SIGGRAPH symposium on Geometry processing* (2004), pp. 45–54.



outer hull + normal map | original mesh | our result

Figure 16: Rendering quality comparison for various meshes. Please refer to Table 1 for performance timing and mesh statistics.