6.838: Shape Analysis, 2021
Instructor: Justin Solomon
TAs: David Palmer and Paul Zhang

Geometry Processing for Real-Time Pencil Sketching

Eric Zhang[1] and Victor Rong[2]

[1]Harvard University
[2]Massachusetts Institute of Technology

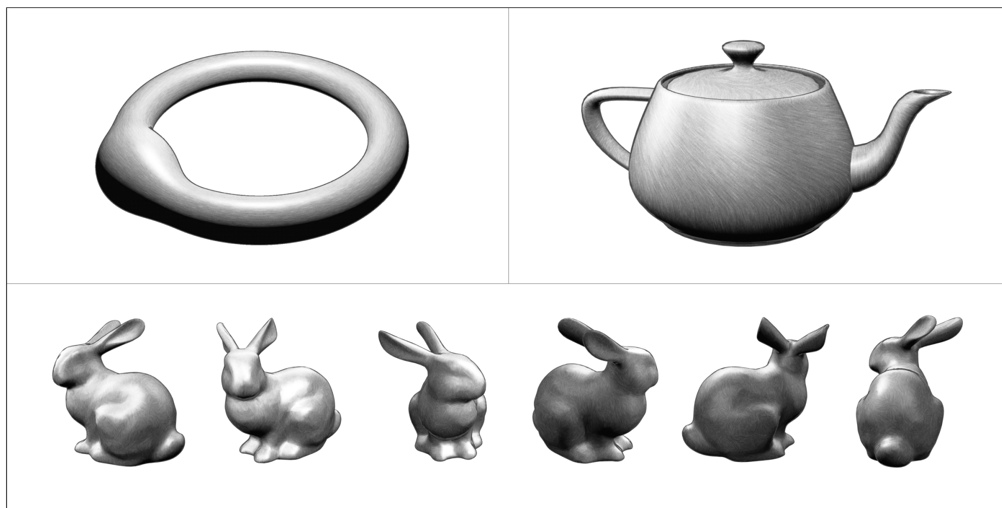

Figure 1: *Pencil sketches rendered in real time: torus SDF (top-left), Utah teapot mesh (top-right), and bunny mesh (bottom).*

Abstract

We explore the task of drawing 3D objects, made of triangle meshes or implicitly represented as signed distance fields (SDFs), in the art style of a pencil sketch. Several theoretical geometric tools, such as principal curvature, are helpful in planning the stroke directions during rendering. We review, implement, and extend existing methods with geometry processing techniques. In the process of exploring mesh curvature methods, we found difficulties in robustly selecting the size of triangles, local tolerance to noise, and other parameters that affect output quality. Our primary contribution is a new scale-invariant algorithm for estimating surface curvatures of an SDF within the graphics pipeline, which has been a previously unexplored topic in the literature. This algorithm has the advantage of enabling real-time rendering of changing geometries at arbitrary scales (modeled by implicit functions), without the noise sensitivity of previous mesh-based methods. Software artifacts can be found on the project web page: `https://pencil-sketching.vercel.app/`.

1. Introduction

The goal of many non-photorealistic rendering methods is to pursue images that efficiently and expressively convey information about shape, while eliminating extraneous details. Line drawings are a particularly effective medium for conveying geometric information, as many properties like stroke width, stroke direction, hatching, tone, contour selection, and density can be freely adjusted and combined with artistic intent. Therefore, our task is to render a smooth surface in a way that resembles a pencil sketch.

One of the most important variables in a sketching algorithm is the choice of line direction. As shown in [GIHL00], selecting line directions based on principal curvatures is an effective means for conveying surface shape in artistic drawings. Furthermore, principal curvatures are invariant of light direction, shading, and view direction, which makes them an ideal candidate to be utilized in rendering algorithms, such as in [PHWF01] and [LKL06]. However, an important issue that stops principal curvatures from being effectively used in practice is robustness to local noise. Trian-

Figure 2: *A stylized artwork image from [Qui15], rendered in real time by ray tracing a signed distance field with WebGL.*

gle meshes consist of many finite flat elements that approximate a smooth 3D surface. Since surface curvature is an inherently local property, it is very sensitive to small perturbations in the locations of points. Mesh-based methods have issues dealing with low-fidelity and noisy meshes, such as those obtained from a 3D scan, without additional smoothing steps. Even then, it is difficult and error-prone to determine what *scale* of smoothing to supply. Too high of a scale, and important surface features will be lost. Too low of a scale, and the computed principal directions will be incoherent. Sometimes we would even ideally desire adaptively choosing smoothing scales throughout the mesh, as not all shape models are regular with uniformly distributed near-equilateral triangles, which further complicates the problem.

An alternative, fairly popular recent approach to geometry modeling is to use signed distance fields, which are implicit functions that mathematically describe the surface of a shape. These functions can be quickly derived and naturally produce effects like extrusion, transformation, constructive solid geometry, and rounding of corners with very little modeling work. Signed distance fields have also been used in various contexts like differentiable rendering and inverse graphics [PFS*19], similar to other implicit representations that lend themselves to backpropagation. They are also used in real-time ray tracing contexts, as demonstrated in Figure 2.

Signed distance fields are mathematically elegant, have strong smoothness properties, are scale-invariant, and are not subject to the pathological cases that arise in badly formed polygonal meshes. This makes them an ideal candidate for computing local properties such as surface curvature directions.

In this project, we implement non-photorealistic pencil rendering based on various curvature estimation algorithms described in the literature and make the following contributions:

- A **new scale-invariant method for principal curvature** estimation on implicit geometries modeled by a signed distance field.
- An **efficient GPU implementation** of our method that is suitable for real-time rendering, including an interactive demonstration that runs on any modern web browser using WebGL.

We also compare our SDF rendering method in efficiency and robustness to previous pencil sketching algorithms based on curvature estimation on triangle meshes.

2. Related Work

Computation of curvature and the principal curvature directions is a well studied problem. [Tau95] computes a per-vertex matrix such that its eigenvalues correspond to terms easily expressed by the principal curvature and its eigenvectors correspond to exactly the principal curvature direction. [Rus04] uses the differences of normals in each vertex's 1-ring to estimate its second fundamental form.

Also in non-photorealistic rendering, an object-space algorithm for suggestive contours was derived based on radial curvature [DFRS03]. More modern work on neural contours build on these same ideas, relying on curvature features to learn to create line drawings [LNHK20].

Prior work in [MBF92] uses partial derivatives of 3D biomedical images to compute principal directions at the isosurface of an intensity function. However, unlike our work, they use discrete 3D images with scalar voxel values. Our method is simpler and uses implicit signed distance functions to model shapes at any level of detail, which makes it practical for ray-traced rendering. It is also easy to parallelize without having to send large amounts of 3D intensity data to the GPU.

Real-time pencil sketching was explored in [LKL06], which proposes an interior shading method that aligns the strokes with a discrete principal curvature direction at each vertex. The computation for this follows [ACSD*03] which also performs smoothing on the directional field. An older cross-hatching paper is [PHWF01], which uses stylized textures. Both of these papers use principal curvature estimation methods on a triangle mesh.

We were inspired to build WebGL-based interactive rendering demos by the non-photorealistic sketches in [San20]. Similarly, our interactive demos interfaced with [Lys16], a lightweight layer on top of standard graphics APIs. There are other alternatives we did not pursue but are worth mentioning for completeness. The primary fully featured open-source library in web graphics is `three.js`, which supports various materials, volumetrics, lights, and objects in 3D scenes. Also, we explored running compute-intensive geometry processing code on the CPU using WebAssembly [HRS*17], which is a compilation target of C++ and Rust. This has been used in industry by companies such as *Figma* to achieve real-time performance in their graphical applications [Wal17].

3. Technical Approach

In this section, we describe the geometric methods that we used to compute principal directions on a surface. We also provide a more detailed explanation of signed distance fields.

3.1. Curvature Approximation on Triangle Meshes

We tested both Taubin's algorithm and Rusinkiewicz's algorithm for computing the per-vertex principal curvature directions to be used in directing the stroke textures [Tau95, Rus04]. The backbone of both is to estimate a matrix for each vertex from their 1-ring neighborhood whose eigenvectors are the principal curvature directions.

Taubin's algorithm constructs a curvature matrix, which in the continuous setting is defined by

$$M_{\mathbf{p}} := \frac{1}{2\pi}\int_{-\pi}^{\pi} \kappa_\theta t_\theta t_\theta^T \, d\theta.$$

This matrix is not the second fundamental form, but it has the principal curvature directions as its eigenvectors. Taubin's method discretizes this integral by considering a weighted sum across the mesh edges adjacent to $\mathbf{p}$.

Rusinkiewicz's algorithm attempts to directly find the second fundamental form $\mathbf{II}$ using its definition as the shape operator. For each triangle neighboring a point $\mathbf{p}$, we can approximate the per-face Weingarten matrix as it applies to each edge of the triangle. More precisely, say that our triangle has edges $\mathbf{e}_{12}, \mathbf{e}_{23}, \mathbf{e}_{31}$ and vertex normals $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$. Let $\mathbf{u}, \mathbf{v}$ be an orthonormal basis of the triangle's plane. We want to solve for $\mathbf{II}$ such that for all i and j,

$$\mathbf{II}\begin{pmatrix}\mathbf{e}_{ij}\cdot\mathbf{u}\\ \mathbf{e}_{ij}\cdot\mathbf{v}\end{pmatrix} = \begin{pmatrix}(\mathbf{n}_j-\mathbf{n}_i)\cdot\mathbf{u}\\ (\mathbf{n}_j-\mathbf{n}_i)\cdot\mathbf{v}\end{pmatrix}.$$

This gives six equations to solve for the three unknown entries of $\mathbf{II}$. A simple least squares fit gives a good $\mathbf{II}$. These face-wise matrices are then carefully transformed into the plane orthogonal to the vertex normal of $\mathbf{p}$ and then aggregated in a weighted average.

As both methods relied heavily on the 1-ring neighborhood, ensuring the correct connectivity of the mesh was crucial. Often, two faces in a mesh are parts of separate sections of the mesh's surface (e.g. the sides of a box), and combining their vertices would lead to incorrect smooth shading of vertex normals. We used the *edge split* modifier in Blender to enable smooth computation of vertex normals, even at sharp corners [Ble21].

Furthermore, we can adapt these methods to be done in a single pass of the fragment shader. Following [PVK16], we can compute screen-space curvature given the screen-space vertex positions and normals.

3.2. Signed Distance Fields

Signed distance fields (SDFs) provide an alternative representation of geometric shapes, modeled by implicit functions. Instead of approximating the surface of a compact shape $\mathcal{M}$ by a collection of discrete triangles, an SDF is a function $f_{\mathcal{M}} : \mathbb{R}^3 \to \mathbb{R}$ taking

$$f_{\mathcal{M}}(\mathbf{p}) = \min_{\mathbf{x}\in\mathcal{M}} d(\mathbf{x},\mathbf{p}),$$

where d represents the Euclidean metric on $\mathbb{R}^3$. Given a manifold in $\mathbb{R}^3$, signed distance fields satisfy the *Eikonal equation* $\|\nabla f_{\mathcal{M}}(\mathbf{p})\|_2 = 1$ almost everywhere, due to being a distance function. Also, we can see that $f_{\mathcal{M}}(\mathbf{p}) = 0$ if and only if $\mathbf{p} \in \mathcal{M}$.

Furthermore, if we assume that the manifold $\mathcal{M}$ is the boundary of some 3D shape, then we can modify the signed-distance field definition to satisfy the Eikonal equation at almost all points on $\mathcal{M}$ as well, by setting the value of the distance function to be negative at points in the interior of the shape. For example, the signed distance field representing a sphere of radius r would be

$$f_{\mathcal{M}}(\mathbf{p}) = \|\mathbf{p}\|_2 - r.$$

The contours of this function are visualized in Figure 3.

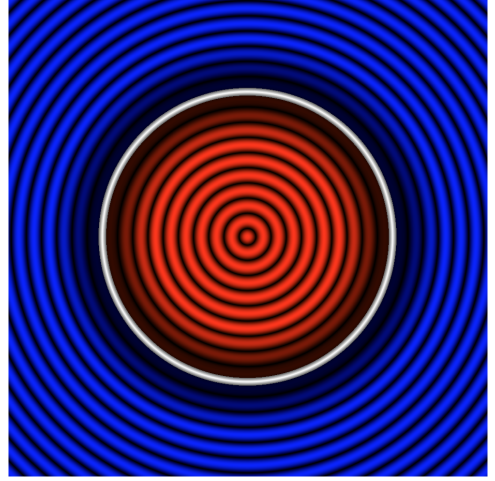

Figure 3: *Cross-section of the sphere SDF (red is negative).*

Signed distance fields can either be constructed by hand or automatically generated by shape learning techniques from computer vision, as in [LNHK20]. For the hand-constructed case, there are several explicit formulas for simple SDFs listed in [Qui13], along with various combinators that produce transformations, binary constructive solid geometry, and smooth variants of those operations. This allows one to compactly represent a large number of shapes, both geometric and organic, with very little modeling effort.

In contrast to triangle meshes, which can be rasterized as individual faces or ray traced with mesh acceleration data structures (as described in [PJH16]), implicit shapes with SDFs can be efficiently ray traced using the field as a functional distance estimator. This has the advantage of enabling real-time ray tracing without specialized graphics hardware, and it is also scale-invariant, with no individual face data being required.

3.3. Curvature Estimation on SDFs

One interesting property of SDFs, as an implicit representation, is greater smoothness at local scales compared to explicit triangle meshes. The main problem with curvature estimation algorithms, such as those described in §3.1, is sensitivity to local noise. Oftentimes, surface fairing algorithms are needed to correct for this. In addition, curvature estimation algorithms tend to be expensive and run in a separate step from the rendering process.

In this section, we describe our new curvature estimation algorithm, which can compute principal curvatures at each point of a signed distance field. The algorithm is easily parallelizable, running in real time as an integrated part of the rendering process.

The first step is to compute the normal vector. Suppose that we have a point $\mathbf{p}$ on a 2-manifold $\mathcal{M} \subset \mathbb{R}^3$, with SDF $f_{\mathcal{M}}$. Then, the outward-pointing normal vector $\mathbf{n}$ is given by the gradient

$$\mathbf{n} = \nabla f_{\mathcal{M}}(\mathbf{p}).$$

We can estimate this quantity without needing to take symbolic

Figure 4: *Generated textures at varying shades of darkness.*

derivatives, by instead taking values of the signed distance field at points close to $\mathbf{p}$ and using the discrete difference quotient.

After estimating the normal vector, let $\mathbf{u}$ and $\mathbf{v}$ be orthogonal vectors that form a positively-oriented orthonormal basis for the tangent plane $T_{\mathbf{p}}\mathcal{M}$. We can approximate the second fundamental form at $\mathbf{p}$ by the Hessian matrix

$$\mathcal{H}_{(x,y)}\left[f_{\mathcal{M}}(\mathbf{p}+x\mathbf{u}+y\mathbf{v})\right] = \begin{bmatrix} f_{xx} & f_{xy} \\ f_{xy} & f_{yy} \end{bmatrix},$$

where for small $\varepsilon > 0$, we approximate the mixed partials by

$$\begin{aligned} f_{xx} &\approx \frac{1}{\varepsilon^2}\left[f_{\mathcal{M}}(\mathbf{p}+\varepsilon\mathbf{u}) + f_{\mathcal{M}}(\mathbf{p}-\varepsilon\mathbf{u}) - 2f_{\mathcal{M}}(\mathbf{p})\right], \\ f_{yy} &\approx \frac{1}{\varepsilon^2}\left[f_{\mathcal{M}}(\mathbf{p}+\varepsilon\mathbf{v}) + f_{\mathcal{M}}(\mathbf{p}-\varepsilon\mathbf{v}) - 2f_{\mathcal{M}}(\mathbf{p})\right], \\ f_{xy} &\approx \frac{1}{4\varepsilon^2}\left[f_{\mathcal{M}}(\mathbf{p}+\varepsilon\mathbf{v}+\varepsilon\mathbf{u}) + f_{\mathcal{M}}(\mathbf{p}-\varepsilon\mathbf{v}-\varepsilon\mathbf{u})\right. \\ &\qquad \left. - f_{\mathcal{M}}(\mathbf{p}+\varepsilon\mathbf{v}-\varepsilon\mathbf{u}) - f_{\mathcal{M}}(\mathbf{p}-\varepsilon\mathbf{v}+\varepsilon\mathbf{u})\right]. \end{aligned}$$

Since this is a symmetric 2×2 matrix, it can be diagonalized to estimate the principal curvature directions (eigenvectors) and eigenvalues at $\mathbf{p}$. Letting $D = \sqrt{(f_{xx} - f_{yy})^2 - 4f_{xy}^2}$, the eigenvectors and corresponding eigenvalues are

$$\begin{aligned} \lambda_1 &= \frac{f_{xx}+f_{yy}+D}{2}, & v_1 &= (2f_{xy})\mathbf{u} + (f_{yy} - f_{xx} + D)\mathbf{v}, \\ \lambda_2 &= \frac{f_{xx}+f_{yy}-D}{2}, & v_2 &= (2f_{xy})\mathbf{u} + (f_{yy} - f_{xx} - D)\mathbf{v}. \end{aligned}$$

These mathematical formulas are branchless and can be efficiently evaluated in parallel in a GPU shader, computing surface principal curvatures at every point visible on the screen.

4. Implementation

In this section, we describe the practical design of our method and share some of our choices, challenges, and tradeoffs in writing a rendering pipeline for pencil sketches. Similar to [San20], we implemented our software artifacts in WebGL to produce interactive demos that can be run on any modern web browser. We feel that this choice makes the software much more accessible.

4.1. Texture Generation

Textures for a variable number of shades are created to be used in the rendering pipeline. For each texture, a number of strokes are simulated upon an integer array representing the canvas. The path of each stroke follows a simple forward integration equation with a small amount of white noise added. The strokes wrap around the edges of the texture. For each point which this path steps on, the neighborhood of pixels with a distance of 0.5 pixels from the point is shaded in where the weight is stronger for close pixels. The per-pixel shading is done as described in §5.1 of [LKL06].

To simulate the variety of shades seen in Figure 4, each texture is parameterized by a number $d \in [0,1]$ representing the average darkness of the texture with 0 being a white texture with no strokes and 1 being completely black. This term implicitly controls the number of strokes draw. When the strokes are being drawn, the sum of values among the pixels, which are each between 0 and 255, is maintained. When it drops below $255 \cdot (1-d)WH$, no more strokes are drawn and the texture is complete. The weight of each stroke is also proportional to d^2 though it has a lower limit so that the strokes are not unrealistically light.

4.2. Screen-Space Texture Blending

After textures are synthesized, they need to be painted onto a surface and blended, taking advice from the principal curvature field. This is a nontrivial task, so we will describe our approach for both mesh rasterization and SDF ray tracing.

4.2.1. Rasterization

In the rasterization pipeline, since we have triangles at each vertex, we can simply texture each triangle individually using a blend of three textures. First, every vertex has its precomputed curvature directions loaded from Taubin's method, and we compute Gourard shading based on vertex normals and a simple lighting algorithm. We also project the curvature, which is a unit vector in $\mathbb{R}^3$, into screen space based on the projective camera. These three pieces of data (color, normal vector, and screen-space curvature vector) are outputs of the vertex shader.

Then, at each fragment of a triangle, we compute its barycentric coordinates. We then blend textures centered at each vertex shader, chosen from our precomputed map based on the luma intensity at that vertex. The textures are rotated in screen space to prevent distortion and ensure even pencil lines, based on the curvature vector. This produces smoothly varying lines that avoid the sharp discontinuities seen in previous methods, such as [PHWF01].

As a technical detail, we had to create three distinct vertices for each face to have enough information for our rendering method. We could not share vertices between adjacent faces. The reason is that the standard OpenGL pipeline does not support fetching information about individual vertices of the triangle within the fragment shader, so we instead encoded that information directly by using three varying outputs. This slightly increases memory footprint, since the number of vertices is usually around half the number of faces in a regular mesh, but this issue can be resolved fairly reasonably by rendering the faces in batches.

4.2.2. Real-Time Ray Tracing

In the case of real-time ray tracing, our algorithm effectively computes world-space principal curvature vectors at each point in the camera frame. However, it is unclear how to use these curvatures to render an image that appears smooth and well-textured.

One approach that we initially tried was to sample a value at each

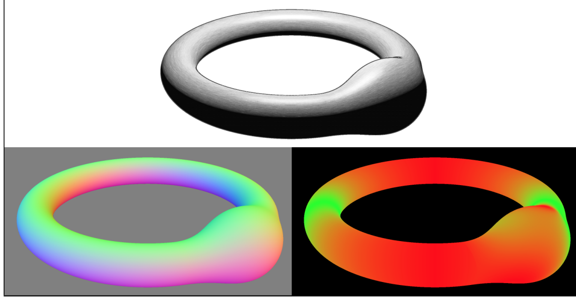

Figure 5: *Composited textures (top), normal vectors (bottom-left), and principal curvature directions (bottom-right) for a torus melded with a sphere using SDF operations.*

point based on a texture coordinate sampled from a smooth function of the curvature direction and screen coordinates. In the case of constant curvature direction, this would ideally tile the same pencil texture repeatedly on the screen. We used 2D screen-space projections of the curvature direction, rather than the 3D world-space curvature direction, due to $\mathbb{RP}^2$ being non-orientable. However, this approach had several issues:

- Composited textures would form strange hyperbolic artifacts and Moiré patterns, since the principal curvature direction would smoothly vary, distorting the straight lines of the pencil textures.
- If we used many textures (one texture for each of the 256 levels of brightness), then in most areas of the image where lighting is non-constant, the selected texture at each pixel would change rapidly, forming a noisy pattern that loses the pencil effect. Conversely, using few textures leads to undesirable banding.

To fix these problems, we devised the following two-pass rendering method. In the first pass, we compute luma, normals, and principal directions at each pixel and store that as texture data in framebuffers. Then, in the second pass, we fix some *scale* parameter S and divide the screen into a grid of $(S \times S)$-pixel squares. At each vertex of unit squares, we sample the luma intensity and use that to select one of our precomputed pencil textures. Then, we tile that texture centered at the vertex and rotate it so that the lines are pointing in the principal direction. At each interior point of a grid cell, we perform bilinear interpolation on the texture values centered at each of the four vertices. We then adjust the color by scaling to match the true Phong shaded luma value, compute contours using the normal value, and output the final color.

The advantage of this method is that texturing becomes *locally flat* rather than distorted, so our pencil textures retain convincing straight line patterns. It also allows us to efficiently implement rendering without having to regenerate textures at every pass. An example of our compositing is shown in Figure 5, and the grid does not produce noticeable artifacts in this example.

4.3. Cartoon Contours

Our method for rendering contours in both mesh and SDF renderers was based on screen-space partial derivatives. We modified a short

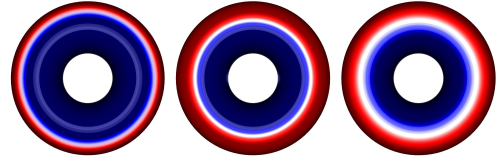

Figure 6: *Top view of the Gaussian curvatures of a torus computed with Taubin's algorithm (left), Rusinkiewicz's algorithm (middle), and the exact analytic equations (right).*

segment of fragment shader code from [Reu20]. This code considers the term $\mathbf{v} \cdot \mathbf{n}$, where $\mathbf{v}$ is the view ray and $\mathbf{n}$ is the surface normal. It draws a contour in regions where $\mathbf{v} \cdot \mathbf{n} / \|\nabla(\mathbf{v} \cdot \mathbf{n})\|_1$ is larger than a threshold, approximating a constant-width line whenever the surface normal is orthogonal to the view ray. This is similar to the basic screen-space algorithm described in [DFRS03]. We did not implement suggestive contours computed with screen-space partial derivatives of curvature direction, although it would be easy to extend our method to support this.

5. Results

We qualitatively examine results for both mesh curvature approximation and SDF ray tracing methods. Both methods easily run in real-time in the browser, at greater than 30 FPS, but the exact speed depends on factors like screen size and graphics hardware. We omit a detailed performance analysis due to lack of standard hardware (such as a high-end graphics card).

5.1. Curvature Approximation on Triangle Meshes

As detailed in §3.1, we evaluated the mesh curvature algorithms in [Tau95] and [Rus04] and used both in rendering the stylized images. We found that both gave visually aesthetic directional fields, though Rusinkiewicz generally gives more accurate approximations. The stroke directions defined in Figure 8 reflect the bunny's curvatures but are noisy due to the smaller bumps of the mesh, particularly in the middle and lower regions of the bunny. We also demonstrate real-time curvature computations on a triangle mesh in Figure 9.

On the other hand, the principal curvature values $\kappa_{\min}$ and $\kappa_{\max}$ were often incorrect in both algorithms. Figure 6 illustrates the Gaussian curvature, which is equal to $\kappa_{\min}\kappa_{\max}$, on a torus. We conjecture that the reason why the curvature values are wrong while the directions are mostly aligned correctly is because area weighting in the algorithms affects the eigenvalues of the obtained matrix more strongly than it affects the eigenvectors.

The incorrect eigenvalues impacts which eigenvector is selected as the minimal principal curvature direction. Ironically, the Taubin implementation looks better on many meshes because it is wrong. The Taubin implementation has a tendency for the eigenvalues to never swap order. This means that it consistently chooses the same eigenvector, whereas the Rusinkiewicz implementation swaps between perpendicular directions as visible in Figure 7.

Figure 7: *Utah teapot rendered using the minimal principal directions from the Rusinkiewicz implementation. The strokes defining the bottom of the teapot, though correct, go against the strokes defining the walls.*

5.2. Real-Time Ray Tracing of SDFs

We provide two large, full-resolution example images that were rendered from a signed distance field and our compositing method. Figure 10 shows a common constructive solid geometry model, which was built out of combining simple sphere, cylinder, and cube SDFs. Once again, there are not any noticeable artifacts from the grid-based algorithm. The cylindrical interior portions of the shape have consistent principal curvature directions. However, the flat and spherical portions have no distinguished principal directions, since the second fundamental form has two equal eigenvalues.

Figure 11 illustrates an organic shape, built out of a smooth union operation between two tori and a rounded pill primitive. Here, the principal curvature algorithms are able to produce consistent orientations on a vast majority of the surface. Our contour algorithm based on screen-space partial derivatives also produces a very smooth and desirable result.

5.3. Limitations

As discussed in our analysis of Figure 10, many geometric shapes have sections that are perfectly spherical or flat. This leads to problems when trying to compute principal directions, as they are ambiguous. To smoothly shade such shapes, we would need to extend the principal curvature direction field smoothly to regions that lack a well-defined direction, by perhaps using parallel transport, vector field diffusion, or other methods. We would be interested in future work that explores this.

6. Contributions

Eric wrote the framework and initial shader code for the project, developed the SDF principal curvature algorithm, modeled and cleaned some meshes with Blender, and implemented the rendering pipeline. Victor implemented pencil texture generation and developed Python scripts to load meshes, subdivide them, and compute curvature with algorithms by Taubin and Rusinkiewicz. Both authors contributed to background research and writing.

References

[ACSD*03] ALLIEZ P., COHEN-STEINER D., DEVILLERS O., LÉVY B., DESBRUN M.: Anisotropic polygonal remeshing. In *ACM SIGGRAPH 2003 Papers*. ACM, 2003, pp. 485–493. 2

[Ble21] BLENDER ONLINE COMMUNITY: Blender - a 3d modelling and rendering package, 2021. URL: `http://www.blender.org`. 3

[DFRS03] DECARLO D., FINKELSTEIN A., RUSINKIEWICZ S., SANTELLA A.: Suggestive contours for conveying shape. In *ACM SIGGRAPH 2003 Papers*. ACM, 2003, pp. 848–855. 2, 5

[GIHL00] GIRSHICK A., INTERRANTE V., HAKER S., LEMOINE T.: Line direction matters: an argument for the use of principal directions in 3d line drawings. In *Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering* (2000), pp. 43–52. 1

[HRS*17] HAAS A., ROSSBERG A., SCHUFF D. L., TITZER B. L., HOLMAN M., GOHMAN D., WAGNER L., ZAKAI A., BASTIEN J.: Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017), pp. 185–200. 2

[LKL06] LEE H., KWON S., LEE S.: Real-time pencil rendering. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering* (2006), pp. 37–45. 1, 2, 4

[LNHK20] LIU D., NABAIL M., HERTZMANN A., KALOGERAKIS E.: Neural contours: Learning to draw lines from 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2020), pp. 5428–5436. 2, 3

[Lys16] LYSENKO M.: Regl: Fast functional WebGL. `https://github.com/regl-project/regl`, 2016. 2

[MBF92] MONGA O., BENAYOUN S., FAUGERAS O. D.: From partial derivatives of 3-d density images to ridge lines. In *Visualization in Biomedical Computing'92* (1992), vol. 1808, International Society for Optics and Photonics, pp. 118–129. 2

[PFS*19] PARK J. J., FLORENCE P., STRAUB J., NEWCOMBE R., LOVEGROVE S.: Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the Conference on Computer Vision and Pattern Recognition* (2019), pp. 165–174. 2

[PHWF01] PRAUN E., HOPPE H., WEBB M., FINKELSTEIN A.: Real-time hatching. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), p. 581. 1, 2, 4

[PJH16] PHARR M., JAKOB W., HUMPHREYS G.: *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016. 3

[PVK16] PRANTL M., VÁSA L., KOLINGEROVÁ I.: Fast screen space curvature estimation on GPU. In *VISIGRAPP (1: GRAPP)* (2016), pp. 151–160. 3

[Qui13] QUILEZ I.: 3d distance functions. `https://iquilezles.org/www/articles/distfunctions/distfunctions.htm`, 2013. 3

[Qui15] QUILEZ I.: Greek temple. `https://www.shadertoy.com/view/ldScDh`, 2015. 2

[Reu20] REUSSER R.: Sphere eversion. `https://rreusser.github.io/explorations/sphere-eversion/`, 2020. 5

[Rus04] RUSINKIEWICZ S.: Estimating curvatures and their derivatives on triangle meshes. In *Proceedings. 2nd International Symposium on 3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004.* (2004), IEEE, pp. 486–493. 2, 5

[San20] SANCHEZ J.: Sketch: Explorations on cross-hatching, engraving, and similar non-photorealistic rendering. `https://github.com/spite/sketch`, 2020. 2, 4

[Tau95] TAUBIN G.: Estimating the tensor of curvature of a surface from a polyhedral approximation. In *Proceedings of IEEE International Conference on Computer Vision* (1995), IEEE, pp. 902–907. 2, 5

[Wal17] WALLACE E.: WebAssembly cut Figma's load time by 3x. `https://tinyurl.com/5s34axju`, 2017. 2

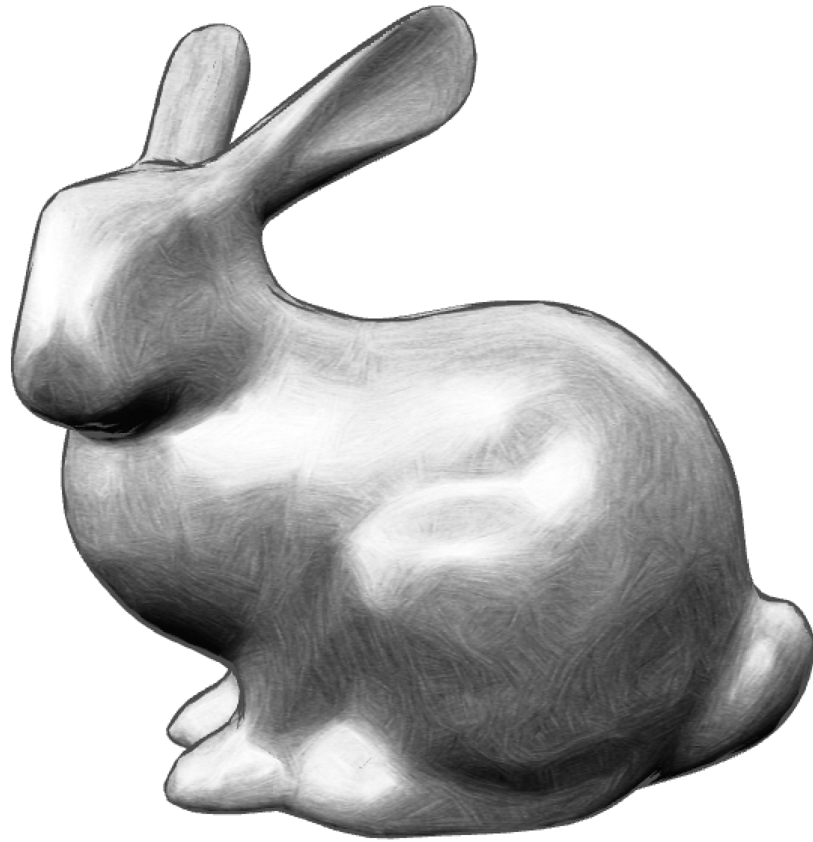

Figure 8: *Rendered triangle mesh of a bunny, from the Stanford 3D Scanning Repository.*

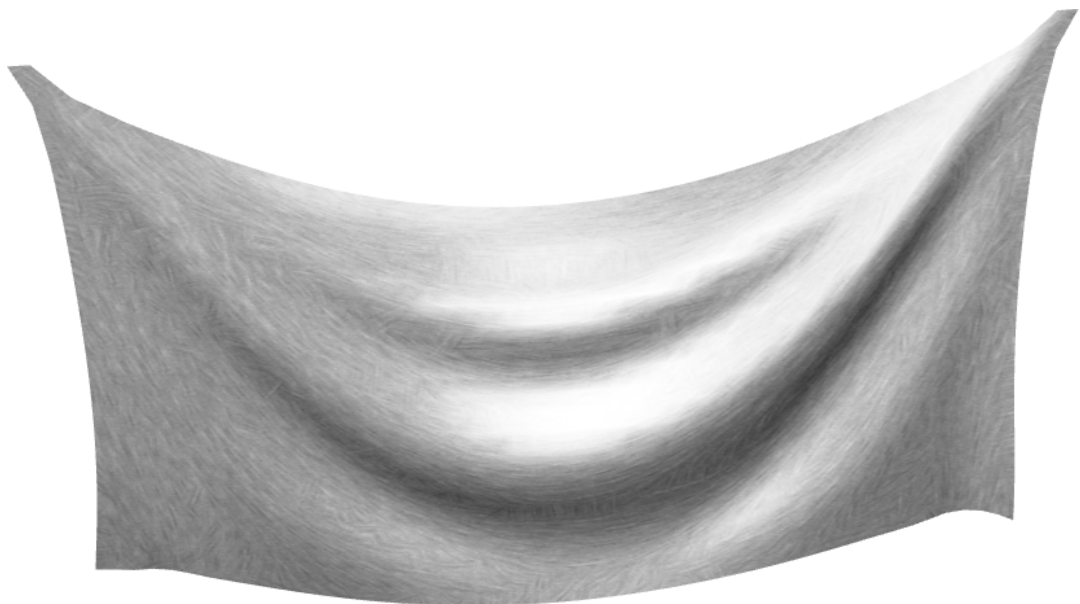

Figure 9: *Cloth simulation with principal directions computed in real time. This demo can be found on the project web page.*

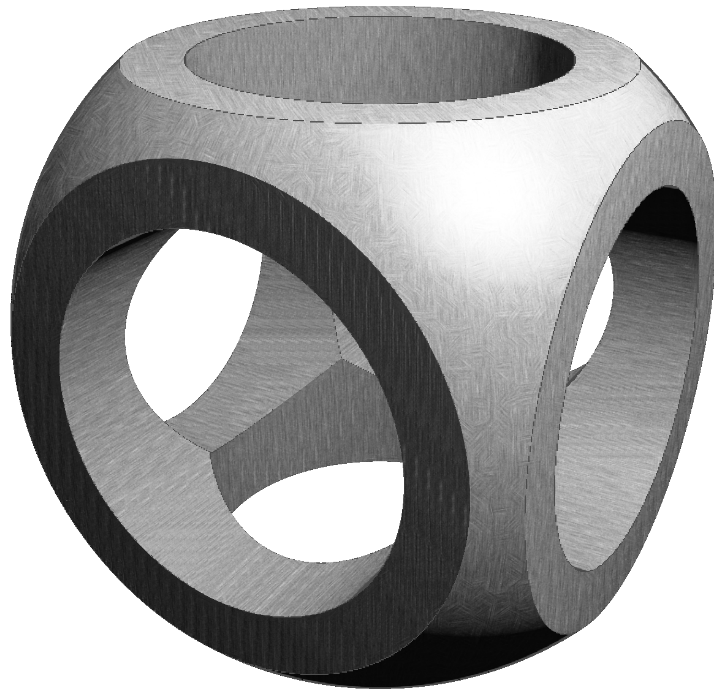

Figure 10: *Constructive geometry model rendered using an SDF built from sphere, cylinder, and cube primitives.*

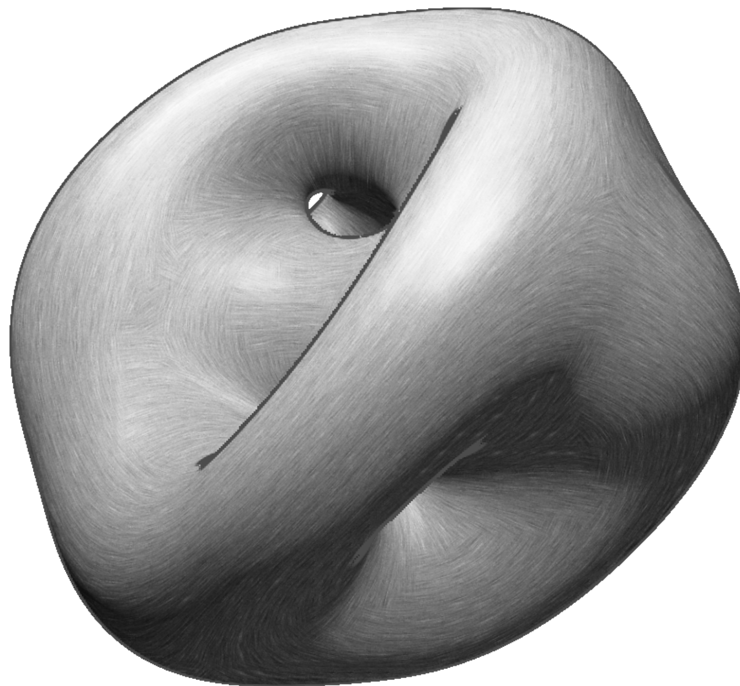

Figure 11: *Blobs model rendered using an SDF, illustrating an organic shape with well-defined principal curvature.*