The Visual Computer *(SIBGRAPI'10 Special Issue)* manuscript No.
(will be inserted by the editor)

A Fast Hybrid Method for Apparent Ridges

Eric Jardim · Luiz Henrique de Figueiredo

last updated April 13, 2011 at 4:08 P.M.

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

Keywords expressive lines · non-photorealistic rendering

1 Introduction

Expressive line drawing of 3D models is a classic artistic technique and remains an important problem in non-photorealistic rendering [8,20]. A good line drawing can convey the model's geometry without using other visual cues like shading, color, and texture [14]. Frequently, a few good lines are enough to convey the main geometric features [3,4,19]. The central problem is how to define mathematically what good lines are: ideally, they should capture all perceptually relevant geometric features of the object and they should depend on how the object is viewed by the observer.

There are several techniques (e.g., [6,7,9,12,16]) for expressive line rendering of 3D models, but no single method has emerged as the best for all models and viewing positions [3,4]. Apparent ridges [12] 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 [12] is CPU-based and works over the mesh entirely in object space, our method combines object-space and image-space computations and runs partially on the GPU, producing faster results in real time.

IMPA – Instituto Nacional de Matemática Pura e Aplicada, Rio de Janeiro, Brazil. E-mail: ejardim@impa.br, lhf@impa.br

2 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 Fig. 1.

Object contours or *silhouettes* [11] 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 [9]. 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 [10,13] 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 [16].

Suggestive contours [5,6] 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* [7] 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 [12] were proposed recently and produce 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 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. [15] 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.

3 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 now review how the view-dependent curvature is defined and computed. For details, see Judd et al. [12].

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

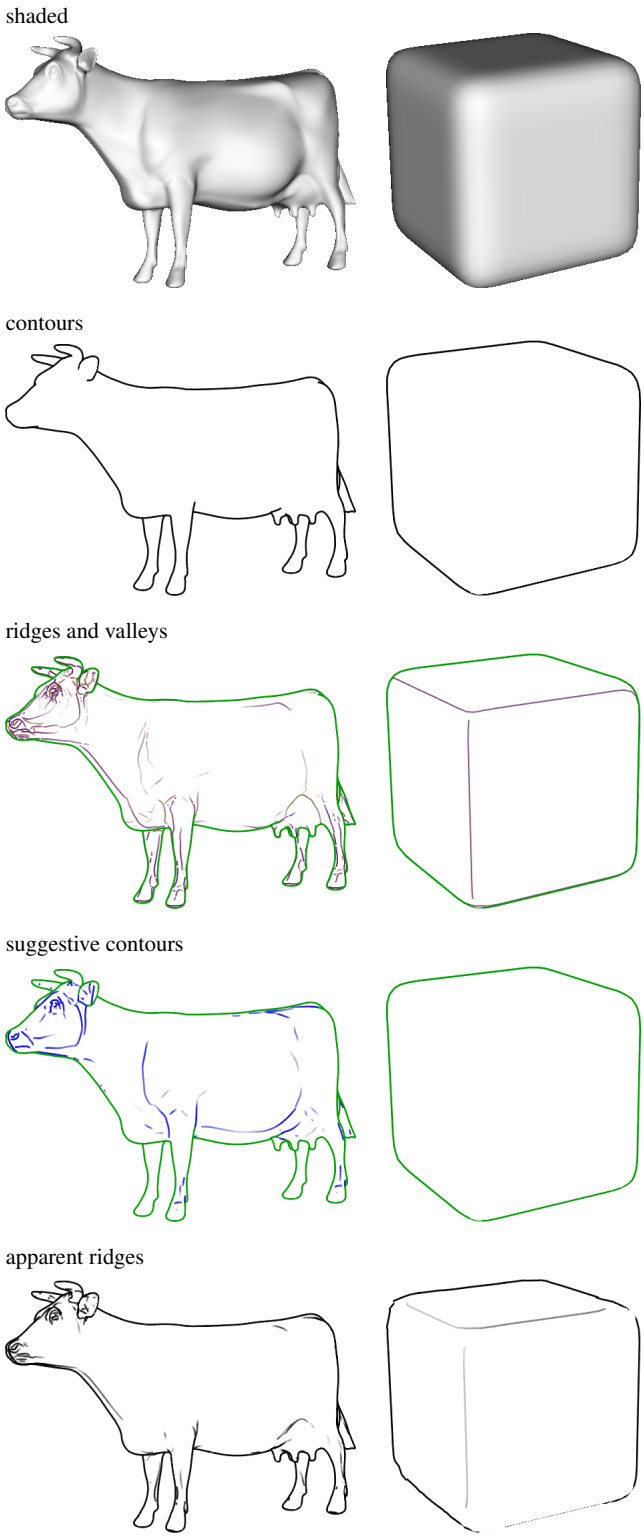

Fig. 1 Comparison of expressive lines for two 3D models. Contours are essential, but insufficient to depict a shape. Ridges and valleys extend contours, but angles are too sharp and appear at rigid places due to their view independence (contours in green). Suggestive contours smoothly complement contours 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.

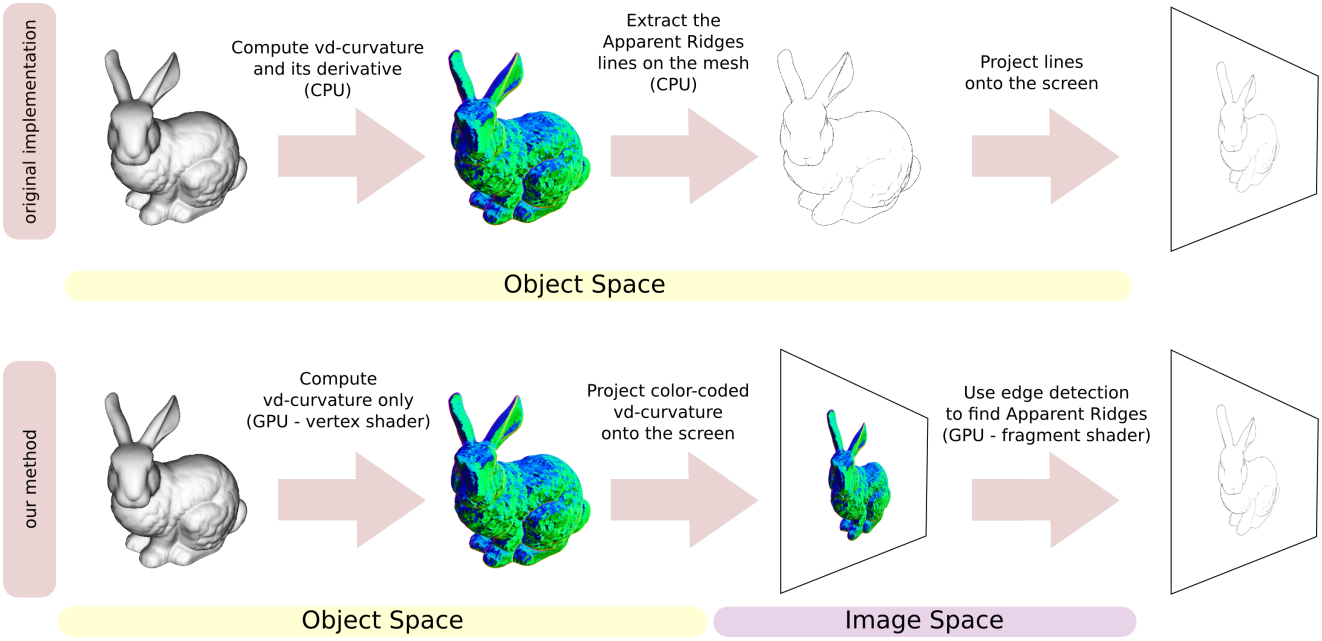

Fig. 2 The original method happens in object space and runs entirely on the CPU. Our method is a hybrid method that runs partially on the GPU.

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.

4 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. [12] 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 Fig. 2). We now explain the modifications needed in their approach and some implementation details to achieve this goal.

As seen in Section 3, 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. [12] 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 object-space method: the computation of derivatives and the detection of its zero crossings is done over the whole mesh 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 Fig. 2). In the first stage, which happens in object space, we estimate the view-dependent curvature data over the mesh and encode it into an image. In the second stage, which happens in image space, we extract apparent ridges on the visible part of the model using edge detection; the required derivatives are computed at each pixel of the image output by the first stage.

While the performance of the first stage depends on the mesh size, the performance of the second stage depends only on the image 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 speedup, which we shall discuss in Section 5.

4.1 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 et al. [12]. The vertex shader is executed every time the viewpoint changes. However, the 3D data required for estimating q_1 and t_1 does not depend on the viewpoint and is computed only once.

More precisely, the normal n, the principal curvatures k_1 and k_2, and the first principal direction e_1 are estimated on the CPU using a technique by Rusinkiewicz [17] implemented in the *trimesh2* library [18]. This data is passed to the vertex shader as follows: n as the vertex normal, k_1 in the red channel of the primary vertex color, k_2 in the green channel of the primary vertex color, and e_1 as the secondary vertex color. The second principal direction e_2 is computed in the vertex shader as $n \times e_1$; this helps to reduce the amount of data transferred from the CPU to the GPU.[1]

From q_1 we compute a scaled curvature value $q = f^2 q_1$, where f is the feature size of the mesh. This is equivalent to the scaling of the threshold done by Judd et al. [12] to make it dimensionless.

The values of q and t_1 are rasterized to an off-screen floating-point framebuffer object (FBO) using one channel for q and two channels for t_1 (a 2D screen vector). The values between the vertices are interpolated at the pixels by the GPU. This framebuffer will be input to the fragment shader in the second stage. Using a floating-point framebuffer avoids having to clamp q to the interval $[0,1]$ and provides enough precision for the edge detection. As mentioned in Section 3, q_1 achieves extremely high values near the contours, and so does q.

4.2 Image-space stage

The second stage is run in a fragment shader using a standard technique for image processing on the GPU [21]. We draw a quad covering the screen using as texture the framebuffer computed in the first stage. This gives the values of q and t_1 for each screen pixel because the GPU performs the necessary interpolation.

Like Judd et al. [12], we extract apparent ridges by finding the zero-crossings of $D_{t_1} q_1$. The main difference is that we find those zero-crossings at the pixel level. More precisely, we find the local maxima of q_1 in the t_1 direction in image space by using *edge detection*.

We estimate $D_{t_1} q_1$ at a pixel p using finite differences $\Delta q^+ = q^1 - q^0$, $\Delta q^- = q^0 - q^{-1}$, where $q^k = q(p + k t_1)$ (see Fig. 3). We detect a decreasing zero-crossing when $\Delta q^- > 0$ and $\Delta q^+ < 0$. In this case, if q^0 is greater than a user-selected threshold τ, the pixel is painted with an intensity of $v = (q^0 - \tau)/q^0$. This produces a nice line fading effect, similar to the one in the object-space method (see Fig. 4). This fading effect can be disabled by setting the pixel to black. Because near the contours the estimated value q^0 is not always high enough, we actually set $v = \max(v, L)$, where $L = \Delta q^- - \Delta q^+ = 2q^0 - q^1 - q^{-1}$ estimates the (negative of the) Laplacian of q at p. To enhance edges, we can also use central differences $\Delta q^+ = q^2 - q^0$, $\Delta q^- = q^0 - q^{-2}$ instead of forward differences (see Fig. 5). This choice also affects the value of the Laplacian L.

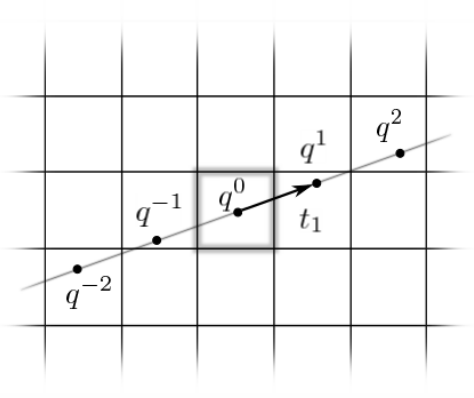

Fig. 3 Sampling q for estimating $D_{t_1} q_1$.

Fig. 4 Results with (left) and without (right) the fading effect.

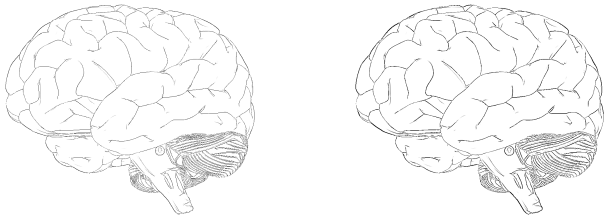

Fig. 5 Results with forward (left) and with central differences (right).

[1] If further data reduction is necessary, it is possible to pack k_1, k_2, and the x and y coordinates of e_1 in the primary vertex color, and compute the z coordinate of e_1 in the shader using that e_1 has norm 1.

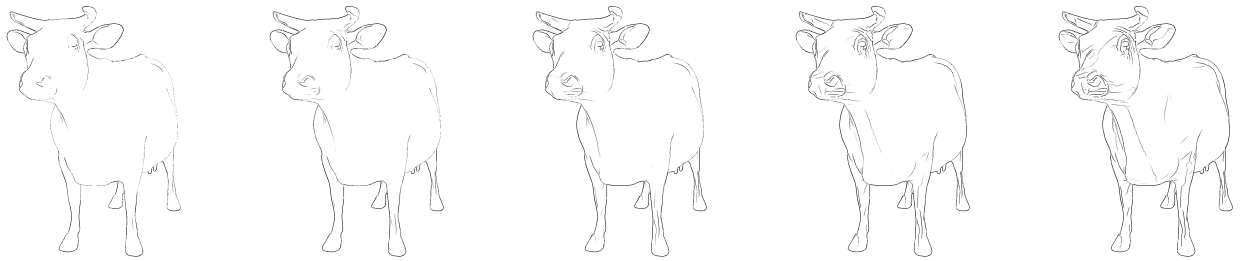

Fig. 6 Threshold variation from left to right: $\tau = 0.43, 0.21, 0.10, 0.05, 0.02$.

5 Results

As expected from apparent ridges, our method generates nice line drawings that capture most of the geometric features of the model. Compare each line drawing with the corresponding shaded view in Fig. 7. Further results can be seen in Fig. 8 and 9. We shall now discuss the effects of varying the threshold τ and compare our results with the ones obtained with the object-space method by Judd et al. [12] in terms of image quality and especially performance.

All images generated by our method used central differences to enhance edges. All images generated by Judd's method used a line width of 4.

5.1 Threshold variation

Like Judd et al. [12], we use a single threshold τ to control what lines are shown and which the user can tune to improve image quality. Fig. 6 shows the results of our method on the cow model for increasing values of τ. Low values of τ mainly capture just the contours. Apparent ridges appear in detail as τ increases, but when τ is very high the image has too many lines. In practice, there is a clear range of suitable values of τ that the user can explore interactively to create a good image.

All images generated by our method used a threshold of 0.10, except for these: *buddha* 0.02, *column* 0.05, *ecat* 0.06, *golfball* 0.21, *minerva* 0.06, *tablecloth* 0.21, *turtle* 0.21.

5.2 Image comparison

Our main goal was to increase the performance of apparent ridges, not to reproduce the exact results obtained by Judd et al. [12]. Nevertheless, we did compare the results of both methods, as follows. Given a model and a viewpoint, we chose an appropriate threshold for the object-space method, which was then used in our method. Although in principle both methods compute the same lines, they do so differently and the lines are rendered differently: the object-space method draws lines in space, whereas our methods paint pixels directly on the image. Nevertheless, in all cases tested, the same threshold was suitable for both methods.

As illustrated in Fig. 8, the images produced by both methods are quite similar; apparent ridge lines appear mostly in the same places. Some visual differences appear because the approximations used in the two methods are not exactly the same. In particular, the fading effect is slightly different: while Judd et al. [12] use only values of q_1 above a certain threshold, we also use an estimate of the Laplacian when q_1 is not high enough. Other visual differences appear in large models with small triangles. In the object-space method, apparent ridge lines in small triangles are projected into a single pixel. Our hybrid model extract lines from the projected model and so is less sensitive to small triangles.

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, our images may seem worse. In these cases, the excessive interpolation of q_1 and t_1 may produce visual artifacts. In general, our method works well for larger models and these artifacts can be eliminated by mesh subdivision if desired. (This subdivision could be done on the GPU.)

5.3 Performance comparison

We ran timing experiments for rendering all models shown in Fig. 7. on a 2.67GHz Intel Core i7-920 Linux machine with 6GB of RAM and a NVidia GeForce GTX 480 card. We implemented our method in C++ using the *trimesh2* library [18] with support for OpenMP enabled. The shaders were written in GLSL [21]. The original apparent ridges code from *rtsc* [2] 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 [1, 2]. All images are 1024×1024. Full-size images and code are available at `http://w3.impa.br/~lhf/har/`.

The results in Table 1 show that our hybrid method provides significant speedup (except for the smallest model), even when the object-space method uses multiple cores. As we expected, our method performs much better for larger models and the speedup grows with the mesh size.

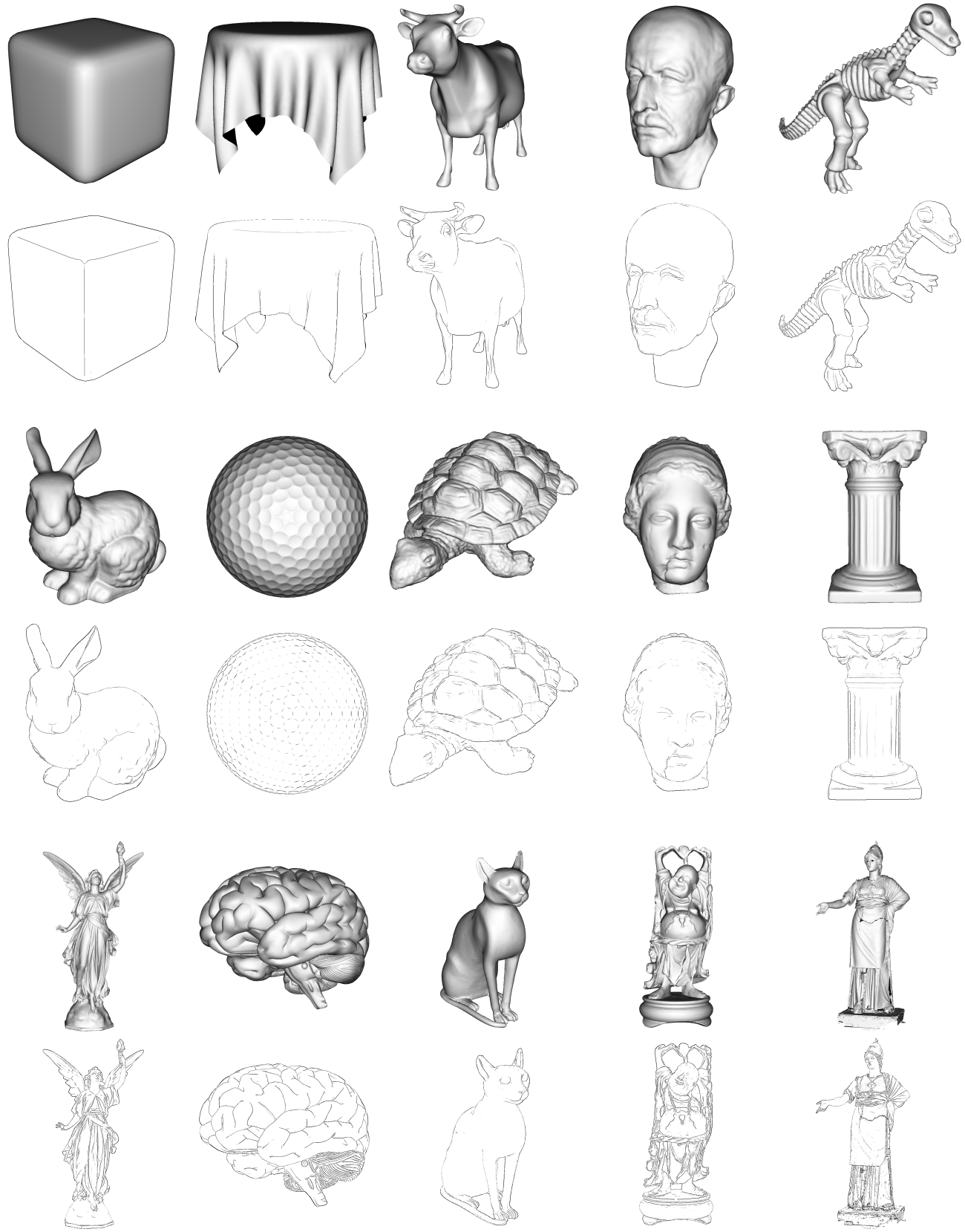

Fig. 7 The models used in our experiments: shaded views (top), apparent ridges computed with our method (bottom). Full-size images are available at `http://w3.impa.br/~lhf/har/`.

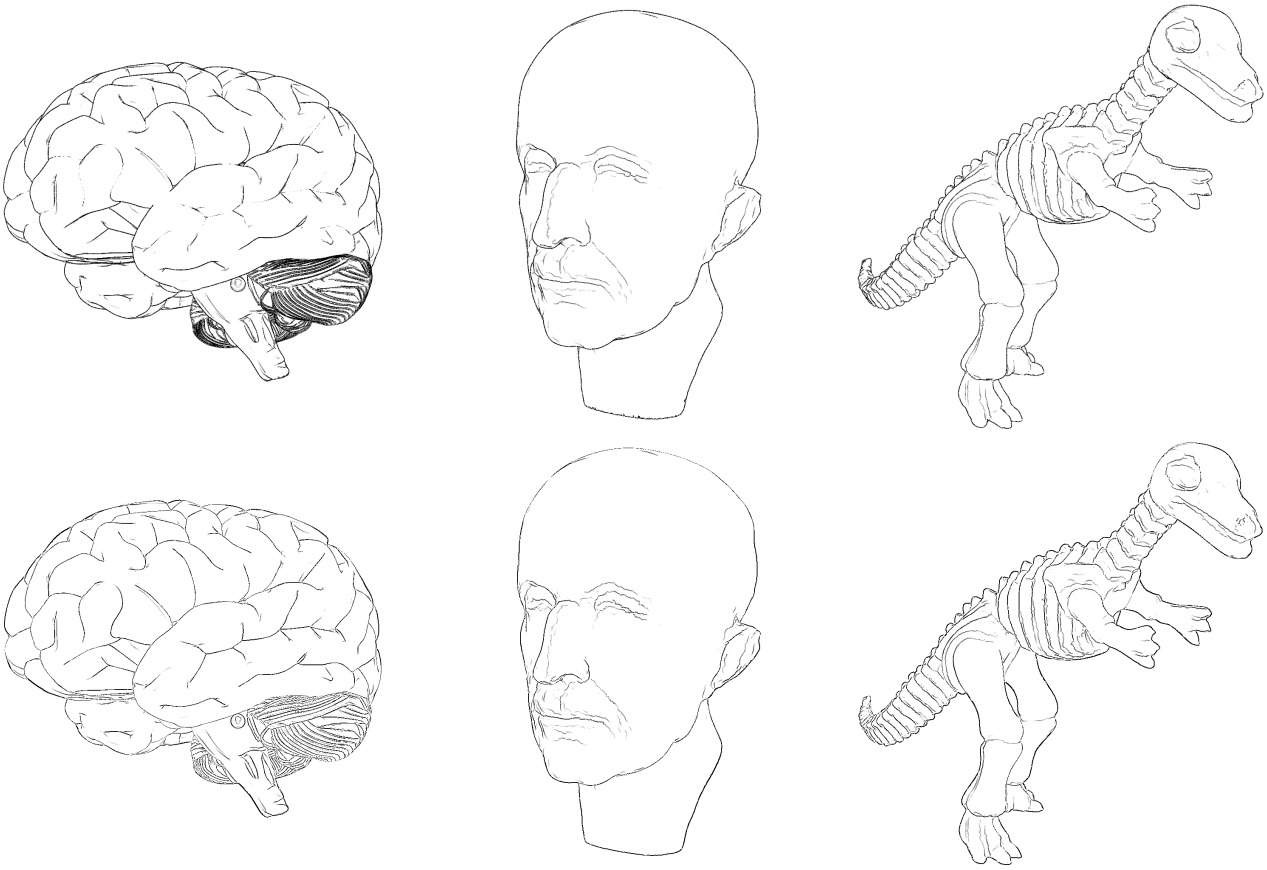

Fig. 8 Image comparison: original object-space method (top) and our hybrid method (bottom).

Fig. 9 Further results obtained with our hybrid method.

model	vertices	object-space	hybrid	speedup
roundedcube	1538	1582.3	1576	1
tablecloth	22653	183.2	1456	8
cow	46433	123.7	1357	11
maxplanck	49132	109.6	1276	12
dinosaur	56194	64.7	1315	20
bunny	72027	74.2	1194	16
golfball	122882	43.5	911	21
turtle	134057	29.7	980	33
igea	134345	40.6	941	23
column	262653	16.4	732	45
lucy	262909	15.7	733	47
brain	294012	16.2	634	39
ecat	342246	10.4	628	60
buddha	543652	7.0	463	66
minerva	830288	3.9	341	87

Table 1 Performance in frames per second (fps).

6 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, with the original object-space method of Judd et al. [12], apparent ridges are slower to compute because they need expensive computation that is performed over the whole mesh every time the viewpoint changes.

Our method provides images of similar quality and is faster than the original method because it computes the view-dependent curvature on the GPU and finds its directional derivatives in image space and their zero crossings using edge detection. The performance of the image-space stage does not depend on the mesh size, only on the image size. With this improved performance, apparent ridges become even more competitive, especially for large meshes.

We find that the results are very encouraging and show that GPU-based solutions for line extraction may be useful when performance matters, such as in NPR-rendered games and scientific visualization. The high frame rate allows further processing to take place on both the CPU and the GPU.

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 probably be adapted to extract other feature lines, such as suggestive contours. Properties like the radial curvature and its derivative would be rasterized to an off-screen buffer and appropriate screen operations would be applied to find those lines.

Finally, we intend to investigate how to remove object-space computations completely.

Acknowledgements We thank Waldemar Celes, Diego Nehab, and the referees for their comments and suggestions. A previous version of this paper was presented at SIBGRAPI 2010 and was based on 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. Apparent ridges for line drawings. `http://people.csail.mit.edu/tjudd/apparentridges.html`
2. Suggestive contours. `http://www.cs.princeton.edu/gfx/proj/sugcon/`
3. Cole, F., Golovinskiy, A., Limpaecher, A., Barros, H.S., Finkelstein, A., Funkhouser, T., Rusinkiewicz, S.: Where do people draw lines? ACM Trans. Graph. **27**(3), 1–11 (2008)
4. Cole, F., Sanik, K., DeCarlo, D., Finkelstein, A., Funkhouser, T., Rusinkiewicz, S., Singh, M.: How well do line drawings depict shape? ACM Trans. Graph. **28**(3), 1–9 (2009)
5. DeCarlo, D., Finkelstein, A., Rusinkiewicz, S.: Interactive rendering of suggestive contours with temporal coherence. In: NPAR '04, pp. 15–145. ACM (2004)
6. DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., Santella, A.: Suggestive contours for conveying shape. In: ACM SIGGRAPH '03, pp. 848–855 (2003)
7. DeCarlo, D., Rusinkiewicz, S.: Highlight lines for conveying shape. In: NPAR '07, pp. 63–70. ACM (2007)
8. Gooch, B., Gooch, A.: Non-Photorealistic Rendering. A K Peters (2001)
9. Hertzmann, A.: Introduction to 3d non-photorealistic rendering. In: Non-Photorealistic Rendering (SIGGRAPH 99 Course Notes), pp. 7–1–7–14. ACM (1999)
10. Interrante, V., Fuchs, H., Pizer, S.: Enhancing transparent skin surfaces with ridge and valley lines. In: Visualization '95, pp. 52–59. IEEE Computer Society (1995)
11. Isenberg, T., Freudenberg, B., Halper, N., Schlechtweg, S., Strothotte, T.: A developer's guide to silhouette algorithms for polygonal models. IEEE Computer Graphics and Applications **23**(4), 28–37 (2003)
12. Judd, T., Durand, F., Adelson, E.: Apparent ridges for line drawing. ACM Trans. Graph. **26**(3), 19 (2007)
13. Koenderink, J.J.: Solid Shape. MIT Press (1990)
14. Koenderink, J.J., van Doorn, A.J., Christou, C., Lappin, J.S.: Shape constancy in pictorial relief. Perception **25**(2), 155–164 (1996)
15. Lee, Y., Markosian, L., Lee, S., Hughes, J.F.: Line drawings via abstracted shading. In: ACM SIGGRAPH '07, p. 18 (2007)
16. Na, K., Jung, M., Lee, J., Song, C.G.: Redeeming valleys and ridges for line-drawing. In: PCM 2005, Lecture Notes in Computer Science 3767, pp. 327–338. Springer (2005)
17. Rusinkiewicz, S.: Estimating curvatures and their derivatives on triangle meshes. In: 3DPVT '04, pp. 486–493. IEEE Computer Society (2004)
18. Rusinkiewicz, S.: trimesh2 library, version 2.10 (December 2010). `http://www.cs.princeton.edu/gfx/proj/trimesh2/`
19. Sousa, M.C., Prusinkiewicz, P.: A few good lines: suggestive drawing of 3d models. Computer Graphics Forum **22**(3), 327–340 (2003)
20. Strothotte, T., Schlechtweg, S.: Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation. Morgan Kaufmann (2002)
21. Wright, R., Lipchak, B., Haemel, N.: OpenGL Superbible, fourth edn. Addison-Wesley Professional (2007)