A Hybrid Method for Computing Apparent Ridges

Eric Jardim Luiz Henrique de Figueiredo
IMPA – Instituto Nacional de Matemática Pura e Aplicada, Rio de Janeiro, Brazil
ejardim@impa.br lhf@impa.br

***Abstract*—We propose a hybrid method for computing apparent ridges, expressive lines recently introduced by Judd et al. Unlike their original method, which works entirely over the mesh in object space, our method combines object-space and image-space computations and runs partially on the GPU, taking advantage of modern graphic cards processing power and producing faster results in real time.**

***Keywords*-expressive lines; non-photorealistic rendering**

I. Introduction

Expressive line drawing of 3D models is a classic artistic technique and remains an important problem in non-photorealistic rendering [1], [2]. A good line drawing can convey the model's geometry without using other visual cues like shading, color, and texture [3]. Frequently, a few good lines are enough to convey the main geometric features [4], [5], [6]. The central problem is how to mathematically define what good lines are: ideally, they should capture all perceptually relevant geometric features of the object and should depend on how the object is viewed by the observer.

There are several techniques (e.g., [7], [8], [9], [10], [11]) for expressive line rendering of 3D models, but no single method has emerged as the best for all models and viewing positions [5], [6]. Apparent ridges [11] have a relatively simple definition and produce good results in many cases.

In this paper, we propose a hybrid method for computing apparent ridges. Our main goal and motivation is achieving better performance without compromising image quality. While the original method [11] is CPU-based and works entirely over the mesh in object space, our method combines object-space and image-space computations and runs partially on the GPU, exploiting the processing power of modern graphic cards and producing faster results in real time.

II. Previous work on line rendering

We start by briefly reviewing some of the lines that have been proposed for expressive line drawing of 3D models. These lines are illustrated in Figure 1.

Object contours or *silhouettes* [12] are probably the most basic type of feature line: they separate the visible and the invisible parts of an object. Geometrically, contours are the loci of points where the normal to the surface of the object is perpendicular to the viewing vector. Thus, contours are first-order view-dependent lines, that is, they depend only on the surface normal and on the viewpoint. Contours alone may not be enough to capture all perceptually relevant geometric features of an object, but every line drawing should contain them [7]. Moreover, other lines, such as ridges and valleys and suggestive contours, must be combined with silhouette contours to yield pleasant and perceptually complete pictures.

Ridges and valleys [13], [14] are another traditional type of feature line: they are the loci of points where the maximum principal curvature assumes an extremum in the principal direction (maxima at ridges and minima at valleys). Ridges and valleys are second-order curves that complement contour information because they capture elliptic and hyperbolic maxima on the surface. However, ridges and valleys often convey sharper creases than the surface actually has. Moreover, some models have so many ridges and valleys that the resulting image is not a clean drawing. Finally, since ridges and valleys depend only on the geometry of the model, and not on the viewpoint, these lines can appear too rigid in animated drawings. View-dependent fading effects have been proposed to mitigate this problem [9].

Suggestive contours [8], [15] are view-dependent lines that naturally extend contours at the joints. Intuitively, suggestive contours are contours in nearby views. More precisely, suggestive contours are based on the zeros of the radial curvature in the viewing direction projected onto the tangent plane. For the radial curvature to achieve the zero value in some direction, the interval between the principal curvatures must contain zero. Thus, suggestive contours cannot appear in elliptic regions, where the Gaussian curvature is positive, and so suggestive contours cannot depict convex features. Suggestive contours are visually more pleasant than the previous lines because they combine view dependency and second-order information to yield cleaner drawings. Nevertheless, they still need contours to yield perceptually complete pictures. *Suggestive and principal highlights* [10] complement suggestive contours by including positive minima or negative maxima of directional curvatures. These highlight lines typically occur near intensity ridges in the shaded image (suggestive contours typically occur near intensity valleys).

Apparent ridges [11] is a recent technique that produces good results in many cases. With a single mathematical definition of what a good line is, apparent ridges depict most features that are captured by other definitions and some additional features not captured before. As explained below, apparent ridges are based on a *view-dependent curvature* that plays an analogue role for apparent ridges as the curvature

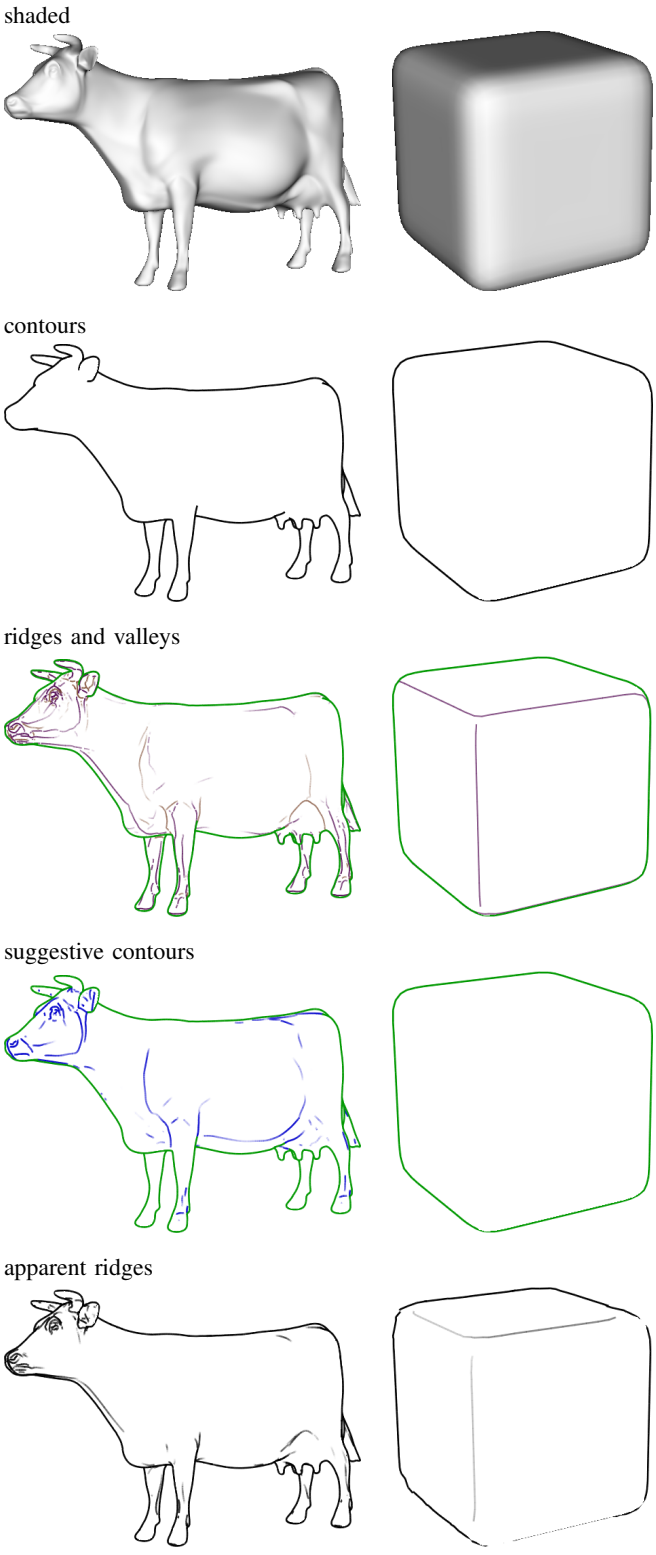

Figure 1. Comparison of expressive lines for two 3D models. Contours are essential, but insufficient to depict a shape. Ridges and valleys extend contours, but angles are too sharp and appear at rigid places due to their view independence (contours in green). Suggestive contours smoothly complement contours in a view-dependent way but do not appear on convex regions (contours in green). Apparent ridges depict features in a smooth and clean view-dependent way. They appear at convex regions and contain contours.

does for ridges and valleys. Like suggestive contours, apparent ridges combine both second-order information and view-dependency. Unlike suggestive contours, however, contours are a special case of apparent ridges and so do not require extra computation or special treatment.

Lee et al. [16] described a GPU-based method that renders lines and highlights along tone boundaries that can include silhouettes, creases, ridges, and generalized suggestive contours. Their work bears some resemblance to ours but differs in its goals and methods.

III. Apparent Ridges

The key idea of the view-dependent curvature used to define apparent ridges is to measure how the surface bends with respect to the viewpoint, taking into account the perspective transformation that maps a point on the surface to a point on the screen. We shall now review how the view-dependent curvature is defined and computed. For details, see Judd et al. [11].

Given a point p on a smooth surface M, the *shape operator* at p is the linear operator S defined on the tangent plane to M at p by $S(r) = D_r n$, where r is a tangent vector to M at p and $D_r n$ is the derivative of the normal to M at p in the r direction. The shape operator is a self-adjoint operator, and so has real eigenvalues k_1 and k_2, known as the *principal curvatures* at p; the corresponding eigenvectors e_1 and e_2 are called the *principal directions* at p.

Let Π be the parallel projection that maps M onto the screen and let $q = \Pi(p)$. If p is not a contour point, then Π is locally invertible and we can locally define an inverse function Π^{-1} that maps points on the screen back to the surface M. The inverse Jacobian J_Π^{-1} of Π maps screen vectors at q to tangent vectors at p. The *view-dependent shape transform* Q at $q = \Pi(p)$ is defined by $Q = S \circ J_\Pi^{-1}$, where S is the shape operator at p. The view-dependent shape transform is thus the screen analogue of the shape operator.

The *maximum view-dependent curvature* is the largest singular value of Q:

$$q_1 = \max_{||s||=1} ||Q(s)||$$

This value can be computed as the square root of the largest eigenvalue of $Q^T Q$. The singular value q_1 has a corresponding direction t_1 on the screen called the *maximum view-dependent principal direction*. *Apparent ridges* are the local maximum of q_1 in the t_1 direction, or

$$D_{t_1} q_1 = 0 \quad \text{and} \quad D_{t_1} (D_{t_1} q_1) < 0$$

This definition adds view dependency to ordinary ridges. When a point moves towards a contour, q_1 will tend to infinity due to projection. Although the view-dependent curvature is not defined at contours, q_1 is well-behaved and achieves a maximum at infinity. This means that contours can be treated as a special case of apparent ridges.

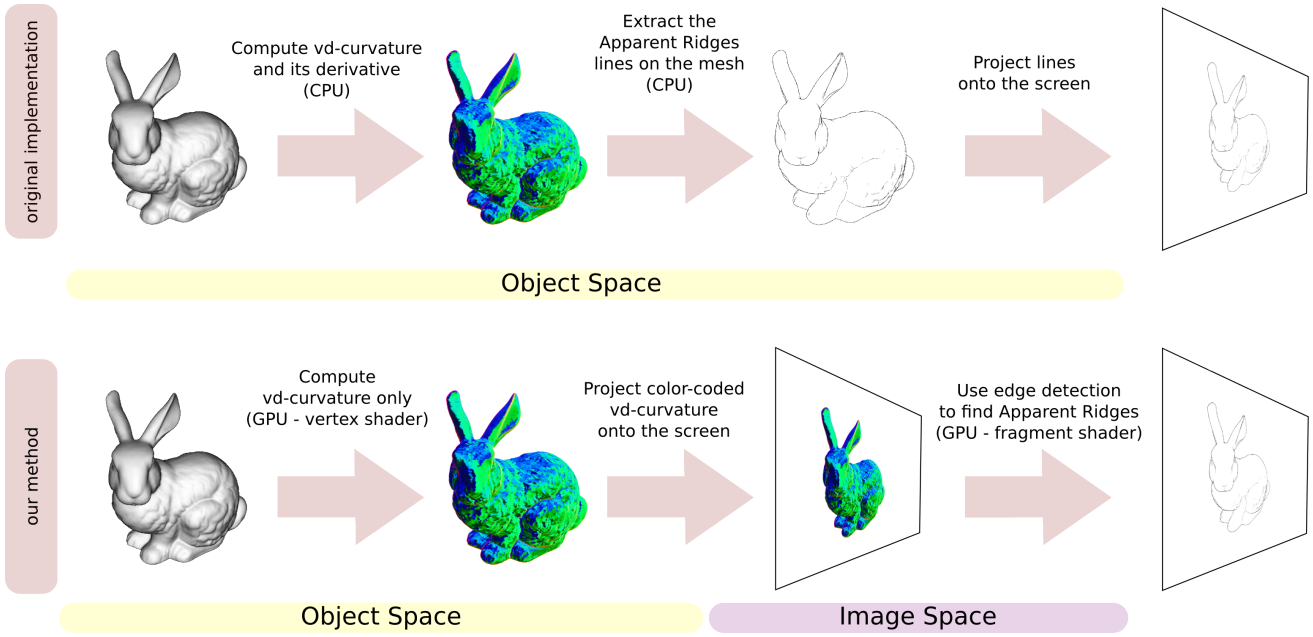

Figure 2. The original method happens in object space and runs entirely on the CPU. Ours is a hybrid method that runs partially on the GPU.

IV. Our method

The main motivation of our method is to exploit the GPU processing power to speed up the extraction of apparent ridges without compromising image quality. Judd et al. [11] presented a CPU-based method for finding apparent ridges on triangle meshes. All computations are performed in *object space*, over the 3D mesh. The result is a set of 3D lines that lie on the mesh and approximate the actual apparent ridges. These lines are then projected and drawn onto the screen. In contrast, our method is a *hybrid method*: it has an object-space stage and an image-space stage (see Figure 2). We now explain the modifications needed in their approach and some implementation details to achieve this goal.

As seen in Section III, apparent ridges are the loci of local maxima of the view-dependent curvature q_1 in the maximum view-dependent principal direction t_1. Judd et al. [11] extract apparent ridges by estimating $D_{t_1} q_1$ at the vertices of the mesh and finding its zero crossings between two mesh edges for each triangle of the mesh. The estimation of $D_{t_1} q_1$ at each vertex is done by finite differences, using the q_1 values of the adjacent vertices. Here lies the main bottleneck of their method: the derivative computation is expensive and must be repeated every time the viewpoint changes.

In our method, we split the rendering process into two stages, which we shall discuss in detail below (see Figure 2). In the first stage, which happens in object space, we estimate the view-dependent curvature data over the 3D mesh. In the second stage, which happens in image space, we extract apparent ridges using edge detection, without computing derivatives. The performance of this stage depends only on the image size, not on the mesh size, providing an overall performance improvement. Moreover, this split allows us to use vertex and fragment shaders to run each stage on the GPU, exploiting its processing power and parallelism. These changes provide significant speedups, which we shall discuss in Section V.

A. Object-space stage

In the first stage, q_1 and t_1 are estimated at each vertex of the mesh. This is done in a vertex shader by the GPU using the same computations performed by Judd's method. The result is packed into the color output RGB channels of the vertex shader, using one channel for q_1 and two channels for pack t_1, because t_1 is a 2D screen vector (see Figure 3). The vertex shader is executed every time the viewpoint changes.

The required 3D data (normal n, principal curvatures k_1 and k_2, principal directions e_1 and e_2) is estimated on the CPU using a technique by Rusinkiewicz [17] implemented in the *trimesh2* library [18] and is passed to the vertex shader as vertex information like colors and texture coordinates. This 3D data is computed only once because it does not depend on the viewpoint.

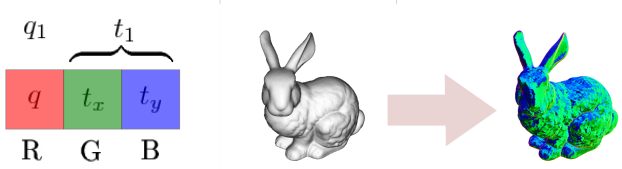

Figure 3. Color coding of q_1 and t_1.

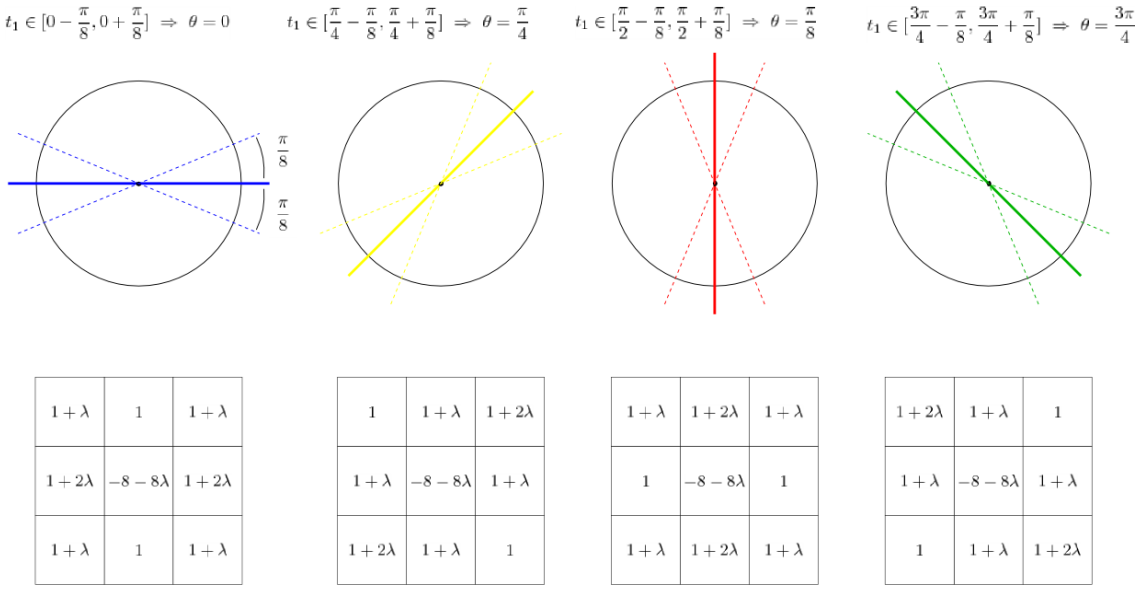

Figure 4. Laplacian-like adaptive filter. Filter setup is chosen according to the maximum view-dependent principal direction.

As discussed in Section III, q_1 is a non-negative value and achieves extremely high values near the contours. Thus, we need to truncate the q_1 value to fit into the channel interval $[0, 1]$. Before this truncation, q_1 must be scaled so that all the local maxima lie in $[0, 1]$. To preserve precision, the scaling factor cannot be too high or too low. We guess an acceptable value for this factor empirically, using the feature size of the model estimated by the *trimesh2* library [18]. The user can then fine-tune it by hand if needed. For easy manipulation, we introduced an exponential parameter τ so the user can rapidly switch from small to large scales. The scaled curvature value q is defined as

$$q = 2^{\tau} q_1$$

Since t_1 is a normalized 2-dimensional vector, its components lie in the $[-1, 1]$ range. We use a simple affine transform to map t_1 to the $[0, 1]$ range:

$$t = \frac{1}{2}((1, 1) + t_1)$$

After that, the packed values of q_1 and t_1 are rasterized to the screen, following the natural rendering pipeline. The packed values are interpolated between the vertices by the GPU. This will pass the information automatically to an off-screen buffer, which will be input to the fragment shader in the second stage.

B. Image-space stage

The main difference of our method is that we do not extract apparent ridges by computing the zero-crossings of $D_{t_1} q_1$. Instead, since t_1 is a screen vector by definition, we find the maxima of q_1 in the t_1 direction in image space. We perform a simple *edge detection* in the off-screen buffer computed in the first stage with a standard fragment shader technique for doing image processing on the GPU [19].

More precisely, the edge detection is done with a 3×3 Laplacian-like adaptive filter that considers the t_1 direction. This filter gives more weight to the pixels that are more aligned with the t_1 direction. The t_1 direction is quantized into one of four main directions and the filter is set for the chosen direction (see Figure 4). The filter is weighted by a λ parameter that controls how sensitive the filter is to the θ direction. When $\lambda = 0$, the filter ignores the t_1 direction and becomes a Laplacian filter.

At the end of the process, the filtered value of the curvature is used as an apparent ridge intensity, which we use as a gray-scale pixel value. This produces a line fading effect, similar to the one used in object-space methods of suggestive contours and apparent ridges. We invert the color intensity to produce "black on white" effect, otherwise we would have "white on black". The fading effect can be avoided by setting the pixel value to black when intensity is higher than a minimum threshold (see Figure 5). Low values are clamped to avoid noise artifacts produced by the filter. High λ values are more sensitive to noise and may require a higher clamping value.

Figure 5. Results with (left) and without (right) the fading effect.

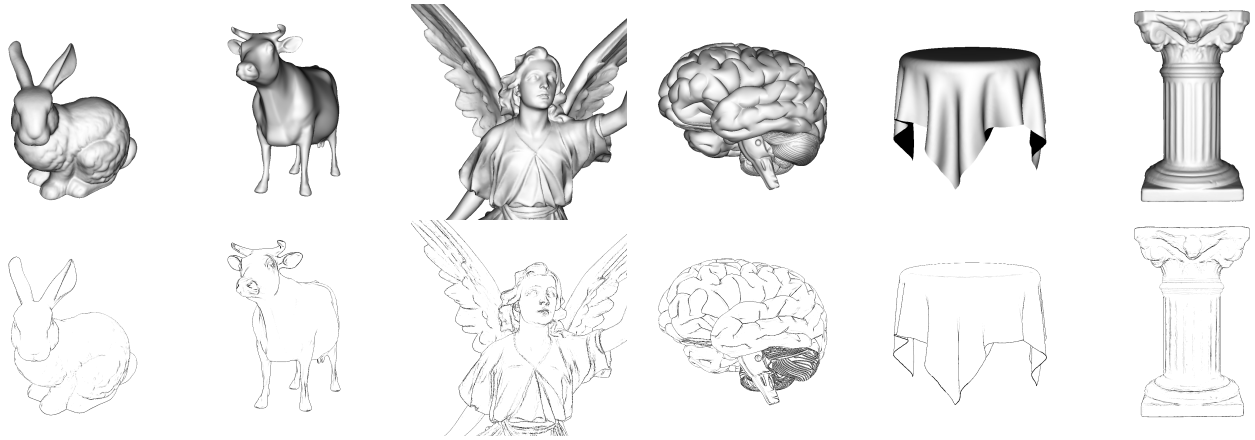

Figure 6. Our results (bottom) on some models along with shaded views (top) for comparison.

V. Results

We now present some of our method's results, discuss the effects of varying our method's parameters (τ and λ), and compare our results side-by-side with the ones obtained with the original method by Judd et al. [11] in terms of image quality and performance. Apparent ridges can depict complex models with very few lines in a clean way. Our results exhibit nice line drawings that capture most of the geometric features of the model. This can be seen comparing each drawing with its respective shaded view (see Figure 6).

A. Parameter variation

To fine-tune the results of our method, one can manipulate the parameters τ and λ to achieve maximum quality.

Figure 7 shows the results of our method for the cow model when τ varies linearly from -9.9 to -2.4. Note how higher τ values give more detailed apparent ridges. However, too high τ values promote loss of precision and the lower maxima are rounded to zero (see the rightmost cow).

Setting λ to a low value such as 2 or 3 usually produces good results for most models. However, because the maxima estimation is performed at the pixel level, some features may be harder to capture when the projected faces of the mesh are much larger than the pixel. Such large faces produce large areas of interpolated curvature. To overcome this problem, higher values of λ can be set. Figure 8 shows the effect of varying λ. Note how the Laplacian filter, chosen by setting $\lambda = 0$, detects most of the apparent ridges, but not the main top ridge. Higher λ values capture this feature and produce sharper apparent ridges.

B. Image comparison

We compared the results of Judd's method with ours for several models (see Figures 9–11) using the following methodology: given a model, we chose an appropriate threshold for Judd's method; then, we chose our method's parameters (τ and λ) so that the resulting image was visually as close as possible to the one produced by Judd's method.

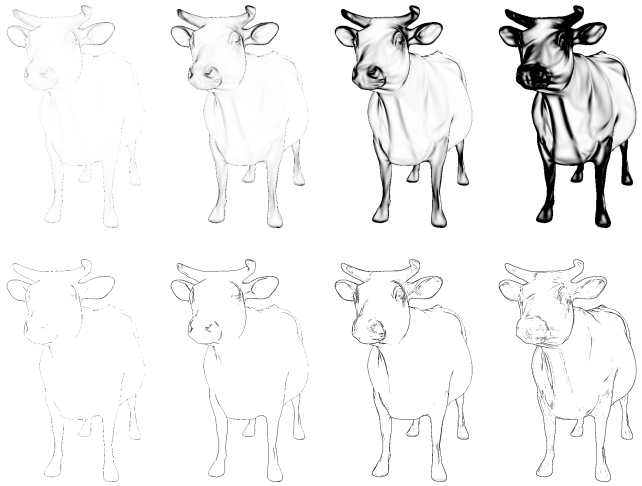

Figure 7. Variation of $\tau = -9.9, -7.4, -4.9, -2.4$, from left to right: scaled view-dependent curvature maps (top) and apparent ridges (bottom).

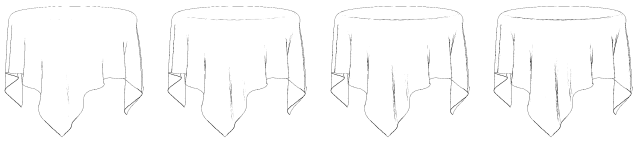

Figure 8. Variation of $\lambda = 0.0, 1.5, 3.0, 4.5$, from left to right.

As it can be seen, our method produces images that are quite similar to the ones produced by Judd's method; apparent ridge lines appear generally in the same places. In some cases, it is very hard to notice the slight differences with bare eyes (especially in a printed version); image closeups reveal pixel-level differences due to the nature of the estimation. In the original method, 3D lines are extracted on the mesh and then rasterized with an arbitrary line size. Our method performs edge detection on the rasterized view-dependent curvature.

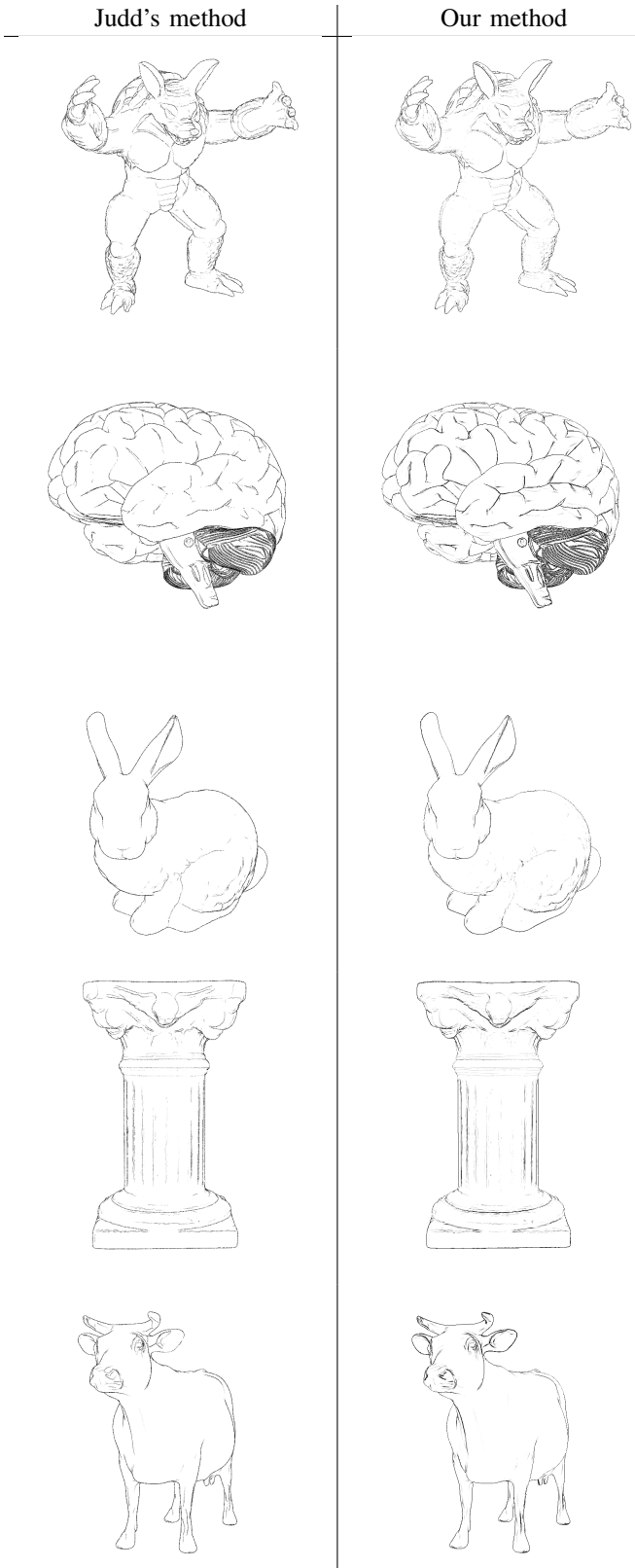

Figure 9. Comparison for armadillo, brain, bunny, column, cow.

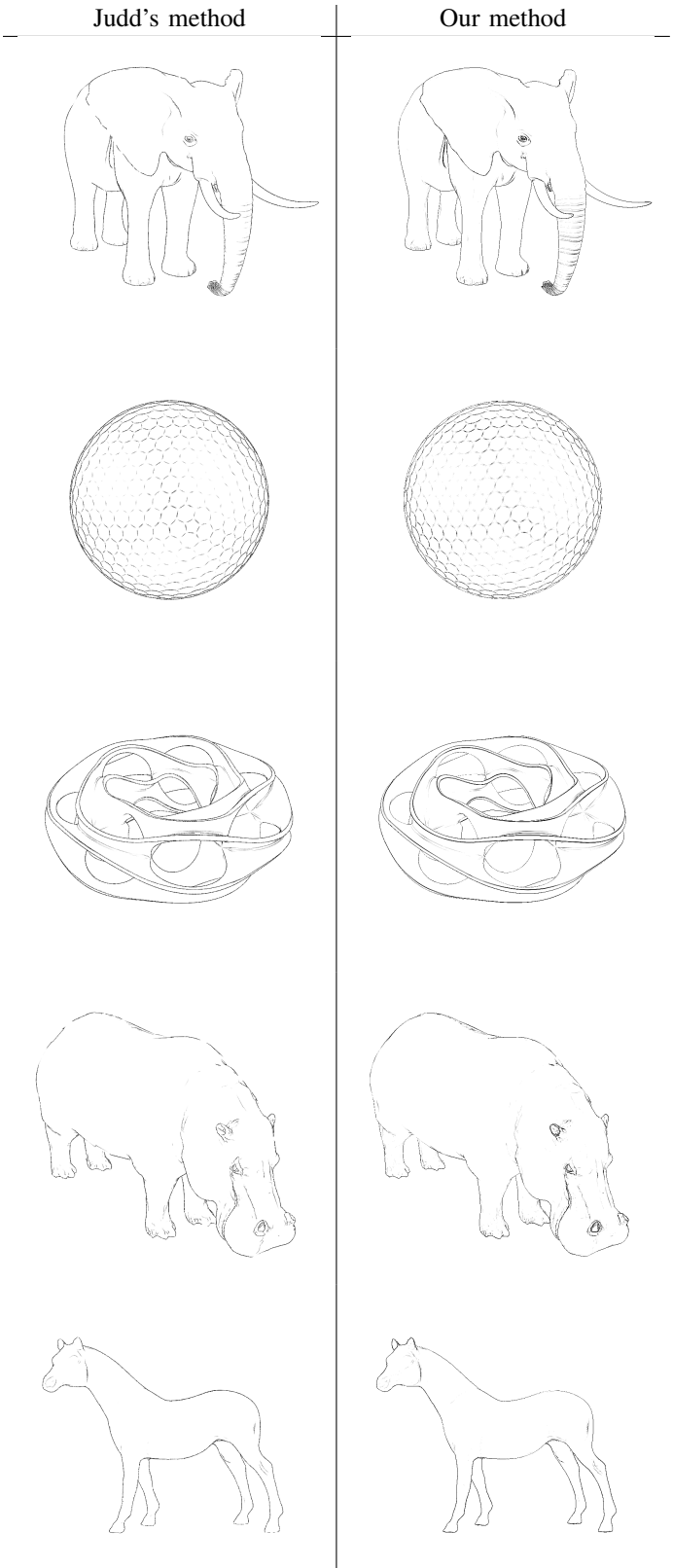

Figure 10. Comparison for elephant, golfball, heptoroid, hippo, horse.

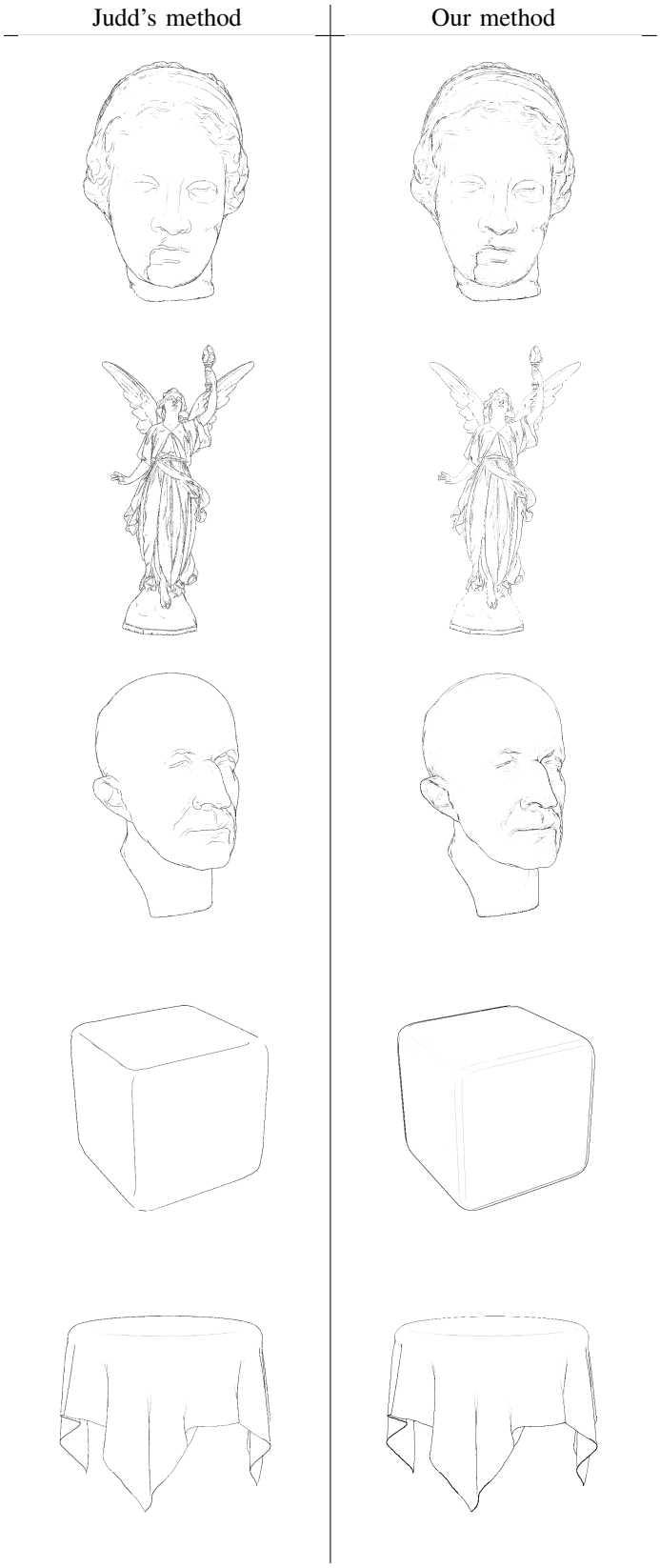

Figure 11. Comparison for igea, lucy, maxplanck, roundedcube, tablecloth.

While the images produced by both methods are equally pleasant, we find ours a little sharper due to the pixel-level estimation, especially for more detailed models. However, for images where the projected face size is much larger than the pixel size, the image quality is worse. In these cases, the excessive interpolation of q_1 and t_1 may produce visual artifacts (see the rounded cube in Figure 11). In general, our method works well for larger models and these artifacts can be eliminated by mesh subdivision if desired.

C. Performance comparison

In the original, object-space approach by Judd et al. [11], apparent ridges tend to be slower to compute than other lines because they rely on the view-dependent curvature and its derivative, which must be recomputed every time the viewpoint changes. As we shall see now, our method improves the performance of apparent ridges, leading them to be competitive in speed with other feature lines, with all its visual advantages.

The main code was written in C++ using the *trimesh2* library [18]. The shaders were written in GLSL [19]. The original apparent ridges code from *rtsc* [20] was adapted to run inside our program for side-by-side and performance comparisons. The vertex shader was also based on code from *rtsc*. The 3D mesh models were collected from the internet [20], [21]. All images are 800×800.

We performed our experiments on two computers: an ordinary laptop with a regular graphics card and a high-end workstation with a powerful graphics card. The laptop had an AMD Turion X2 processor with a 512MB NVidia GeForce 8200M card. The workstation had a dual AMD Opteron processor with a 1.5GB NVidia Quadro FX 5600 card. Both cards have GPU support for vertex and fragment shaders.

The results in Table I show that our method provides significant speedups, except for the smallest model on the laptop (however, the frame rate is still high). In the laptop results, we see that our method performs better for larger models. Indeed, the speedups between the methods grow with the mesh size. Another way of interpreting this result is that the impact of the mesh size is different for the methods, and our method takes advantage of the graphics card to minimize this impact. In the workstation results, our method performs better both in absolute frame rate and relative speedup to Judd's method. However, it is not possible to see a clear tendency of speedup growth with the mesh size. Since Judd's method is CPU-based, we expected that it would be faster on the workstation. Although this indeed occurs, the frame rates of the larger models are almost the same. These results show that CPU-based solutions may not take much advantage of expensive hardware.

Overall, we find that the results are very encouraging and show that GPU-based solutions for line extraction may be a good choice when performance matters, such as NPR rendered games.

model	vertices	laptop J	laptop O	laptop O/J	workstation J	workstation O	workstation O/J
roundedcube	1538	202	90	0.5	600	1100	1.8
tablecloth	22653	32	64	2.0	32	160	5.0
hippo	23105	41	66	1.6	42	160	3.8
cow	46433	24	58	2.4	23	198	8.6
horse	48484	22	52	2.4	26	147	5.6
maxplanck	49132	20	46	2.3	22	90	4.1
bunny	72027	15	42	2.8	16	102	6.4
elephant	78792	14	47	3.4	15	113	7.5
golfball	122882	9	30	3.3	9	52	5.8
igea	134345	8	29	3.6	8	58	7.3
armadillo	172974	5	18	3.6	5	30	6.0
column	262653	3	9	3.0	3	15	5.0
lucy	262909	4	13	3.3	4	26	6.5
heptoroid	286678	4	21	5.3	5	19	3.8
brain	294012	4	20	5.0	4	34	8.5

Table I
PERFORMANCE IN FRAMES PER SECOND (FPS).
J = JUDD'S METHOD, O = OUR METHOD, O/J = SPEEDUP.

VI. CONCLUSION

Apparent ridges are perceptually pleasant and also visually competitive with other lines like suggestive contours by depicting the same features in a clear and smooth way, including convex features. However, apparent ridges are slower to compute since they depend on high-order derivatives of the view-dependent curvature, which change with the viewpoint.

Our method is faster because it replaces the computation of the view-dependent curvature in object space with a simple edge detection in image space, while providing similar image quality. The performance of this stage does not depend on the mesh size, only on the image size. With this improved performance, apparent ridges become even more competitive.

As future work, we intend to experiment with some techniques to improve image quality, such as avoiding direction quantization via sub-pixel sampling, using better filters and a Phong-like shading of the view-dependent curvature.

The image-space stage of our method can be used as part of a pipeline to extract apparent ridges from volume data and implicit models. One would just need to extract the view-dependent curvature from the isosurfaces and rasterize it to an off-screen buffer.

Our method can also be adapted to extract other lines, like suggestive contours. Properties like the radial curvature and its derivative would be rasterized to an off-screen buffer where appropriate screen operations would be applied to find them. We intend to investigate how to remove object space computations completely.

Finally, we would like to explore the use of the view-dependent curvature for shading and modeling.

Acknowledgments: This work is part of the first author's M.Sc. work at IMPA. The second author is partially supported by CNPq. This work was done in the Visgraf laboratory at IMPA, which is sponsored by CNPq, FAPERJ, FINEP, and IBM Brasil.

REFERENCES

[1] B. Gooch and A. Gooch, *Non-Photorealistic Rendering*. A K Peters, 2001.

[2] T. Strothotte and S. Schlechtweg, *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*. Morgan Kaufmann, 2002.

[3] J. J. Koenderink, A. J. van Doorn, C. Christou, and J. S. Lappin, "Shape constancy in pictorial relief," *Perception*, vol. 25, no. 2, pp. 155–164, 1996.

[4] M. C. Sousa and P. Prusinkiewicz, "A few good lines: suggestive drawing of 3d models," *Computer Graphics Forum*, vol. 22, no. 3, pp. 327–340, 2003.

[5] F. Cole, A. Golovinskiy, A. Limpaecher, H. S. Barros, A. Finkelstein, T. Funkhouser, and S. Rusinkiewicz, "Where do people draw lines?" *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–11, 2008.

[6] F. Cole, K. Sanik, D. DeCarlo, A. Finkelstein, T. Funkhouser, S. Rusinkiewicz, and M. Singh, "How well do line drawings depict shape?" *ACM Trans. Graph.*, vol. 28, no. 3, pp. 1–9, 2009.

[7] A. Hertzmann, "Introduction to 3d non-photorealistic rendering," in *Non-Photorealistic Rendering (SIGGRAPH 99 Course Notes)*. ACM, 1999, pp. 7-1–7-14.

[8] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, "Suggestive contours for conveying shape," in *ACM SIGGRAPH 2003*, pp. 848–855.

[9] K. Na, M. Jung, J. Lee, and C. G. Song, "Redeeming valleys and ridges for line-drawing," in *PCM 2005*, Lecture Notes in Computer Science 3767. Springer, 2005, pp. 327–338.

[10] D. DeCarlo and S. Rusinkiewicz, "Highlight lines for conveying shape," in *NPAR '07*. ACM, 2007, pp. 63–70.

[11] T. Judd, F. Durand, and E. Adelson, "Apparent ridges for line drawing," *ACM Trans. Graph.*, vol. 26, no. 3, p. 19, 2007.

[12] T. Isenberg, B. Freudenberg, N. Halper, S. Schlechtweg, and T. Strothotte, "A developer's guide to silhouette algorithms for polygonal models," *IEEE Computer Graphics and Applications*, vol. 23, pp. 28–37, 2003.

[13] J. J. Koenderink, *Solid Shape*. MIT Press, 1990.

[14] V. Interrante, H. Fuchs, and S. Pizer, "Enhancing transparent skin surfaces with ridge and valley lines," in *Visualization '95*. IEEE Computer Society, 1995, pp. 52–59.

[15] D. DeCarlo, A. Finkelstein, and S. Rusinkiewicz, "Interactive rendering of suggestive contours with temporal coherence," in *NPAR '04*. ACM, 2004, pp. 15–145.

[16] Y. Lee, L. Markosian, S. Lee, and J. F. Hughes, "Line drawings via abstracted shading," in *ACM SIGGRAPH '07*, p. 18.

[17] S. Rusinkiewicz, "Estimating curvatures and their derivatives on triangle meshes," in *3DPVT '04*. IEEE Computer Society, 2004, pp. 486–493.

[18] ——, "trimesh2 library," 2009, http://www.cs.princeton.edu/gfx/proj/trimesh2/.

[19] R. Wright, B. Lipchak, and N. Haemel, *OpenGL Superbible*, 4th ed. Addison-Wesley Professional, 2007.

[20] "Suggestive contours," http://www.cs.princeton.edu/gfx/proj/sugcon/.

[21] "Apparent ridges for line drawings," http://people.csail.mit.edu/tjudd/apparentridges.html.