Contours and Suggestive Contours for 3D Mesh

Cecilia Berti
Télécom Paris
cecilia.berti@telecom-paris.fr

***Abstract*—This report presents an implementation of suggestive contours, as proposed by De Carlo et al., to enhance the visualization of 3D meshes. Starting from one of the lab exercises, the project employs subdivision and smoothing techniques to improve mesh quality. The computation of silhouette and suggestive contours is performed using the object-space algorithm, with principal curvatures and directions calculated via quadric fitting. Principal curvatures and gradient computations are handled on the CPU, while the rest of the algorithm is executed on the GPU using shaders. The usage of GPU also allows real-time rendering. The report discusses the implementation details, parameter tweaking for visualization, and potential improvements.**

I. Introduction

In the field of computer graphics, effective visualization of 3D meshes is crucial for various applications, including animation, gaming, and scientific visualization. This project aims to enhance mesh visualization by implementing suggestive contours, a technique that highlights important features of a 3D model. Building upon the subdvision work seen in class, this report details the steps taken to achieve better mesh visualization through subdivision and smoothing, followed by the computation of silhouette and suggestive contours. The implementation follows the algorithm proposed by DeCarlo et al. [1] , exploiting Shaders, instead of heavy CPU computations, to optimize the performance.

II. Background and Motivation

A. Overview

Contours are the set of points on a surface where the normal $\mathbf{n}$ is perpendicular to the viewing direction $\mathbf{v}$, formally defined as:

$$\mathbf{n} \cdot \mathbf{v} = 0. \tag{1}$$

Contours indicate the boundary between visible and occluded regions but may be insufficient to convey finer shape details. Suggestive contours extend this idea by marking locations where contours would appear under slight viewpoint changes. Mathematically, they correspond to points where the radial curvature κ_r is zero,

$$\kappa_r = 0, \tag{2}$$

and where the directional derivative of κ_r along $\mathbf{w}$, the projection of $\mathbf{v}$ onto the tangent plane, is positive:

$$D_{\mathbf{w}}\kappa_r > 0. \tag{3}$$

These conditions ensure suggestive contours anticipate true contours and seamlessly integrate with them, enhancing the perception of shape in line drawings [1].

B. Related work

Contour-based techniques have been widely explored for 3D shape visualization. Early works, such as those by DeCarlo et al. [1], introduced *suggestive contours*, which generalize occluding contours by capturing surface regions where contours are likely to appear in nearby views. This approach has since been extended to include *apparent ridges* [2], which consider view-dependent curvature extrema, and *ridges and valleys* [3], which detect intrinsic geometric features.

Despite their effectiveness, traditional geometric approaches often require fine-tuned thresholds and can struggle with artifacts on low-resolution meshes. Tang et al. [4] propose an optimization-based framework that improves the stability of suggestive contours by refining the underlying mesh geometry using the *CEPS algorithm*. By remeshing the surface to conformal equivalence, their method reduces noise and enhances feature preservation in contour extraction.

Recent advances in deep learning have enabled data-driven contour extraction methods that overcome some of the limitations of purely geometric techniques. Liu et al. [5] introduce *Neural Contours*, a hybrid approach that combines explicit geometric constraints with a neural network trained on human-annotated contour datasets. Their method leverages both *view-based image translation networks* and *differentiable geometric modules*, learning to approximate human-drawn lines while automatically tuning contour thresholds.

Unlike traditional methods, which require handcrafted threshold tuning for each model, Neural Contours use a ranking-based optimization to select the best combination of contour types. This approach demonstrates significantly improved perceptual accuracy, as evaluated by both objective metrics and crowdsourced human judgments.

C. Objectives

Our objective is to implement a simplified approach for generating suggestive contours, leveraging both CPU and GPU computation to improve efficiency. We aim to achieve results comparable to those presented in [1], while optimizing performance and enabling real time drawing.

III. Proposed approach

A. Preprocessing

Since we worked with meshes that had a relatively limited number of vertices and faces, a good practice to achieve better contouring was to subdivide the mesh. We utilized the **subdivideLoop()** function, which was already implemented in one of the previous labs. The meshes we worked on included

the provided Suzanne monkey and a simplified Stanford bunny with 500 faces. The Stanford bunny, in particular, required subdivision before the application of the smoothing algorithm to enhance its detail and smoothness.

Moreover, we applied the Taubin Smoothing algorithm, which is a well-known method for mesh smoothing. This algorithm involves two consecutive Gaussian smoothing steps. The first step applies a Gaussian smoothing with a positive scale factor λ to all the vertices of the shape. The second step applies another Gaussian smoothing, but with a negative scale factor μ, which is greater in magnitude than the first scale factor ($0 < \lambda < -\mu$). To produce a significant smoothing effect, these two steps are repeated multiple times, alternating the positive and negative scale factors [6].

Both functions are triggered via keyboard input. Pressing the key `L` executes a single iteration of the `subdivideLoop` function, while pressing the key `R` applies the smoothing algorithm to the mesh. The smoothing parameters are set to $\lambda = 0.33$ and $\mu = -0.34$, and the process is repeated for 10 iterations.

For the Stanford bunny mesh presented in this paper, we applied two smoothing passes followed by one subdivision iteration to achieve the desired level of smoothness and detail.

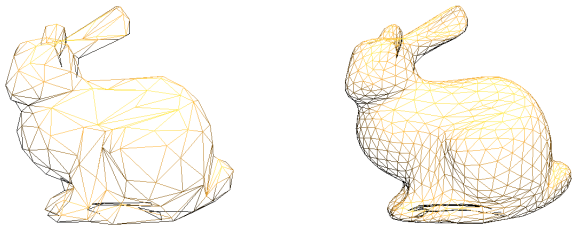

Fig. 1: Mesh before and after the pre-processing.

B. Object-Space Algorithm

The algorithm proceeds first by finding the solution to $K_r = 0$ over the entire mesh. The suggestive contours are computed by trimming this solution using $D_{\mathbf{w}}\kappa_r > 0$.

1) Principal Curvatures and Directions: The computation of the principal curvatures and directions is a fundamental passage of the algorithm. In the CPU we compute the principal curvatures k_1 and k_2 and the principal directions e_1 and e_2. The two principal curvatures at a point on a surface measure how much the surface bends in different directions. The directions of maximum and minimum (signed) bending are called principal directions and are always orthogonal [7].

There are several methods to find curvatures and directions, in our implementation, we leverage the libigl library with the function **principal_curvature** which uses *quadric fitting*; For each vertex, we consider its one-ring neighbors or a local patch, and we fit a quadratic surface to this neighborhood using least squares. Then, we compute the Weingarten matrix from the fitted surface: the eigenvalues of the Weingarten matrix give the principal curvatures and the eigenvectors give the principal directions.

2) Gradient Computation: The gradient of the radial curvature is essential for filtering suggestive contours, ensuring that only perceptually meaningful lines are drawn. The gradient of radial curvature is computed by first calculating the radial curvature at each vertex. For a given vertex, the radial curvature κ_r is derived from the principal curvatures κ_1 and κ_2 and the principal directions e_1 and e_2. The principal directions are unit orthogonal vectors, and any unit tangent vector e_θ can be expressed as a linear combination of e_1 and e_2:

$$e_\theta = e_1 \cos\theta + e_2 \sin\theta$$

The curvature κ_θ in the direction of e_θ is given by:

$$\kappa_\theta = \kappa_1 \cos^2\theta + \kappa_2 \sin^2\theta$$

which can be rewritten as:

$$\kappa_\theta = \kappa_1 (e_\theta \cdot e_1)^2 + \kappa_2 (e_\theta \cdot e_2)^2$$

To compute the radial curvature κ_r at a vertex, we first calculate the view vector v from the camera to the vertex. This vector is projected onto the tangent plane by subtracting its component in the direction of the vertex normal n:

$$v_{\text{proj}} = v - (n \cdot v)n$$

The projected vector v_{proj} is then normalized to obtain the unit vector w:

$$w = \frac{v_{\text{proj}}}{\|v_{\text{proj}}\|}$$

Finally, the radial curvature κ_r is computed as:

$$\kappa_r = \kappa_1 (w \cdot e_1)^2 + \kappa_2 (w \cdot e_2)^2$$

This radial curvature is used to compute the gradient across the mesh, which is then averaged at each vertex to produce a smooth gradient field.

3) GPU Shader Implementation: Once we compute the principal curvatures and directions, and the gradient of the radial curvature, all these variables are sent to the shaders. In the shaders we check condition (1) for the contours, then we compute the radial curvature per vertex and check condition (2), and finally we normalize the gradient of the curvature and check for condition (3). The fragments that satisfy the condition are drawn in black.

IV. Results and Parameter Tweaking

A. Normal Contours

As mentioned above, for contours we check condition (1), Since the dot product $n \cdot v$ rarely equals zero exactly, a small threshold ensures stable contour detection. Below we report the results obtained for some of the meshes used. The silhouette threshold for both figures was set at *0,2*. Since the view vector is dynamically computed, it is possible to observe how the contours are redefined when the mesh is animated (toggling `T`).

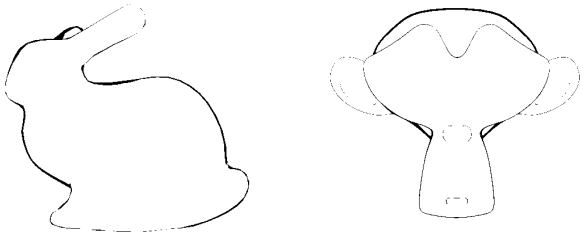

Fig. 2: Silhouettes for the meshes tested.

B. Suggestive contours

Also for suggestive contours we tweak the parameters a bit, we consider a threshold both for condition (2) and condition (3). In particular, to avoid spurious zero crossings, we apply a small positive threshold on (3), as suggested in [1]:

$$\frac{D_{\mathbf{w}}\kappa_r}{\|\mathbf{w}\|} > t_d. \tag{4}$$

where t_d is chosen arbitrarly. Below we report the results for contours and suggestive contours, first only for condition (2), this yields loops (Fig. 3), which are then subjected to the other condition (Fig. 4). Comparing Fig. 3 and Fig. 4, we see the importance of applying the derivative.

Fig. 3: Solution to (2)

Fig. 4: Solution to (2) and (3)

C. Post-processing

While the initial rendering pass provides a reasonable approximation of suggestive contours and silhouettes, the results exhibit noticeable noise and inconsistencies. Many fine details are lost, and some edges are either too fragmented or insufficiently emphasized. To address these shortcomings, we implemented a post-processing pass that enhances the contours by leveraging image-space edge detection techniques.

In this post-processing stage, the rendered image from the first pass is processed using a second fragment shader (*postProcessingFragment.glsl*) that detects and refines edges based on depth discontinuities and color variations. The key objectives introduced by this approach include:

- **Depth-based edge detection:** By analyzing the depth buffer, the shader identifies discontinuities that correspond to object boundaries, effectively improving silhouette definition.
- **Sobel filter on color gradients:** A Sobel edge detection operator is applied to the color buffer to highlight intensity changes, capturing edges that may have been missed by the depth-based approach alone.
- **Adaptive edge refinement:** By combining depth and color-based edge information, the post-processing shader smooths out noise while preserving meaningful contours, reducing the presence of small fragmented lines.

The post-processing pass ultimately improves the visual clarity of the contours while mitigating some of the limitations inherent in the curvature-based approach used in the first pass, the final result is shown in Fig. 5. While post-processing enhances contour clarity, some noise persists, suggesting the need for additional smoothing techniques or edge-thinning strategies

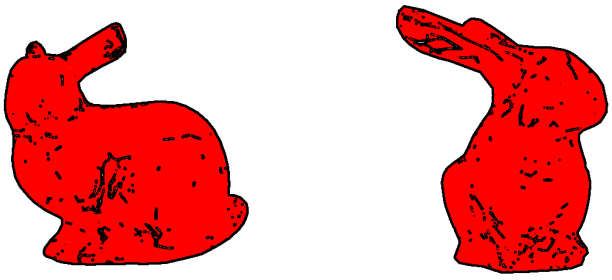

Fig. 5: Final results obtained after the post-processing pass.

V. Problems encountered and Improvements

During the implementation process, we encountered several challenges. Initially, we attempted to compute all necessary attributes—including principal curvatures, principal directions, radial curvatures, and their derivatives—on the CPU and pass them directly to the shaders. However, this approach proved to be computationally expensive, ultimately causing the program to crash due to excessive memory usage and data transfer overhead.

Moreover, the resulting visualization (see Fig. 6) was imprecise, with noticeable artifacts such as disconnected loops instead of well-defined curves. To overcome these issues, we opted to compute the radial curvature directly on the GPU. This significantly reduced the performance bottleneck associated with data transmission between the CPU and the shaders.

It is worth noting that the gradient of the radial curvature, which was still computed on the CPU, required recalculating the radial curvature. However, since this computation was only performed when explicitly needed for gradient evaluation—and without continuously passing large data structures to the shaders—the program ran significantly more efficiently, avoiding crashes and ensuring smoother execution.

Another problem, which is also clear from the output, is that using a single pass CPU-shader implementation, for suggestive contours, is not the best approach to obtain precise results.

Fig. 6: Rendering obtained after the first implementation.

As a matter of fact, our shader implementation simply checks whether the radial curvature k_r is close to zero. This "all-or-nothing" thresholding is applied in the fragment shader as follows:

```
if (abs(kappa_r) < curvatureThreshold) {
    fragColor = vec4(1.0, 0.0, 0.0, 1.0);
} else {
    fragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

This method, while straightforward, introduces significant artifacts. Since the thresholding is applied per fragment, it often results in large, noisy loops or blotches, especially in meshes that are not extremely dense or contain small variations in curvature. The moment k_r crosses the threshold, the corresponding fragment immediately changes color, leading to visually unstable and thick suggestive contours that do not consistently align with the geometry.

A more robust approach is to detect actual zero-crossings of the radial curvature k_r at the edges of triangles, rather than relying solely on fragment-level thresholding. This idea originates from the DeCarlo et al. [1] suggestive contours paper, where suggestive contours are formally defined as the set of points where:

- The radial curvature k_r crosses zero, indicating a transition between convex and concave regions.
- The gradient of the radial curvature $D_{\mathbf{w}}k_r$ points towards the viewer, ensuring that the suggestive contour is perceptually relevant.

Instead of evaluating k_r at each fragment, a more principled method involves checking each edge in the mesh. If two vertices v_0 and v_1 connected by an edge have opposite signs of k_r (i.e., one is positive, the other negative), a zero-crossing occurs along that edge. The exact zero-crossing location can be found by linear interpolation:

$$t = \frac{|k_r(v_0)|}{|k_r(v_0)| + |k_r(v_1)|} \tag{5}$$

Using this interpolation factor, the 3D position of the zero-crossing along the edge is:

$$\mathbf{p}(t) = (1-t)\mathbf{p_0} + t\mathbf{p_1} \tag{6}$$

This approach ensures that contours are precisely located where they should be, rather than being arbitrarily defined by a per-fragment threshold. Additionally, an explicit line rendering approach could have been used, rather than relying entirely on fragment-based shading. A line shader, for instance, would allow for smooth, connected suggestive contours, avoiding the thick, disconnected loops that often arise in fragment-based methods.

Several well-established implementations of suggestive contours, including the method from DeCarlo et al., adopt edge-based interpolation and OpenGL line rendering instead of relying solely on per-fragment calculations.

VI. Conclusion

In this project, we implemented an object-space algorithm for detecting suggestive contours on 3D meshes, following the approach proposed by DeCarlo et al. The pipeline involved computing principal curvatures and directions using quadric fitting, estimating radial curvature gradients, and leveraging GPU shaders to efficiently render the contours. Our results demonstrated that suggestive contours enhance the visualization of 3D shapes by providing additional perceptual cues beyond standard silhouettes. Parameter tweaking, particularly for the curvature threshold, played a crucial role in controlling the density and accuracy of the contours.

Despite the successful implementation, some limitations were observed. One major issue was the presence of numerous short segments and spurious contours, which can clutter the final visualization.

A. Future work

A direct extension of this work involves integrating *suggestive highlighting*, as introduced by DeCarlo et al. (2007) in [8]. This method extends the framework of suggestive contours by introducing *highlight lines*, which depict view-dependent highlights to complement contours.

Suggestive contours enhance shape perception by emphasizing *anticipated occluding contours*, effectively abstracting the darkest regions of an image. Highlight lines, on the other hand, capture the *brightest regions*, occurring at maxima of the dot product $\hat{\mathbf{n}} \cdot \hat{\mathbf{v}}$, where $\hat{\mathbf{n}}$ is the normal and $\hat{\mathbf{v}}$ the view direction. Just as suggestive contours approximate intensity valleys in a shading model, highlight lines correspond to intensity peaks.

DeCarlo et al. (2007) propose two types of highlight lines:

- Suggestive Highlights: Appear at *view-dependent inflections*.
- Principal Highlights: Occur when viewing along *principal curvature directions*.

Future research could explore efficient real-time computation of highlight lines, hybrid rendering techniques that combine suggestive contours and highlights, and perceptual studies evaluating their effectiveness in shape understanding. By extending the abstraction of contours to highlights, we can achieve a richer, more intuitive depiction of 3D geometry, further enhancing shape perception in non-photorealistic rendering.

REFERENCES

[1] D. DeCarlo, A. Finkelstein, S. Rusinkiewicz, and A. Santella, "Suggestive contours for conveying shape," *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 22, no. 3, pp. 848–855, 2003.

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

[3] Y. Ohtake, A. Belyaev, and H.-P. Seidel, "Ridge-valley lines on meshes via implicit surface fitting," in *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, (New York, NY, USA), p. 609–612, Association for Computing Machinery, 2004.

[4] Y. Tang, F. Dong, and J. Zhang, "An optimization framework for suggestive contours," in *2024 7th International Conference on Computer Information Science and Application Technology (CISAT)*, pp. 628–634, 2024.

[5] Y. Liu, M. Garland, and L. Kavan, "Neural contours: Learning to draw lines from 3d shapes," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5420–5429, IEEE, 2020.

[6] G. Taubin, "Curve and surface smoothing without shrinkage," in *Proceedings of IEEE International Conference on Computer Vision*, pp. 852–857, 1995.

[7] libigl, "libigl tutorial: Curvature directions," 2023. Accessed: 2025-02-15.

[8] D. DeCarlo and S. Rusinkiewicz, "Highlight lines for conveying shape," in *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '07, (New York, NY, USA), p. 63–70, Association for Computing Machinery, 2007.