Instituto Nacional de Matemática Pura e Aplicada

A Hybrid Method for Computing Apparent Ridges

Eric Jardim

Thesis Advisor: Luiz Henrique de Figueiredo

February 2010

Abstract

We propose a hybrid method for computing apparent ridges on triangle meshes. Our method combines both object-space and image-space computations and runs partially in the GPU, taking advantage of modern graphic cards processing power and producing faster results in real time.

Resumo

Neste trabalho é proposto um método híbrido para extrair *apparent ridges* em malhas triangulares. Este método combina operações tanto no espaço do objeto como no espaço da imagem e é executado parcialmente na GPU, aproveitando o poder de processamento das placas gráficas modernas e produzindo resultados mais rápidos em tempo real.

Dedication

This work is dedicated to my son.

Acknowledgments

I want to thank all people that somehow contributed to make this work possible, in special my advisor Luiz Henrique, who was very patient and supportive every time I needed.

Finally, I want to thank my parents for all the support they gave me along my life. Thank you all!

Contents

1 Introduction **3**

1.1 Realism vs. Expressiveness 4
1.2 Line Drawings 5
1.3 Organization 7

2 Previous Work **8**

2.1 Contours 8
2.2 Ridges and Valleys 9
2.3 Suggestive Contours 10
2.4 Apparent Ridges 11

3 Apparent Ridges **12**

3.1 Curvature Basics 12
3.2 View-Dependent Curvature 13
3.3 Definition 15
3.4 Contours 16

4 Our Method **18**

4.1 Motivation 18
4.2 Characterization 18
4.3 Why a Hybrid Method? 19
4.4 Overview of Our Method 20
4.5 Object-Space Stage 20
4.6 Image-Space Stage 21

5 Results **24**

5.1 Parameter Variation 25
5.2 Image Comparison 27
5.3 Performance Comparison 35

6 Conclusions and Future Work **37**

A Shader codes **40**

Chapter 1

Introduction

Since the beginning of computer graphics, one of its main goals and major efforts was to produce convincing and compelling machine-generated images [7]. In that moment, like artists and illustrators, computers could also produce pictures of arbitrary shapes and models. Naturally, the first attempts produced coarse results (see figure 1.1).

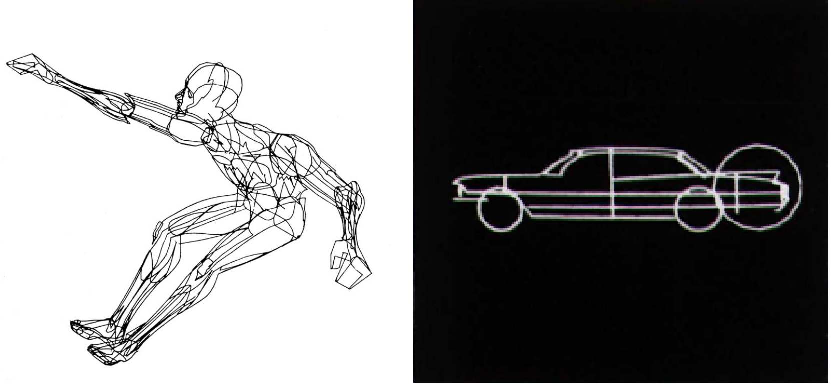

Figure 1.1: Fetter's "First Man" [1] used for cockpit simulations at Boeing and a sample of DAC-1, one of the first CAD systems developed by General Motors [2].

However, a few decades later, with new rendering techniques arising and increasing processing power due to the technology evolution, the realism in computer graphics image grew very fast and achieved a level so high in the present days that sometimes it is difficult to tell whether an image is artificial or not (try figure 1.2).

Figure 1.2: Can you tell which image is real or not? (extracted from [3])

1.1 Realism vs. Expressiveness

The quest for realism led to the widespread experience of new levels of virtual perception in fields like entertainment and engineering, as realistic synthetic images have become able to trick the human brain into seeing things that do not exist or have not been built yet.

However, an issue arose: realism itself is not so efficient in communication. A real picture or scene may contain more information than someone may desire to convey. For example, an illustration of a heart surgery compared to a real picture of that same scene, even with less details can be much more effective for teaching purposes (see figure 1.3).

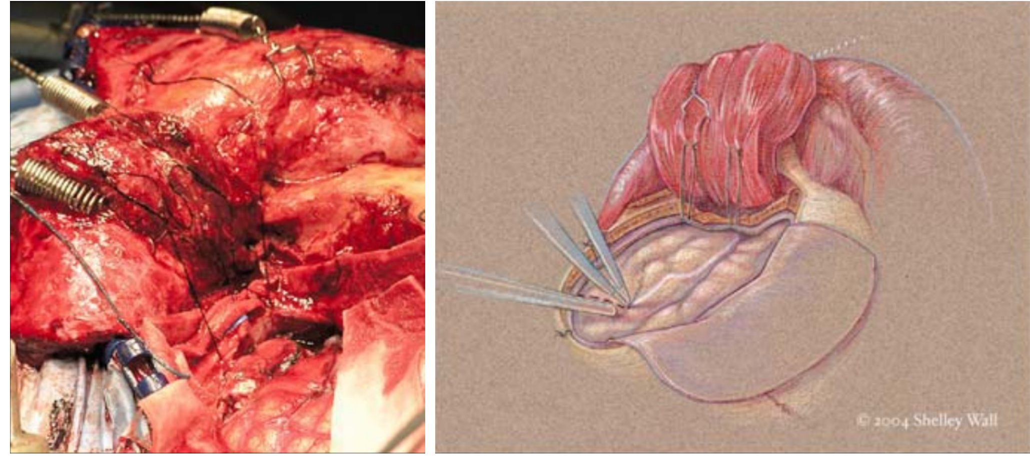

Figure 1.3: Photo vs. illustration (extracted from [4])

Ultimately, an image is a channel of communication and in some situations

Figure 1.4: Changes in image cues can highlight major details and hide minor ones (extracted from an illustration web page [8])

realistic details may act as noise. Artists usually ignore realism when they need expressiveness. To highlight relevant details, illustrators generally appeal to specific techniques to remove excessive detail, manipulating image cues like colors, texture and lighting (see figure 1.4).

Although realism remains a hot topic [5], expressiveness and artistic effects have gained their own space in computer graphics. From this new kind of interest, arose a new field of research called Non-Photorealistic Rendering [6, 7] (briefly called NPR), which plays a complementary role to realism.

1.2 Line Drawings

A line drawing is perhaps the most basic type of artistic expression. It should be the first choice for any simple illustration. With a few strokes, an artist or illustrator can render a compelling image. The compact and clear aspect of a good line drawing ensures the effectiveness in communication.

Although some artistic masterpieces have been created with the exclusive use of lines (see the left drawing in figure 1.5), line drawings are often used as a sketch or simplification of a complex drawing. That means that they are useful to understand and design pictorial representations.

Expressive line drawing of 3D models is a classic artistic technique and remains an important problem in NPR [6, 7]. A good line drawing can convey the shape geometry without using other cues like shading, color, and texture [17]. Frequently, a few good lines are enough to convey the main geometric features [11].

There are several techniques for depicting shapes with lines. The most significant lines have precise mathematical definitions and usually depend on

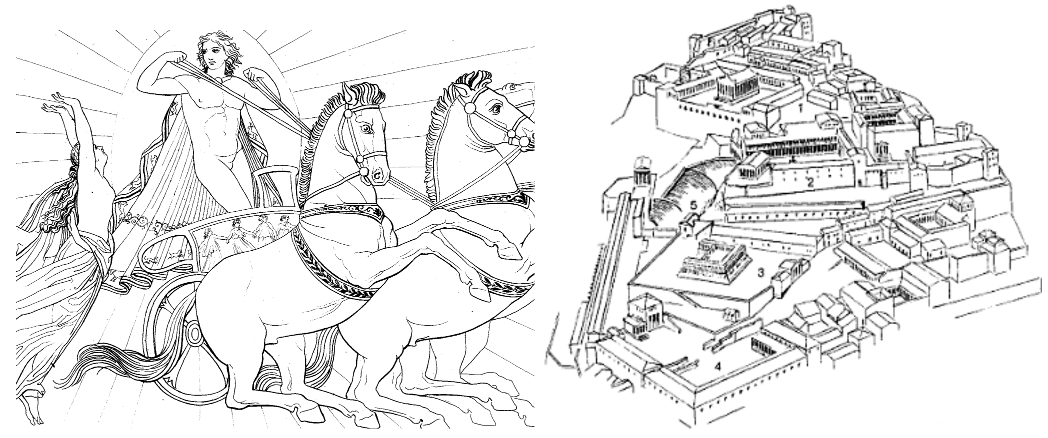

Figure 1.5: An artistic line drawing (extracted from Flaxman's Odyssey [9]) and an architectural drawing (from Koch [10])

the surface geometry and the viewing point.

Apparent ridges [12, 13] are lines defined on the surface of a 3D model. These lines are quite interesting for two reasons: they have a relatively simple mathematical definition and they capture most of the geometric features in a perceptually pleasant way (see fig. 1.6).

Figure 1.6: Sample drawings of apparent riges extracted from 3D models using our method

Apparent ridges were defined by Judd in her M. Sc. thesis [13] along with an implementation for triangle meshes. We propose here a modified version of that implementation, adapted to run mostly on the GPU. Our method

achieve results up to 8 times faster than Judd's method without compromising image quality.

1.3 Organization

The next chapter deals with previous works. There will be shown some feature lines, their issues and how they relates with apparent ridges itself. Chapter 3 contains the mathematical definition of apparent ridges. In chapter 4 our method is explained with all the implementation details. Our results are evaluated in chapter 5, compared side-by-side with the those from the original work done by Judd et al. [12, 13]. We conclude in chapter 6, proposing several extension of this work. The codes for the GPU shaders are in Appendix A.

Chapter 2

Previous Work

In the literature, there are several works on line definitions and on methods for extracting them. This chapter reviews the most important curves related to our work and apparent ridges themselves. For comparison, these lines are going to be extracted from two sample models: the cow model, which presents many curvature variations and the rounded cube model, which is a typical convex shape. Both can be seen below, rendered with a simple Lambertian shading.

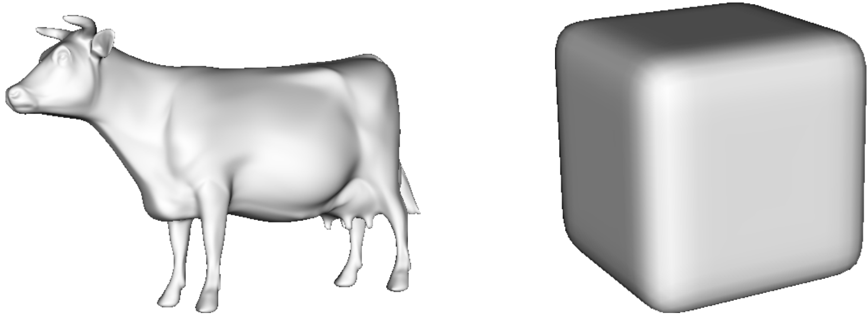

Figure 2.1: Shaded views of the cow and rounded cube models

2.1 Contours

Contours or *silhouettes* are probably the most basic kind of feature line. These lines are known prior to computer graphics, but only recent work have proposed how to extract them [14].

Given a 3D object and a viewpoint, the contours are the loci of points on the surface of the object that separate its visible and invisible parts.

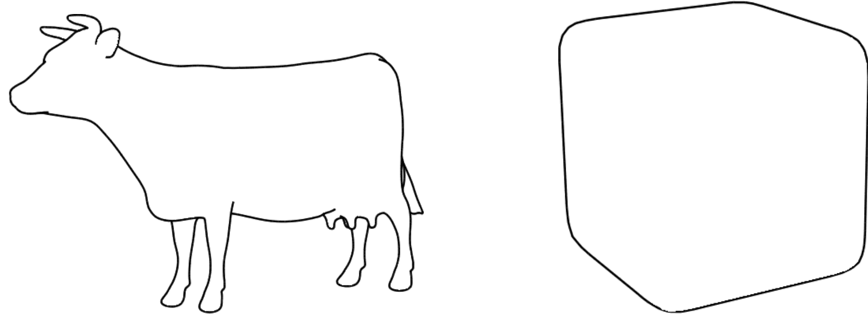

Figure 2.2: Contours are essential, but insufficient lines to depict a shape

Geometrically, it is any visible point p where the viewing vector $v(p)$ is perpendicular to its normal $n(p)$, or simply, where $\langle n(p), v(p) \rangle = 0$. This definition makes contours view-dependent lines, that is, they depend on the viewpoint, since $v(p)$ is a vector from the viewpoint to p.

In general, contours alone are not enough to capture all perceptually relevant features of an object (see figure 2.2). Comparing these images to the shaded ones it is possible to see that many details were not captured.

On the other hand, many other lines, like ridges & valleys and suggestive contours, must be combined with contours to present pleasant and perceptually complete pictures. Although contours are not enough to depict a shape, they are always present in traditional computer line drawings.

Moreover, contours are considered first-order curves, because they only depend on the normal variation.

2.2 Ridges and Valleys

Ridges and valleys [15] are found by computing principal curvatures, principal directions and their derivatives. They are considered second-order curves, because they depend on the surface curvature, which is obtained by differentiating the normal.

Ridges and valleys complement contour information because they add second-order information capturing elliptic and hyperbolic maxima on the surface. Since their definition only takes into account the geometry of the model, ridges and valleys are not view-dependent lines. A downside of these lines is that they often present sharp angles even when the model is smooth (see the cow in figure 2.3). Also, some models have so many ridges and valleys that the resulting image may not look like a clean drawing.

Although some still images showing ridges and valleys may look fine (like

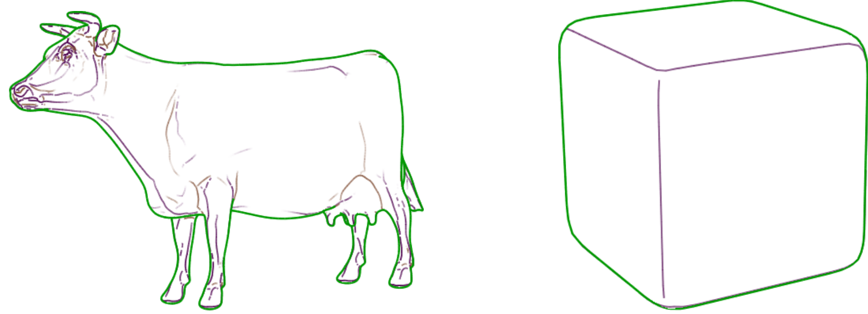

Figure 2.3: Ridges & valleys extend contours, but angles are too sharp and appear at rigid places due to their view independence (contours in green).

the cube in figure 2.3), animated drawings may frequently present motion artifacts, since these features are rigid, independent of the viewing point.

2.3 Suggestive Contours

Suggestive contours [16] are view-dependent lines that naturally extend contours at the joints. They are more visually pleasant than the previous lines because they combine view dependency and second-order information. This combination provides a cleaner drawing (see figure 2.4). Nevertheless, they still need contours to be visually complete.

Intuitively, suggestive contours are contours in nearby views. More precisely, they are based on the zeros of the radial curvature in the viewing

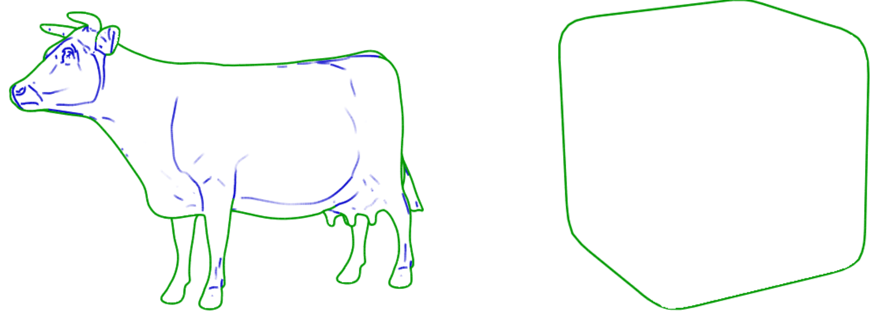

Figure 2.4: Suggestive contours smoothly complement contours in a view-dependent way. However, the cube has no suggestive contours as they do not appear on convex regions (contours in green).

direction projected onto the tangent plane.

However, for the radial curvature to achieve the zero value in some direction, the interval between the principal curvatures must contain zero. So, due to their definition, suggestive contours can not appear where the Gaussian curvature is positive. Thus, convex features cannot be depicted with suggestive contours (see the cube in figure 2.4).

2.4 Apparent Ridges

Apparent ridges [12] are more recent than the previous lines and combine their good properties producing nice results. With a single mathematical definition, apparent ridges depict most features that ridges and valleys capture, but in a clean and view-dependent way. They also capture features in convex regions, which are missed by suggestive contours (see figure 2.5). Another advantage is that contours are a special case of apparent ridges, which means that they can be extracted together in the same computation.

Apparent ridges are based on a new geometric property: the *view-dependent curvature*, which plays an analogue role for apparent ridges as the principal curvatures do for ridges and valleys. In short, apparent ridges combine second-order information with view-dependency, but also appear at convex regions.

In the next chapter, we will define the view-dependent curvature and apparent ridges.

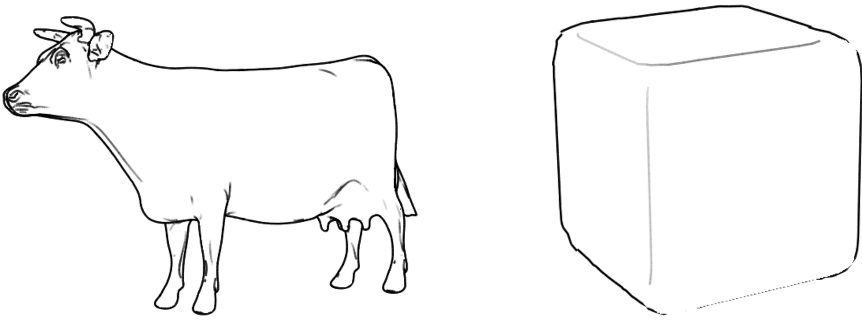

Figure 2.5: Apparent ridges depict features in a smooth and clean view-dependent way, with the advantage of appearing at convex regions. Contours are apparent ridges too.

Chapter 3

Apparent Ridges

The *secret ingredient* of apparent ridges, which allows it to capture features better than other lines, relies on its own definition. The view-dependent curvature is a new geometric property that enables us to perceive curvature in a different way. In this chapter we present the mathematical definitions of the view-dependent curvature and the apparent ridges.

3.1 Curvature Basics

The intuitive notion of curvature is described by how an object bends itself. For complex geometric objects like surfaces, which are the basis of 3D modeling, this curvature notion is not only an intensity measure, but also a qualitative way to describe how the surface bends. In differential geometry, this curvature notion is defined precisely, giving us mathematical tools for calculating this geometric property numerically.

Given a point p on a smooth surface M and a tangent vector r in T_pM, the tangent plane of M at p, it is possible to define the shape operator $S : T_pM \rightarrow T_pM$, as

$$S(r) = D_r n$$

which is the derivative of the normal in the r direction projected onto the tangent plane. This operator linearly maps each tangent vector of M to another tangent vector:

$$\begin{pmatrix} s_x \\ s_y \end{pmatrix} = S \begin{pmatrix} r_x \\ r_y \end{pmatrix} = \begin{pmatrix} a & b \\ b & c \end{pmatrix} \begin{pmatrix} r_x \\ r_y \end{pmatrix}$$

where r_x, r_y and s_x, s_y are the respective vector coordinates, fixed an appropriate basis for the tangent plane.

This operator is known in differential geometry to be the second order derivative tensor of the surface [18] and it describes how the surface bends at p. Since S is a self-adjoint operator, there are two appropriate directions e_1 and e_2 in T_pM which simplify its matrix to

$$S = \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix}$$

where k_1 and k_2 are known to be the *principal curvatures* and e_1 and e_2 are called the *principal directions* of M at p. The principal curvatures give bending intensities at the principal directions, while their product (also known as the Gaussian curvature) gives qualitative information of the bend type, which can vary from elliptic (convex-like) to hyperbolic (saddle-like), depending on the sign of this product.

Along with the shape operator S, we define the *normal curvature* (also known as the radial curvature). Given a unit direction r in the tangent plane at p, the curvature $k(r)$ is the scalar product of $S(r)$ and r

$$k(r) = \langle S(r), r \rangle$$

which measures the curvature of the surface at p in the r direction. It can be shown [18] that $k(e_1) = k_1$, $k(e_2) = k_2$ and $k(r) \in [k_2, k_1]$ for every r, supposing that $k_1 > k_2$.

Ridges and valleys can be computed by finding the extrema of the principal curvature k_1 along its principal direction e_1 (assuming $|k_1| > |k_2|$). These lines mark important features on the surface, but they are not always visually pleasant as they do not consider the viewing point.

Suggestive contours overcome this limitation by finding the zeros of $k(w)$, where w is the viewing vector projected to the tangent plane. Unfortunately, due to what we stated before, suggestive contours cannot appear at convex regions, where the product of k_1 and k_2 is positive, because $0 \notin [k_2, k_1]$.

3.2 View-Dependent Curvature

The key idea of the view-dependent curvature is to measure how the surface bends with respect to the viewpoint. This property takes into account the projection transformation which maps points on the surface to the screen.

Given a surface point $p \in M$, let S be the shape operator at p. Let Π be the parallel projection that maps p onto the screen and $q = \Pi(p)$. If p is not a contour point, then Π is locally invertible and it is possible to define (locally)

an inverse function Π^{-1} that takes points from the screen and maps to the surface. Moreover, if $n(p)$ is the normal at p, then the composite mapping

$$\tilde{n}(q) = n \circ \Pi^{-1}(q)$$

maps the corresponding normal of the surface at p from its projection q.

With this definition, given a screen vector s at q it is possible to define the *view-dependent shape transform* Q

$$Q(s) = D_s \tilde{n}$$

which is a directional derivative similar to the shape operator, but the with respect to a screen vector. This definition is very intuitive, but for computational purposes we will derive an expression of Q that depends on S.

On the tangent plane at p, the Jacobian J_Π of the projection takes tangent vectors to the screen at q (see figure 3.1).

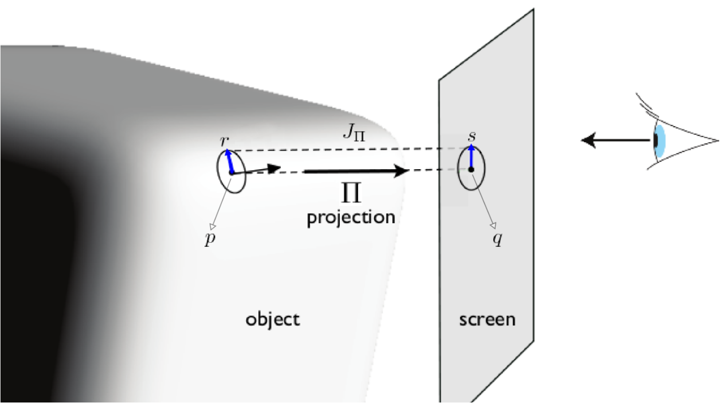

Figure 3.1: The projection setup

If p is not a contour point, J_Π is invertible and we can write $r = J_\Pi^{-1}(s)$. Noting that $D_s \tilde{n} = D_{r(s)} n$, we can deduce that

$$Q(s) = S(J_\Pi^{-1}(s)) \quad \Rightarrow \quad Q = S \circ J_\Pi^{-1}$$

Now we can derive a formula to calculate Q numerically. Given a certain basis s_1, s_2 at a projected point on the screen and taking e_1, e_2 as a basis for the tangent plane, the Jacobian of Π is

$$\begin{pmatrix} \langle e_1, s_1 \rangle & \langle e_2, s_1 \rangle \\ \langle e_1, s_2 \rangle & \langle e_2, s_2 \rangle \end{pmatrix}$$

and putting it all together we derive the formula

$$Q = \begin{pmatrix} k_1 & 0 \\ 0 & k_2 \end{pmatrix} \begin{pmatrix} \langle e_1, s_1 \rangle & \langle e_2, s_1 \rangle \\ \langle e_1, s_2 \rangle & \langle e_2, s_2 \rangle \end{pmatrix}^{-1}$$

As Q takes screen vectors and maps it to tangent vectors at M, Q is not necessarily a self-adjoint operator. Thus, it is not guaranteed that it has two eigenvalues like S. Instead, it does have a maximum singular value

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

that will be defined as the *maximum view-dependent curvature*. This value can be computed as the square root of the maximum eigenvalue of $Q^T Q$. It is important to notice that q_1 is non-negative.

This singular value has an associated direction t_1 on the screen. When s reaches t_1, the norm of $Q(s)$ is maximum. Fixed a screen point, a unit circle of screen vectors will be transformed into a elongated ellipsis on the tangent plane (see figure 3.2). The t_1 direction is called *maximum view-dependent principal direction.*

As an intuitive notion, t_1 and q_1 play the analogue roles of e_1 and k_1 (considering $|k_1| > |k_2|$), but taking into account the perspective transform.

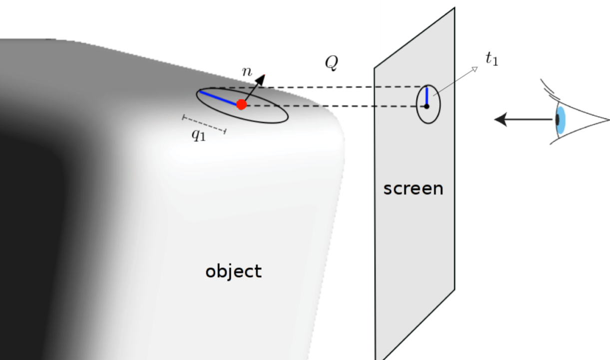

Figure 3.2: A unit circle of tangent vectors at the screen is mapped to a elongated ellipsis thru Q. The direction of maximum value is t_1 and the maximum value is q_1

3.3 Definition

Apparent ridges are the local maxima 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. As long as the normal turns away from the screen plane (approaching a contour), the projection affects the view-dependent curvature, shifting ordinary ridges towards the viewing vector (see figure 3.3). This happens because near contours the projection foreshortening makes q_1 value tend to infinity [13].

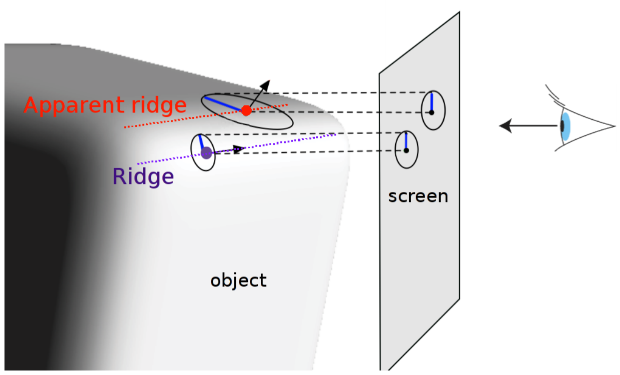

Figure 3.3: Apparent ridges and ordinary ridges

3.4 Contours

As mentioned before, when a point moves towards a contour, q_1 will tend to infinity due to projection (see figure 3.4). Although the view-dependent curvature is not defined at contours, the limit of q_1 is well-behaved and it achieves a maximum at infinity. This means that contours can be treated as a special case of apparent ridges.

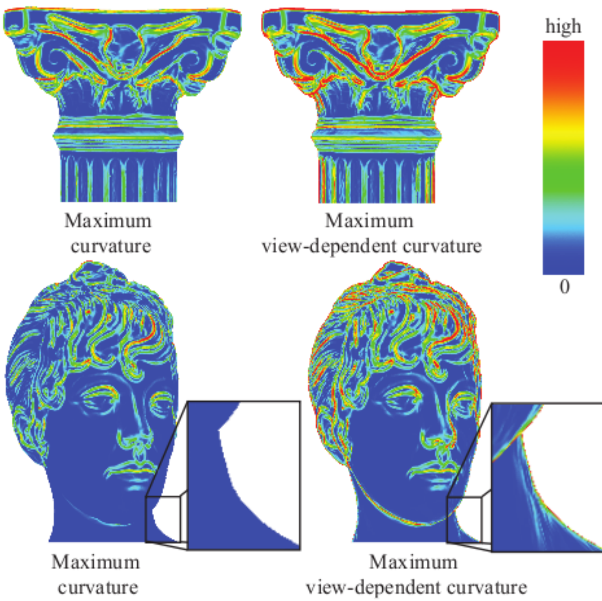

Figure 3.4: Comparison of the curvature and view-dependent curvature magnitudes (illustration extracted from [13]). View-dependent curvature approaches infinity near contours due to projection

Chapter 4

Our Method

In this chapter we present the main idea of our method which is adapted from the original one to take advantage of modern graphic cards. We explain the necessary modifications in the previous approach and some implementation details to achieve this goal.

4.1 Motivation

In the original work [12], Judd et al. presented an approach to find apparent ridges on triangle meshes, along with a CPU-based sample implementation. Since it was the first work about apparent ridges, performance was not a major goal. Instead, the authors were mainly concerned about showing that apparent ridges are good lines for expressive drawing of 3D models.

The main motivation of our method is to speed up the extraction of apparent ridges without compromising image quality, making them more competitive with other lines. For this, we use the GPU processing power.

4.2 Characterization

In the first place, Judd's approach is considered an *object-space* method because all the computations are performed over the 3D mesh. At the end of the process, the output is a 3D apparent ridge line that lies over the mesh[1]. After that, the apparent ridges are projected onto the screen (see the top diagram in figure 4.1). In contrast, our method is not entirely object-space

[1] Actually, the line is an approximation, made of straight line segments over the triangle faces of the mesh.

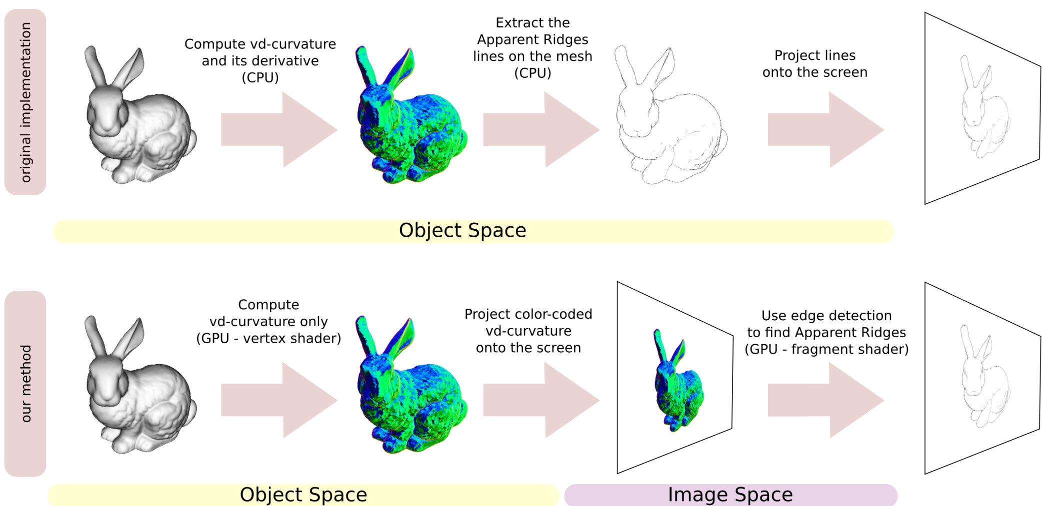

Figure 4.1: Methods work flow comparison

based: part of the process is performed in *image-space*, which is the 2D screen space. For this reason, our approach can be considered a *hybrid method*.

4.3 Why a Hybrid Method?

As shown in chapter 3, apparent ridges are the local maxima of the view-dependent curvature q_1 in the principal view-dependent maximum direction t_1.

In Judd's approach, the lines are extracted 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. The derivative computation is the method's bottleneck, since it is expensive and is repeated every time the viewpoint changes.

Our method works around this issue by using another way to extract apparent ridges without computing the derivative. Instead, we extract them in image-space.

This choice brings two important advantages. Since the extraction is done in image-space, the performance of this stage does not depend on the mesh size, providing an overall performance improvement. The other advantage is that the image-space stage is modular and it can be combined with other methods for estimating the view-dependent curvature. Particularly, it can be used to extract apparent ridges from other sources like images, point clouds and volumetric data.

4.4 Overview of Our Method

We present a method that runs major computations on the GPU. To achieve this, we split the rendering process into two stages. The view-dependent curvature data is estimated in an object-space stage and apparent ridges are extracted in an image-space stage, using edge detection (see the bottom diagram in figure 4.1).

This split will allow us to use vertex and pixel shaders to program each stage in the GPU, exploiting its processing power and parallelism. These changes provide significant speedups that will be discussed in chapter 5.

All the required 3D data, like the normal, the principal curvatures and principal directions are estimated using a popular technique proposed by Rusinkiewicz [19]. They are computed once as they do not change for a static model.

Now we are going to see each stage in detail.

4.5 Object-Space Stage

In this first stage, q_1 and t_1 are estimated at each vertex of the mesh. This is done using the same computations that are performed by Judd's method, except that it is implemented in a vertex shader. The required data (n, k_1, k_2, e_1 and e_2) are passed to the vertex shader program as vertex information like colors and texture coordinates.

After the computation, q_1 and t_1 are packed into the color output RGB channels of the vertex shader. We use one channel to pack q_1 and two channels to pack t_1, because it is a 2D screen vector (see figure 4.2).

As shown in chapter 3, 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 the channel interval that is $[0, 1]$. Before this truncation, q_1 must be scaled so that all the local maxima lie in $[0, 1]$, but also preserving precision. This means that this positive scaling factor cannot be too high or too low. This factor is controlled by the user. We empirically derived a function to guess an acceptable value, based on the feature size of the model. 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

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

For t_1, as it is a normalized 2-dimensional vector, its components t_{1x} and t_{1y} lie in the $[-1, 1]$ range. We use a simple affine transform to pack it in the

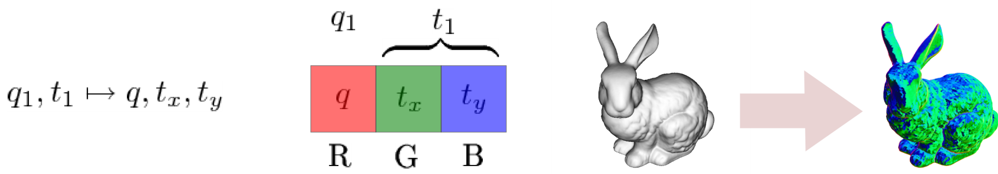

Figure 4.2: Color coding of q_1 and t_1

[0, 1] range:

$$t_x = \frac{1}{2} + \frac{1}{2}t_{1x}$$
$$t_y = \frac{1}{2} + \frac{1}{2}t_{1y}$$

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 (see figure 4.3). This will pass the information automatically to an off-screen buffer, and will be input to the fragment shader in the second stage. This concludes the first stage.

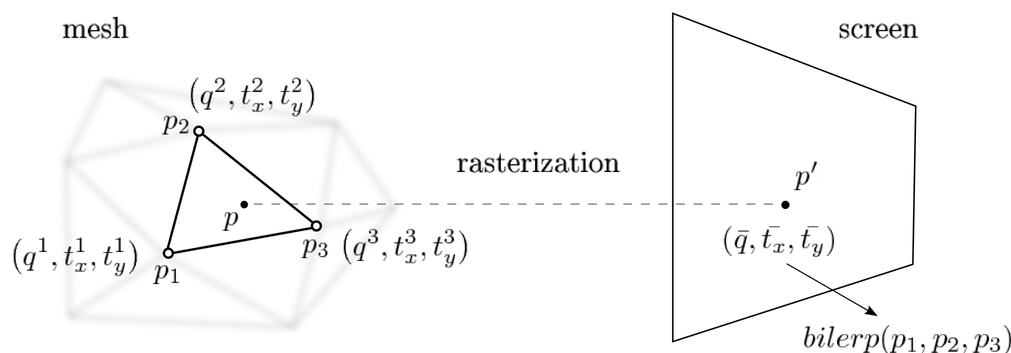

Figure 4.3: Rasterization of the packed values

4.6 Image-Space Stage

The main difference between our method and Judd's is that the apparent ridges are not extracted computing the zero-crossings of $D_{t_1}q_1$. Instead, we use edge detection to find the maxima of q_1 in the t_1 direction. Since t_1 is a screen vector by definition, our choice of finding the maxima in image-space is reasonable.

In the previous stage, q_1 and t_1 were computed and rasterized to an off-screen buffer. To find the apparent ridges, we perform a simple edge-detection

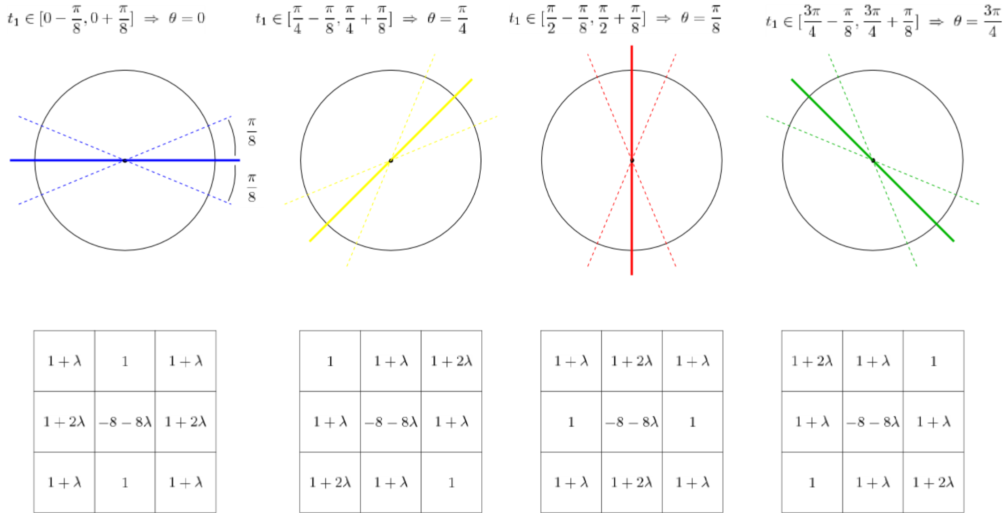

Figure 4.4: Laplacian-like adaptive filter. Filter setup is chosen according to the maximum view-dependent principal direction

in this buffer. We use a standard fragment shader technique for doing image processing on the GPU [21].

The edge detection is done with a 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. First, the t_1 direction is quantized into one of four main directions

$$\theta \in \left\{0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}\right\}$$

and then the filter setup is chosen based on one of these four directions (see figure 4.4). The filter is weighted by a λ parameter that controls how sensitive the filter is to the θ direction. Note that, 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. We use this intensity 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 could be avoided by setting the pixel value to black when intensity is higher than a minimum threshold (see figure 4.5). 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 4.5: Results with (left) and without (right) the fading effect

Chapter 5

Results

In this chapter we present some of our method's results, observe the effects of varying our method's parameters (τ and λ) and compare our results with Judd's in image quality and performance, side-by-side.

Figure 5.1: Our results (bottom) on some models along with shaded views (top) for comparison

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

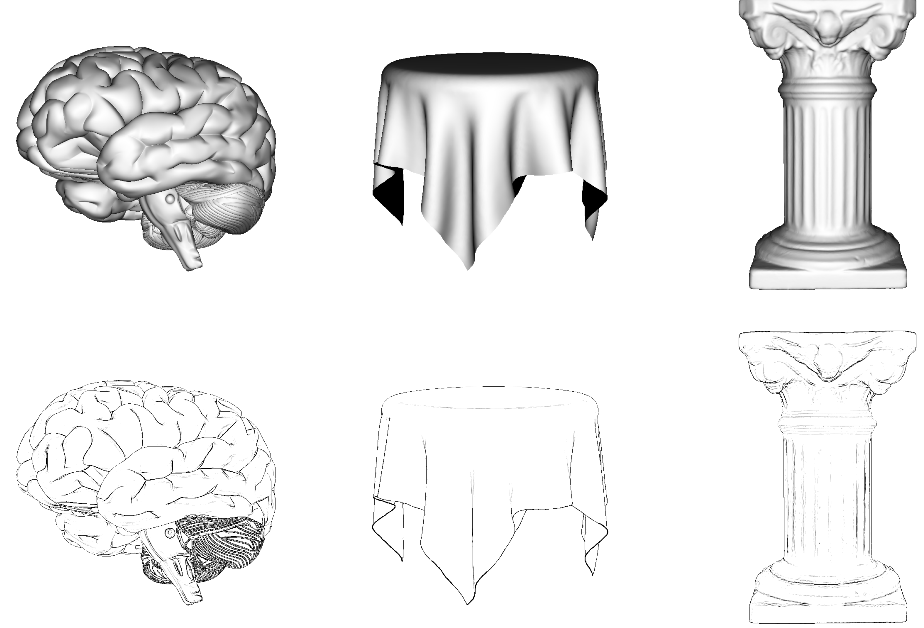

Figure 5.2: Our results (bottom) on some models along with shaded views (top) for comparison

respective shaded view (see figures 5.1 and 5.2).

A great visual property of apparent ridges is that they can depict relatively complex models with very few lines in a clean way. Moreover, if the default setup of our method is not pleasing enough, one can manipulate the parameters τ and λ to achieve maximum quality. These parameters variation will be treated in the next section.

5.1 Parameter Variation

As shown in chapter 4, the τ parameter is used to scale the view-dependent curvature q_1. In figure 5.3 we present results of our method on the top row and the scaled view-dependent curvature $q = 2^{\tau} q_1$ at the bottom row. Each column has the a fixed τ value, that linearly assumes values from -9.9 to -2.4, from left to right.

As can be seen in figure 5.3, 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).

One interesting result is that the Laplacian filter detects most of the

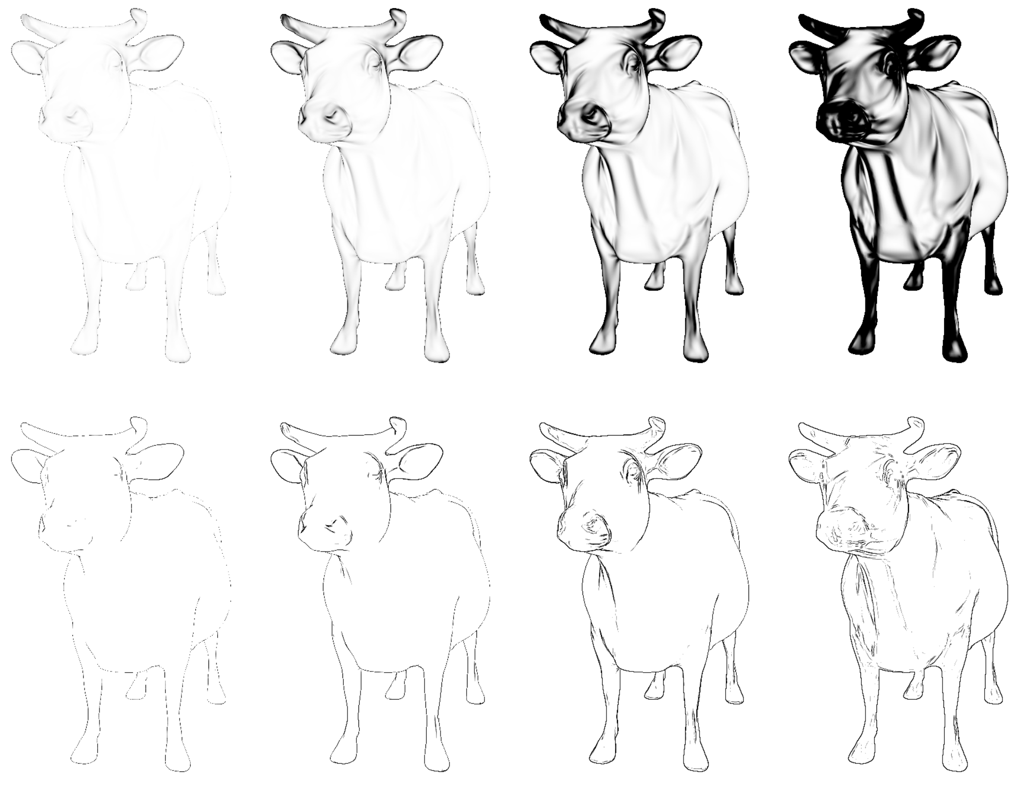

Figure 5.3: Top row: q in grayscale. Bottom row: apparent ridges. τ parameter assuming values -9.9, -7.4, -4.9 and -2.4, from left to right.

apparent ridges. The Laplacian filter is chosen by setting $\lambda = 0$. As the maxima estimation is achieved at pixel-level, some features may be harder to catch when the projected faces of the mesh are much larger than the pixel. Large faces produce large areas of interpolated curvature. To overcome this problem, higher values of λ can be set. See the top-left tablecloth in figure 5.4, where the main top ridge is not captured by the Laplacian filter. Higher values of λ capture this feature and produce sharper apparent ridges.

Believing that apparent ridges produce quite pleasant images, we will not compare our results with other lines like suggestive contours or ridges and valleys. Instead, we will compare our results with Judd's method results, both in image quality and performance.

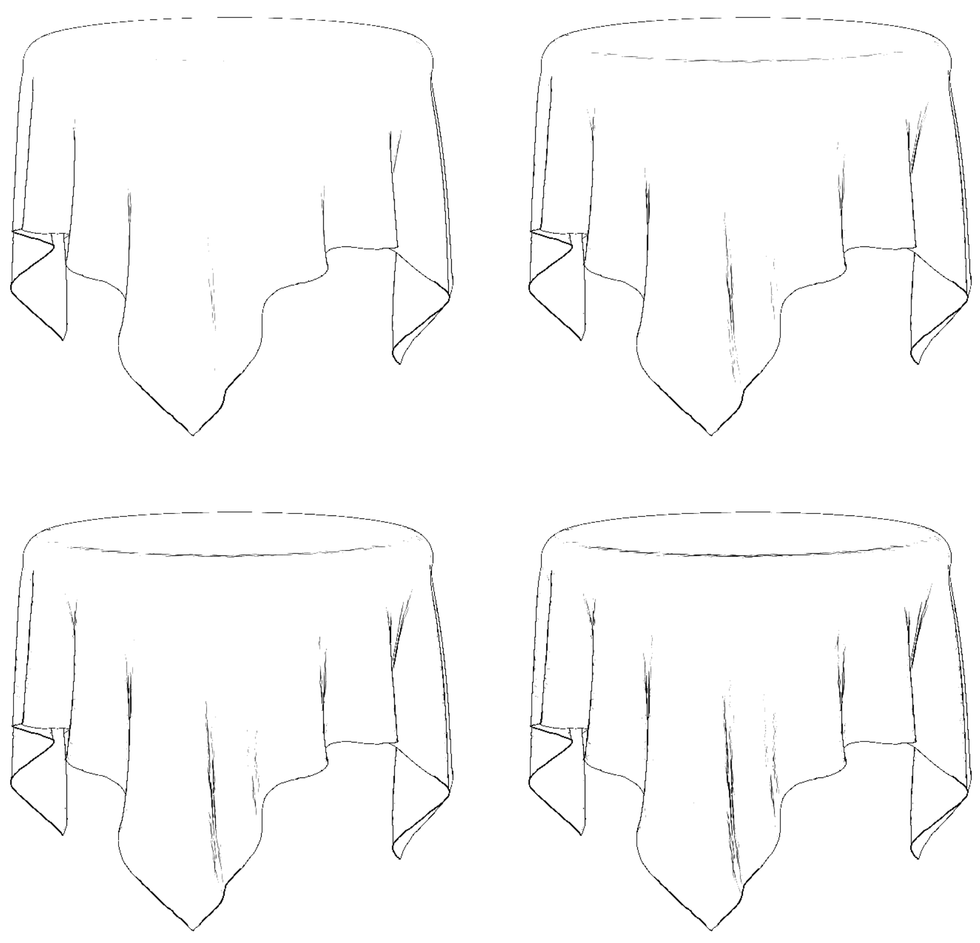

Figure 5.4: Results of the tablecloth with the λ parameter assuming values 0.0, 1.5, 3.0 and 4.5, from left to right, top to bottom

5.2 Image Comparison

We shall compare some results of the Judd's method and ours side-by-side. The methodology used with the results is the following: given a model, an appropriate threshold is set for the Judd's method. Then, we set our method's parameters (τ and λ) in a way the resulting image is visually as close as possible to the Judd's result. The parameter setting is done manually for both methods. We have tested several models and the results can be seen from figures 5.5 to 5.10.

As seen in the results, our method produces images that are quite similar

to the ones produced by Judd's method; apparent ridge lines appear generally in the same place. In some cases, it is very hard to notice the slight differences with bare eyes (especially in a printed version); image closeups should reveal pixel-level differences. The image differences occur 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 does an edge-detection on the rasterized view-dependent curvature.

While the images produced by both methods are equally pleasant, we believe that ours are a little more 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 case of the roudedcube model (figure 5.9). In general, our method works fine for larger models and these artifacts can be eliminated by mesh subdivision.

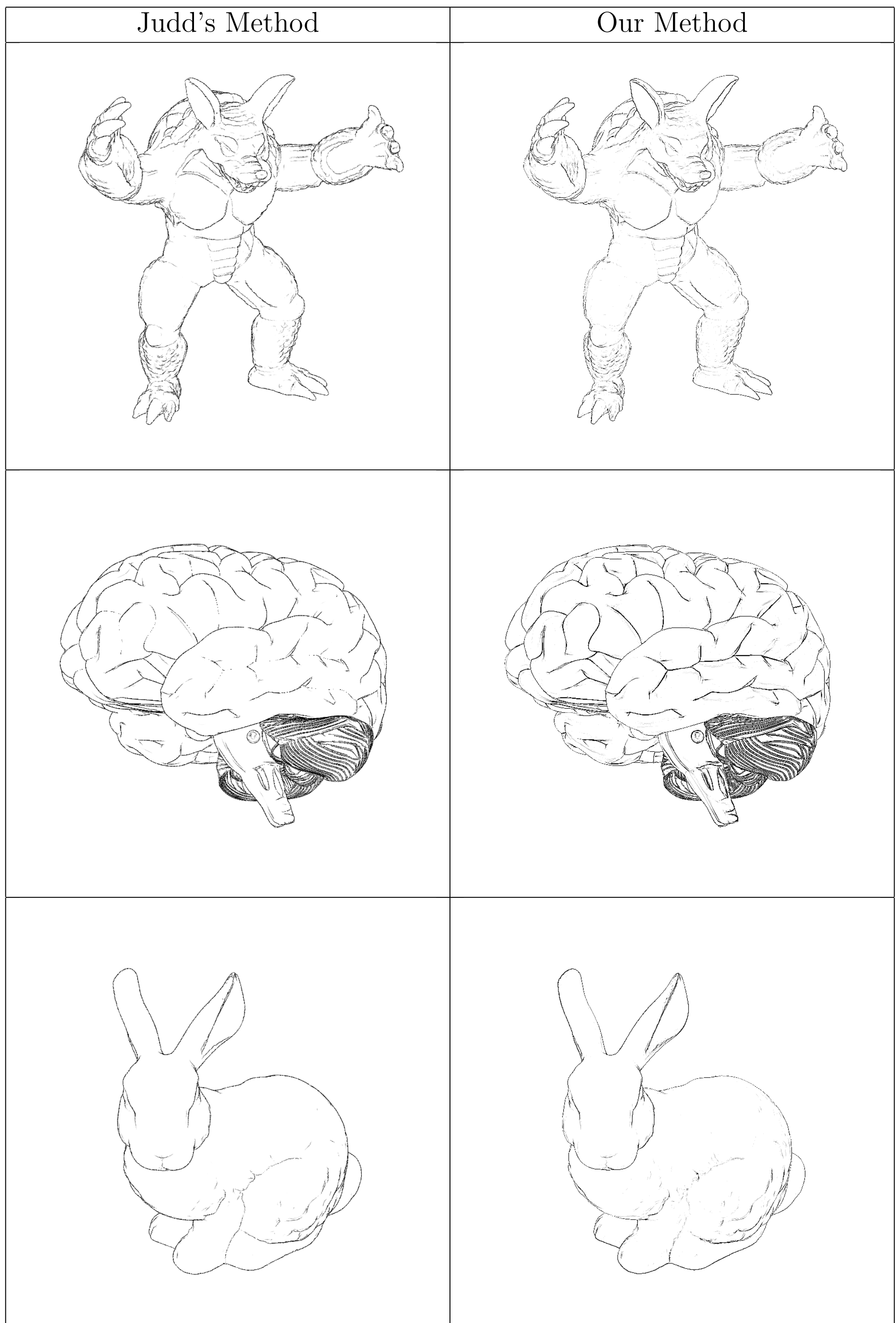

Figure 5.5: Comparison for the armadillo, brain and bunny models

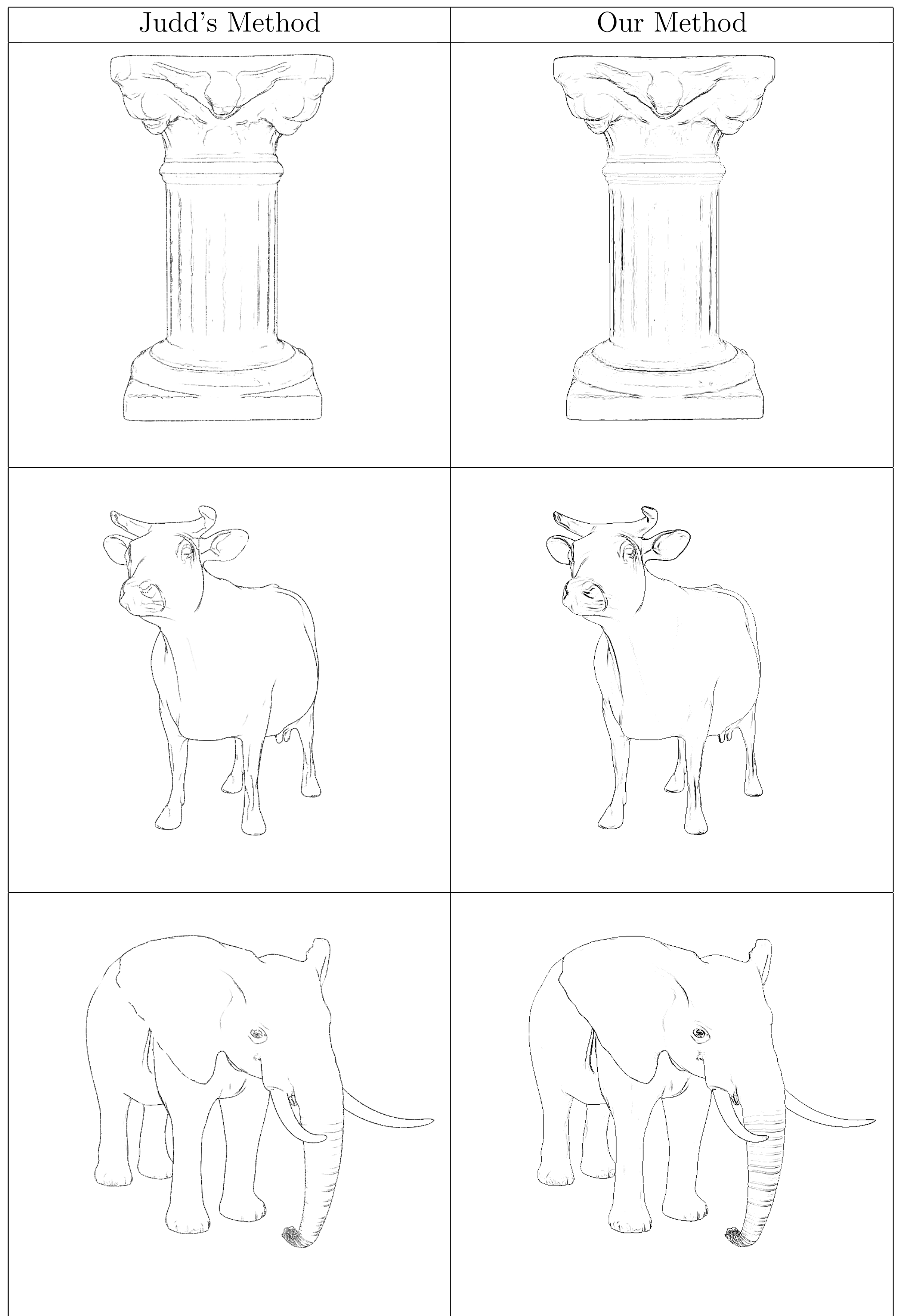

Figure 5.6: Comparison for the column, cow and elephant models

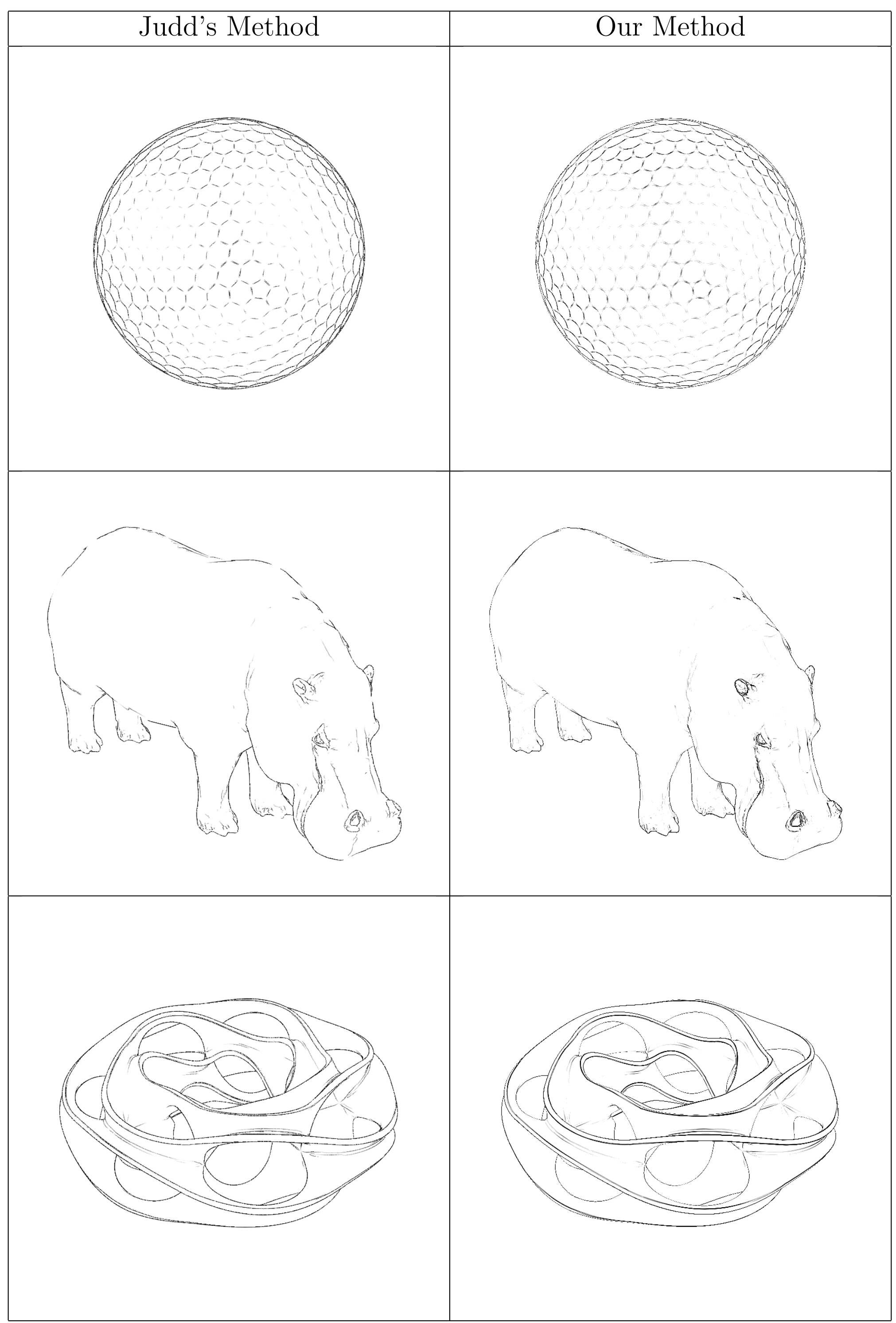

Figure 5.7: Comparison for the golfball, hippo and heptoroid models

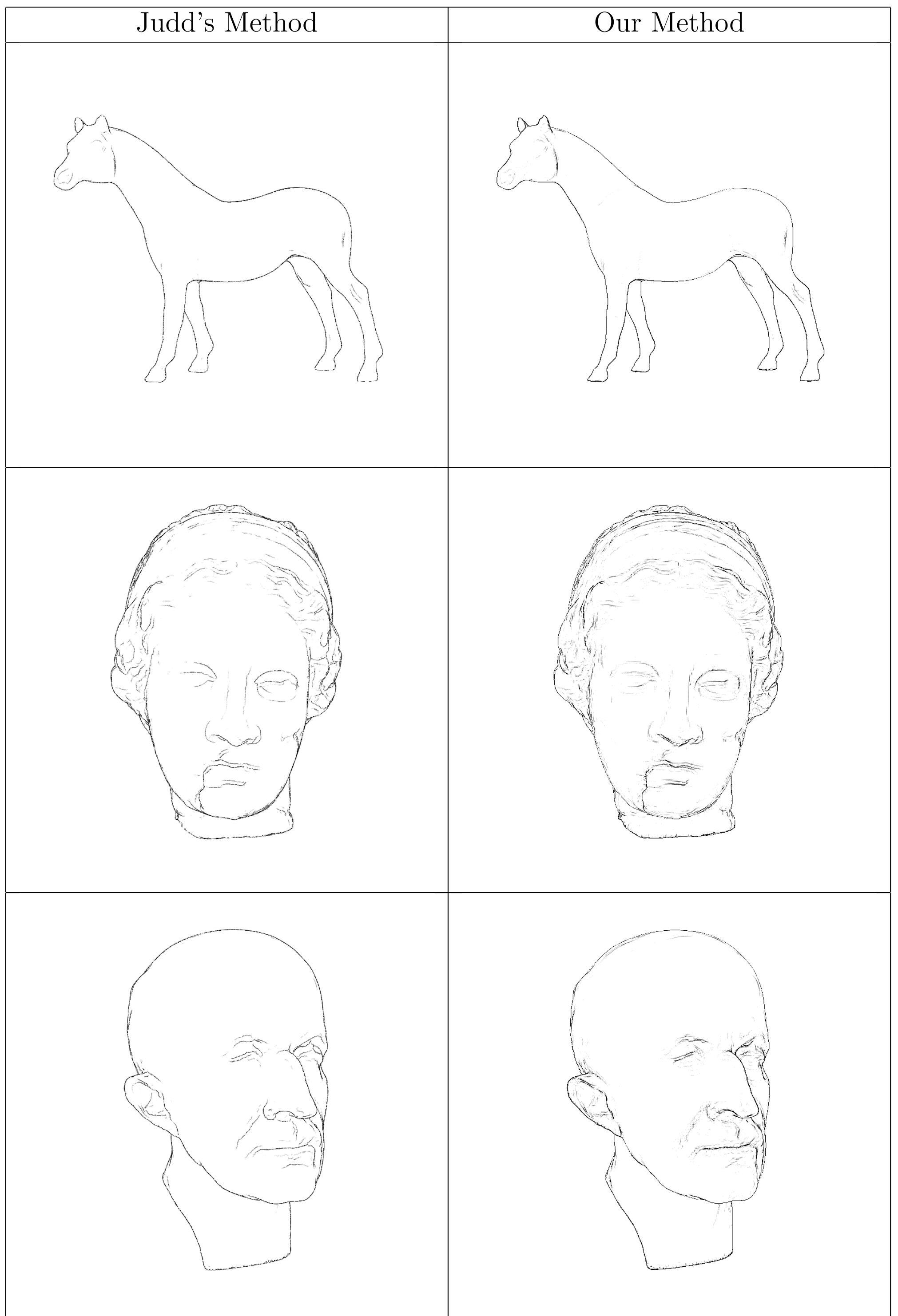

Figure 5.8: Comparison for the horse, igea and planck models

Figure 5.9: Comparison for the lucy model

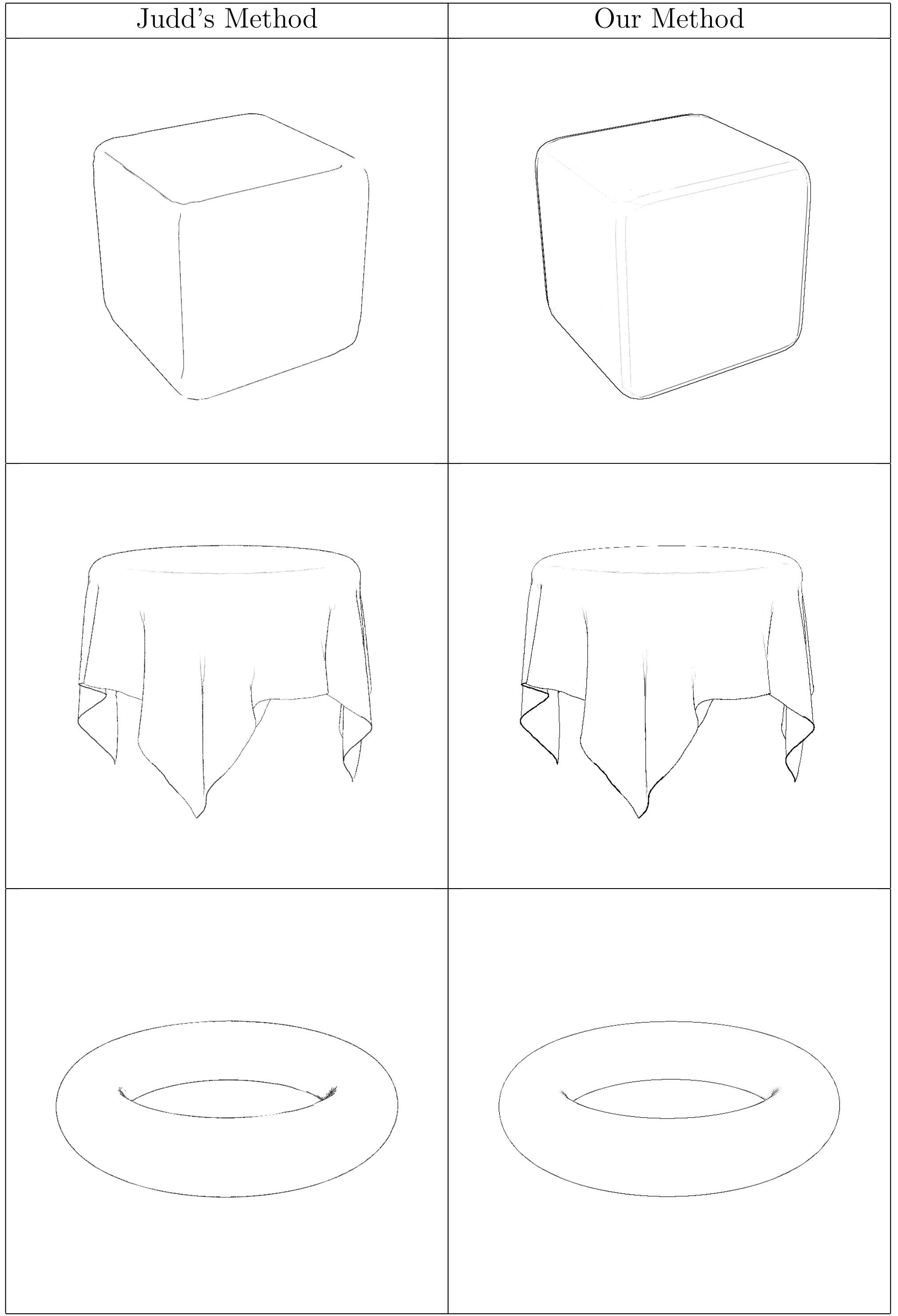

Figure 5.10: Comparison for the roundedcube, tablecloth and torus models

5.3 Performance Comparison

A drawback of the Judd's apparent ridges compared to other lines is that they tend to be slower to compute as they rely on the view-dependent curvature and its derivative information. Our method improves the performance of apparent ridges, leading them to be competitive in speed with other feature lines, with the all the advantages of apparent ridges.

The code was implemented in C++ language for the main application using the *trimesh2* library [20, 19]. Shaders were written in GLSL [21] and the codes can be found in Appendix A. The original apparent ridges code was incorporated and adapted to run inside our program for side-by-side and performance comparisons. The 3D mesh models were collected from [22, 23].

We tested our method and Judd's on two computers: an ordinary laptop with a regular graphics card and a high-end workstation with a powerful graphics card. The laptop has an AMD Turion X2 processor with a 512MB NVidia GeForce 8200M card while the workstation has a dual AMD Opteron processor with a 1.5GB NVidia Quadro FX 5600 card. All the cards have GPU support for vertex and fragment shaders.

The results in tables 5.1 and 5.2 show that our method provides significant speedups compared to Judd's, except from the particular cases of the two smaller models on the laptop. However, the frame rate for these cases is still high (over 80 FPS).

In the laptop results, it is possible to see that our method performs better for larger models. Indeed, it can be seen that 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.

A very interesting result is that, since Judd's method is CPU-based, it is expected a speedup from its results when comparing the laptop to the workstation results. Although it 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.

In general, the results are very encouraging and show that GPU-based solutions for line extraction may be a good choice when performance matters.

Model	# Vertices	Judd's (FPS)	Ours (FPS)	Speedup
roundedcube	1538	202	90	0.5
torus	4800	148	84	0.6
tablecloth	22653	32	64	2.0
hippo	23105	41	66	1.6
cow	46433	24	58	2.4
horse	48484	22	52	2.4
maxplanck	49132	20	46	2.3
bunny	72027	15	42	2.8
elephant	78792	14	47	3.4
golfball	122882	9	30	3.3
igea	134345	8	29	3.6
armadillo	172974	5	18	3.6
column	262653	3	9	3.0
lucy	262909	4	13	3.3
heptoroid	286678	4	21	5.3
brain	294012	4	20	5.0

Table 5.1: Results for the laptop with a NVidia GeForce 8200M

Model	# Vertices	Judd's (FPS)	Ours (FPS)	Speedup
roundedcube	1538	600	1100	1.8
torus	4800	270	870	3.2
tablecloth	22653	32	160	5.0
hippo	23105	42	160	3.8
cow	46433	23	198	8.6
horse	48484	26	147	5.6
maxplanck	49132	22	90	4.1
bunny	72027	16	102	6.4
elephant	78792	15	113	7.5
golfball	122882	9	52	5.8
igea	134345	8	58	7.3
armadillo	172974	5	30	6.0
column	262653	3	15	5.0
lucy	262909	4	26	6.5
heptoroid	286678	5	19	3.8
brain	294012	4	34	8.5

Table 5.2: Results for the workstation with a NVidia Quadro FX 5600

Chapter 6

Conclusions and Future Work

Since their introduction, apparent ridges have shown to be perceptually pleasant and also competitive with lines like suggestive contours by depicting the same features in a clear and smooth way, but also depicting convex features. However, as stated in Judd's work [13], apparent ridges are slower to compute since they depend on high-order derivatives of the view-dependent curvature, which changes with the viewpoint.

We have presented a new method for computing apparent ridges that is faster than the original, replacing the computation of the curvature derivative with a simple edge detection, providing similar image quality. With this new performance rates, apparent ridges become even more competitive.

As future work, we intend to experiment with some techniques to improve image quality, like better filters and the use of a Phong-like shading of the view-dependent curvature. This should improve the quality of image closeups and small models.

Also, the image-space stage of our method could be used as part of a pipeline to extract apparent ridges from volume data and implicit models. To extract apparent ridges, one should just concentrate on the extraction of the view-dependent curvature from the isosurfaces and rasterize it to an off-screen buffer.

Our method could also be adapted to extract other lines, like suggestive contours. Properties like the radial curvature and its derivative could be rasterized to an off-screen buffer where appropriate screen operations can be applied to find them.

The main contribution of Judd's work is certainly the definition of the view-dependent curvature. We would like to explore this property in different contexts like shading, mesh evaluation and modeling.

Bibliography

[1] W. A. Fetter, *Computer Graphics, Aircraft Applications* Document No. D3-424-I, Boeing Airplane Company, Wichita Division, 1961

[2] *Toward a Machine with Interactive Skills, Understanding Computers: Computer Images*, Time-Life Books, 1986

[3] *POV-Ray Hall of Fame* Web Page, http://hof.povray.org, 2009

[4] M. C. Sousa, *Overview of NPR for Computerized Illustration*, in Illustrative Visualization for Medicine and Science, ACM SIGGRAPH 2006 Courses, Boston, Massachusetts, 2006

[5] M. Pharr and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, Morgan Kaufmann, July 2004

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

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

[8] K. Hulsey, Technical Illustration Web Page, http://www.khulsey.com, 2009

[9] *Boston College's Gallery of John Flaxman's Odyssey (1835)*, http://www.bc.edu/bc_org/avp/cas/ashp/flaxman_odyssey.html, 2009

[10] W. Koch, *Baustilkunde.* Mosaik-Verlag GmbH, Mnchen, 1991

[11] 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

[12] T. Judd, F. Durand, and E. Adelson, *Apparent Ridges for Line Drawing*, ACM Transactions on Graphics, vol. 26, no. 3, article 19, July 2007

[13] T. Judd, *Apparent Ridges for Line Drawing.* Master's Thesis, Computer Science, MIT, Jan 2007

[14] A. Hertzmann, *Introduction to 3D Non-Photorealistic Rendering*, in Non-Photorealistic Rendering (SIGGRAPH'99 Course Notes), 1999

[15] K. Na, M. Jung, J. Lee, and C. G. Song, *Redeeming Valleys and Ridges for Line-Drawing*, in Advances in Multimedia Information Processing, Lecture Notes in Computer Science, vol. 3767, pp. 327–338, 2005

[16] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, *Suggestive Contours for Conveying Shape*, ACM Transactions on Graphics, vol. 22, no. 3, pp. 848–855, 2003

[17] J. J. Koenderink, A. J. van Doorn, C. Christou, J. S. Lappin, *Shape Constancy in Pictorial Relief.* Perception vol. 25 no. 2, pp. 155–164, 1996

[18] M. P. do Carmo, *Differential Geometry of Curves and Surfaces.* Prentice-Hall, 1976

[19] S. Rusinkiewicz, *Estimating Curvatures and Their Derivatives on Triangle Meshes.* Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium, pp. 486–493, September 2004

[20] *The trimesh2 Library*, http://www.cs.princeton.edu/gfx/proj/trimesh2, 2009

[21] R. Wright, B. Lipchak and N. Haemel, *OpenGL Superbible: Comprehensive Tutorial and Reference*, Addison-Wesley, 4th Edition

[22] *Suggestive Contour Gallery*, http://www.cs.princeton.edu/gfx/proj/sugcon/models, 2009

[23] *Apparent Ridges for Line Drawings*, http://people.csail.mit.edu/tjudd/apparentridges.html, 2009

Appendix A

Shader codes

The GLSL codes for the vertex and fragment shaders (see listings A.1 and A.2).

Listing A.1: Vertex shader in GLSL

```glsl

void main(void)
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    vec3 N = normalize(gl_NormalMatrix * gl_Normal);
    vec3 e1 = gl_NormalMatrix * gl_SecondaryColor.rgb;
    vec3 e2 = gl_NormalMatrix * gl_MultiTexCoord0.rgb;
    float k1 = gl_Color.r, k2 = gl_Color.g;
    vec4 V = gl_ModelViewMatrix * gl_Vertex;
    vec3 viewdir = normalize(-V.xyz);
    float ndotv = dot(viewdir, N);
    float u = dot(viewdir, e1), v = dot(viewdir, e2);
    float u2 = u*u, v2 = v*v, uv = u*v;
    float csc2theta = 1.0 / (u2 + v2);
    u2 *= csc2theta;
    uv *= csc2theta;
    v2 *= csc2theta;
    float sectheta_minus1 = 1.0 / abs(ndotv) - 1.0;
    float Q11 = k1 * (1.0 + sectheta_minus1 * u2);
    float Q12 = k1 * (      sectheta_minus1 * uv);
    float Q21 = k2 * (      sectheta_minus1 * uv);
    float Q22 = k2 * (1.0 + sectheta_minus1 * v2);
    float QTQ1  = Q11*Q11 + Q21*Q21;
    float QTQ12 = Q11*Q12 + Q21*Q22;
    float QTQ2  = Q12*Q12 + Q22*Q22;
    float a = QTQ12*QTQ12, b = QTQ2-QTQ1;
    float q1 = 0.5 * (QTQ1 + QTQ2);
    if (q1 > 0.0)
        q1 += sqrt(a + 0.25*b*b);
    else
        q1 -= sqrt(a + 0.25*b*b);
    vec2 s1 = normalize(vec2(QTQ2-q1, -QTQ12));
    vec3 w1 = s1[0]*e1 + s1[1]*e2;
    vec2 t1;

    // q1 and t1 packing
    float q = pow(2.0, tau)*q1;
    if (w1.z >= 0.0)
        t1 = normalize(vec2(-w1.x, -w1.y));
    else
        t1 = normalize(vec2(w1.x, w1.y));

    gl_FrontColor.rgb = vec3(q, 0.5+0.5*t1.x, 0.5+0.5*t1.y);
}
```

Listing A.2: Fragment shader in GLSL

```glsl
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
uniform float lambda;

void main(void)
{
    bool border = false;
    vec2 z = vec2(0.0, 0.0);
    float v = 0.0, diff;
    float a = 0.70710678118655; // sqrt(2)/2
    float b = 0.92387953251080; // cos(pi/8)
    float l1 = 1.0+lambda, l2 = 1.0+2.0*lambda;
    vec4 s[9];

    for (int i = 0; i < 9; i++) {
        s[i] = texture2D(sampler0, gl_TexCoord[0].st + tc_offset[i]);
    }

    if (s[4].rgb == vec3(1.0, 0.0, 0.0)) border = true;

    vec2 t1 = vec2(2.0*(s[4].g-0.5), 2.0*(s[4].b-0.5));

    if (t1 != z) {
        vec2 d1 = vec2(1.0, 0.0),
             d2 = vec2(a, a),
             d3 = vec2(0.0, 1.0),
             d4 = vec2(-a, a);
        int d = -1;
        if (abs(dot(t1, d1)) > b) d = 1;
        else if (abs(dot(t1, d2)) > b) d = 2;
        else if (abs(dot(t1, d3)) > b) d = 3;
        else d = 4; // if (abs(dot(t1, d4)) > b)

        v = (8.0+8.0*lambda)*s[4].r;
        if (d == 1) {
            v -= ( l1 * s[0].r + s[1].r + l1 * s[2].r +
                   l2 * s[3].r          + l2 * s[5].r +
                   l1 * s[6].r + s[7].r + l1 * s[8].r          );
        }
        else if (d == 2) {
            v -= ( s[0].r + l1 * s[1].r + l2 * s[2].r +
                            l1 * s[3].r               + l1 * s[5].r +
                            l2 * s[6].r + l1 * s[7].r +      s[8].r    );
        }
        else if (d == 3) {
            v -= ( l1 * s[0].r + l2 * s[1].r + l1 * s[2].r +
                        s[3].r               +      s[5].r +
                   l1 * s[6].r + l2 * s[7].r + l1 * s[8].r          );
        }
        else {
            v -= ( l2 * s[0].r + l1 * s[1].r +      s[2].r +
                   l1 * s[3].r               + l1 * s[5].r +
                        s[6].r + l1 * s[7].r + l2 * s[8].r          );
        }
    }
    // color invertion
    if (v > 0.2) gl_FragColor = vec4(1.0-v);
    else gl_FragColor = vec4(1.0);
}
```