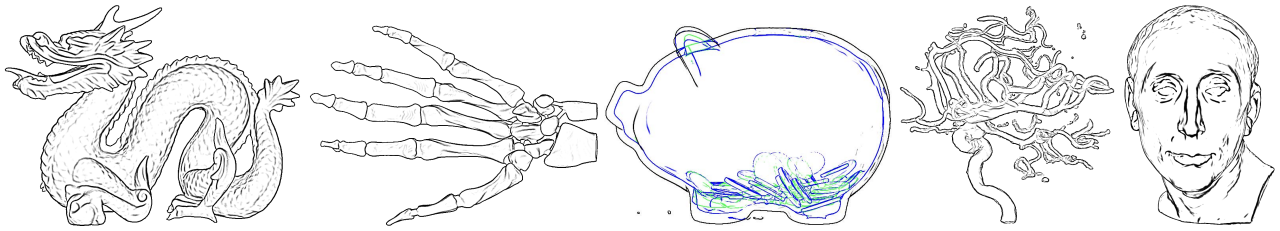
Splatting Lines: An Efficient Method for Illustrating 3D Surfaces and Volumes

Long Zhang[†‡] Qian Sun[‡] Ying He[‡ *]
[†]Hangzhou Dianzi University [‡]Nanyang Technological University



(a) 1.2M points, 65 fps (b) 327K points, 180 fps (c) $512 \times 512 \times 134$, 26 fps (d) 256^3, 160 fps (e) 1.8M tris, 24fps

Figure 1: *Our algorithm can generate visually-pleasing line drawings from various types of models in real time.*

Abstract

While 3D line drawing techniques have become well established in the past decade, most are only suitable for polygonal meshes. This paper presents splatting lines, a unified framework for generating line drawings from various types of 3D models, including point clouds, volumes, and polygonal meshes. In contrast to the existing mesh-based approaches, our method takes a densely sampled point cloud as input. It renders two diffuse shading images using splatting with different parameters and then generates line drawings by subtracting these images. Our point-based approach can be easily extended to polygonal meshes and volumes by an efficient (iso-)surface sampling method. Our method is highly efficient and it does not require any pre-computation. It is also more robust to the mesh tessellation than existing mesh-based techniques. Experimental results indicate the advantages of our method in terms of both performance and quality.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling —Curve, surface, solid, and object representations

Keywords: line drawing, shape illustration, splatting

1 Introduction

Line drawing is a popular shape depiction technique because it can express meaningful information by ignoring less important or distracting details. The generation of high-quality line drawings from a 3D model has received a great deal of attention in the graphics community since the early 2000s, during which time many elegant line drawing algorithms have been proposed. The representative work includes suggestive contours [DeCarlo et al. 2003], ridge-valley lines [Ohtake et al. 2004], apparent ridges [Judd et al. 2007], photic extremum lines [Xie et al. 2007], principal and suggestive highlights [DeCarlo and Rusinkiewicz 2007], abstracted shading based highlights [Lee et al. 2007], demarcating curves [Kolomenkin et al. 2008], relief edges [Kolomenkin et al. 2009], Laplacian lines [Zhang et al. 2009], and many others. Most of these algorithms are mesh-based, which means they produce lines by iterating each edge or face of the polygonal mesh. As a result, these approaches cannot be directly adapted to other geometric domains, such as point clouds and implicit/parametric surfaces. It is also difficult to apply these mesh-based algorithms to volumes unless the iso-surfaces are explicitly extracted, which is a time-consuming process.

To date, only a few algorithms have been shown to work for domains other than meshes. Some methods [Whelan and Visvalingam 2003; Xu et al. 2004; Zakaria and Seidel 2004] draw silhouettes on point models, but are not able to convey the interior of the shapes. Other approaches, like [Pauly et al. 2003; Daniels et al. 2007; Daniels Ii et al. 2008], can detect more features for point-based shape analysis, but cannot produce visually pleasing line drawings. The marching line (ML) [Thirion and Gourdon 1996] algorithm is a practical solution to extract feature lines from volumes. Burns et al. [2005] and Zhang et al. [2011] adopted the ML method to draw suggestive contours and Laplacian lines from volumes. However, their algorithms are too costly to be performed on a per-voxel basis. In order to improve performance, Burns et al. [2005] suggested a seed-and-traversal approach that traces the lines from randomly selected voxels. Although this strategy significantly improves the rendering speed, the computed lines are highly dependent on the random seeds and lack temporal coherence. Furthermore, these methods use finite difference among neighboring voxels to calculate high-order derivatives of some geometry properties, which is known to be sensitive to data noises.

In this paper, we propose splatting lines[1], a unified framework for generating visually pleasing line drawings for a wide variety of geometric domains, including point clouds, polygonal meshes and volumes. See Figure 1 for some results. Our approach is based on two widely-studied techniques: the Difference-of-Gaussian (DoG) for edge detection and the splatting for rendering point clouds. The DoG edge detector has proven to be effective and efficient in terms of generating high-quality line drawings from static images [Kang et al. 2007], videos [Kyprianidis and Kang 2011], and 3D models [Zhang et al. 2012]. Given a 2D signal (such as an image) as input, the DoG operator extracts the high-frequency feature lines by first filtering the signal twice, using Gaussian filters with different kernel sizes, and then subtracting the resulting two filtered signals. Despite its simplicity, the DoG filter can generate robust and smooth results for a wide range of models. On the other hand, splatting [Zwicker et al. 2001; Linsen et al. 2007] is a standard method with which

*E-mail:yhe@ntu.edu.sg (Y. He).

I3D 2014, March 14 – 16, 2014, San Francisco, California, USA.

[1]Part of this work [Sun et al. 2013] was presented at ACM Symposium on Interactive 3D Graphics and Games (I3D'13).

to render the discrete point models and volumes. The basic idea of splatting is to diffuse the property (for example, illumination) of discrete points to its neighborhood, and generate a smooth shading image by blending the illumination of neighboring points. The key observation of this paper is that the splatting process is analogous to filtering the discrete signal (that is, the illumination of all points) using a Gaussian filter. Therefore, we can mimic the DoG edge detector by rendering two diffuse shading images using splatting with different parameters, and then subtracting the two resulting shading images to obtain the line drawings. Our algorithm has the following features, which distinguish it from the existing approaches:

1. Thanks to its point-based rendering nature, our method is more robust to mesh tessellation and geometric noises than the existing mesh-based methods.
2. Besides polygonal meshes, our method can generate line drawings from more types of models, including point models and volumes.
3. Our method is highly efficient and it does not require any pre-computation. Its parallel structure allows implementation on the GPU. It can also generate smooth and coherent line drawings from deformable models in real-time.

2 Previous Work

Object-space line drawing algorithms compute the feature lines on the 3D objects directly. Popular object-space lines include silhouettes, suggestive contours, suggestive highlights, apparent ridges, photic extremum lines (PELs), demarcating curves, Laplacian lines, and relief edges. Feature lines are defined as the loci of points on the surface at which some geometric constraints hold. For example, suggestive contours are the points where the radial curvature is zero, apparent ridges are the points that maximize the view-dependent curvature, etc. All the object-space techniques adopt a common two-pass computational framework: In the first pass, they compute some geometry-based feature function at each vertex. In the second pass, they iterate the triangles and check the function values for all three vertices of each triangle. In order to compute the geometric properties, most object-space algorithms simply apply the discrete differential geometry operators to the vertex's one-ring neighbors. However, such implementations are highly dependent on the mesh quality. Therefore, surface smoothing and remeshing are often required before the line drawing algorithms can be applied. Some methods [Zhang et al. 2009] [Kolomenkin et al. 2009] address the robustness issue by evaluating the feature function based on a large neighborhood. Although the computation is more expensive than the discrete differential geometry method, it can be performed in the preprocessing stage. As a result, these methods can produce visually smoothing results while maintaining high run-time efficiency. However, the long pre-computing time and high storage requirement means it is not practical to apply the pre-computing approaches to dynamic models.

Image-space techniques generate line drawings by using certain image processing techniques. Saito and Takahashi [1990] pioneered an image-space drawing algorithm for discontinuities, edges, contour lines, and curved hatching. Buchanan and Sousa [2000] proposed the edge buffer data structure to efficiently highlight silhouette edges, boundary edges, and any artist-defined edges. Utilizing a camera with multiple flashes to effectively distinguish depth discontinuities from material discontinuities, Raskar et al. [2005] proposed an NPR approach to capture and convey shape features of real-world scenes. Winnemöller et al. [2006] presented an automatic, real-time video and image abstraction framework using difference-of-Gaussian edges. Lee et al. [2007] proposed an algorithm to extract ridges and valleys from an abstract shaded image. The resulting line drawings effectively convey both shape and material cues. Vergne et al. [2009] proposed a novel light warping technique that can significantly enhance surface depiction by locally compressing patterns of reflected lighting. Based on a real-time principal direction estimation algorithm, Kim et al. [2008] presented an efficient method of generating line strokes for dynamic and specular 3D models. The strokes are estimated, propagated and rendered at each pixel. Compared to the object-space techniques, image-space approaches are usually efficient and easy to implement by avoiding the complicated computation. However, they often suffer from pixel-level artifacts and may not be as effective as the object-space techniques, since they do not take advantage of the geometric features of the 3D model.

A few hybrid methods combine object- and image-space attributes. Vergne et al. [2008] introduced the apparent relief, which extracts convexity information in the object-space and curvedness information from normal variations in the image-space, and combines both sources in a single shape descriptor. Zhang et al. [2012] generalized the difference-of-Gaussian (DoG) edge detector to triangle meshes. In the object space, their algorithm smoothes the vertex normal twice with different Gaussian kernel sizes. Given a viewing direction, two diffuse shading images are generated by using the two smoothed normal fields. Then DoG lines are rendered in the image space by normalizing the difference of the two diffuse shading images. As the lines are generated in a single pass without iterating the mesh edges/faces, DoG lines are more efficient than the other object-space lines. However, DoG lines suffer from serious artifacts on large triangles, which are very common for the models created by artists. Our algorithm adopts the difference-of-Gaussian operator in a different manner. Converting the input mesh into a dense point cloud, our method generates the diffuse shading images from the point cloud without connectivity. Thus, splatting lines are insensitive to the mesh tessellation. Furthermore, our method does not require any pre-computing, while DoG lines do, which makes our method suitable for dynamic models. The detailed comparisons and discussions are presented in Section 5.

3 Splatting Lines from Point Clouds

Let $\mathcal{P} = \{\mathbf{p}_i; \mathbf{n}_i; r_i\}_{i=1,\cdots,N}$ denote a point cloud, which represents a 3D surface $\mathcal{S}$. Each point $\mathbf{p}_i \in \mathbb{R}^3$ is associated with an oriented disc, characterized by the normal vector $\mathbf{n}_i$ and the radius r_i. Assume $\mathcal{P}$ is densely sampled so that all discs cover the surface $\mathcal{S}$ without holes.

Taking $\mathcal{P}$ as input, the splatting line algorithm generates line drawings in four steps, as demonstrated in Figure 2.

1. Generate a depth image under the current view using *depth splatting*.
2. Render a diffuse shading image $\mathcal{I}_1$ using *illumination splatting*.
3. Render another diffuse shading image $\mathcal{I}_2$ by applying *illumination splatting* again. $\mathcal{I}_2$ is more blurry than $\mathcal{I}_1$.
4. Generate the line drawing by computing the difference between $\mathcal{I}_1$ and $\mathcal{I}_2$.

3.1 Depth Splatting

The first step is to create a depth image using splatting, which is called depth splatting. The depth image is used to perform visibility test in the following illumination splatting steps. For each point $\{\mathbf{p}_i; \mathbf{n}_i; r_i\}$, we draw a point primitive at $\mathbf{p}_i$, and use the geometry shader to generate a square of side length $2r_i$ centered at $\mathbf{p}_i$ and oriented the normal direction $\mathbf{n}_i$. The square is then shifted along the viewing direction with a small offset to avoid incorrect occlusion of overlapping discs. In the fragment shader, all pixels with distance to the square center greater than r_i are culled, which cuts

(a) Input model (b) Depth image (c) $\mathcal{I}_1$ (d) $\mathcal{I}_2$ (e) Splatting lines

Figure 2: *The splatting line algorithm consists of four steps. Given a densely sampled point cloud (a) as input, it first generates a depth image (b) by depth splatting, and then renders two diffuse shading images (c) & (d) by illumination splatting. The splatting lines (e) are obtained by applying the difference-of-Gaussian operator to $\mathcal{I}_2$ and $\mathcal{I}_1$.*

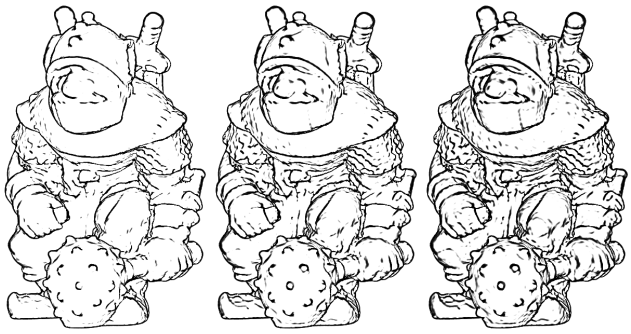
the square into a disc. The obtained depth image stores the depth value of each resulting pixel.

3.2 Illumination Splatting

After obtaining the depth image, we render two diffuse shading images using splatting. For each point $\{\mathbf{p}_i; \mathbf{n}_i; r_i\}$, we compute the diffuse shading f_i by using a single point light source located at the view point $\mathbf{e}$,

$$f_i = \mathbf{n}_i \cdot \frac{\mathbf{e} - \mathbf{p}_i}{\|\mathbf{e} - \mathbf{p}_i\|}.$$

This illumination model has proven effective for extracting feature in PELs [Xie et al. 2007], Laplacian lines [Zhang et al. 2009] and DoG lines [Zhang et al. 2012].



$\tau = 0.980$ $\tau = 0.990$ $\tau = 0.995$

Figure 3: *The parameter τ is like a thresholding which controls the number of splatting lines. The bigger τ, the more lines generated, also the result tends to be more noisy.*

Consider an arbitrary point $\mathbf{q} \in \mathcal{S}$ on the surface $\mathcal{S}$. The illumination at $\mathbf{q}$ is computed by linear interpolation

$$f(\mathbf{q}) = \frac{\sum_i w_i(\mathbf{q}) f_i}{\sum_i w_i(\mathbf{q})}, \tag{1}$$

where the blending weight $w_i(\cdot)$ is a Gaussian function

$$w_i(\mathbf{x}) = \frac{1}{2\pi r_i^2 \sigma^2} e^{-\frac{(\mathbf{x}-\mathbf{p}_i)^2}{2r_i^2\sigma^2}}. \tag{2}$$

The parameter σ controls the blurry effect in the diffuse shading image, i.e., the larger σ, the more blurry image we obtain. Applying the illumination splatting with two different σ values denoted by $\sigma_{\min}$ and $\sigma_{\max}$, we obtain two diffuse shading images $\mathcal{I}_1$ and $\mathcal{I}_2$. We observe that the default value $\sigma_{\min} = 0.4$ leads to visually pleasing results. The user can also change $\sigma_{\min}$ at run time. We set $\sigma_{\max} = 1.6\sigma_{\min}$ as suggested in the Difference-of-Gaussian edge detector.

In order to implement Equation (1) on the GPU, we draw an oriented disc for each point $\{\mathbf{p}_i; \mathbf{n}_i; r_i\}$. For each pixel $\mathbf{q}$ generated by rasterization, the scaled light intensity $w_i(\mathbf{q})f_i$ and the blending weight $w_i(\mathbf{q})$ are sent to the color and alpha channels of the frame buffer, respectively. The summations in the numerator and denominator of Equation (1) are implemented by frame buffer alpha blending. The normalization of $f(\mathbf{q})$ is performed in a full-screen post processing pass that divides $\mathbf{q}$'s color by its alpha value.

3.3 Generating Line Drawings

The above splatting step produces two diffuse shading images I_1 and I_2. We compute the difference between I_1 and I_2, $H = I_1 - \tau I_2$, and then generate the splatting lines by scaling H, $E = \tanh(\phi H)$. τ is a "thresholding" parameter that is closely related to the total number of lines. Usually, a big τ leads to more splatting lines, which also tend to be more noisy (see Figure 3). The parameter ϕ controls the line strength. The bigger ϕ, the stronger liens we obtained (see Figure 4). The user can also control the splat size for fine tuning the splatting lines. We use a factor λ to uniformly scale all the splat radii r_i. Intuitively speaking, the scaling factor λ controls the "level-of-details" of the line drawing (see Figure 5). All these parameters τ, ϕ and λ can be dynamically adjusted by the user at runtime.

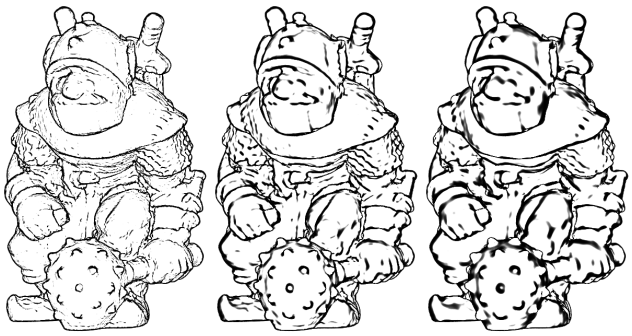
$\phi = 50$ $\phi = 200$

Figure 4: *The parameter ϕ controls the strength of the splatting lines. The bigger ϕ, the stronger lines obtained.*

As Figure 6 (left) shows, when large radii are used in splatting, artifacts may appear near the inner and exterior contours of the model. This is because some splat discs are beyond the object contour. To address this issue, we do not scale the splat radii when performing depth splatting; this results in a more accurate depth image. When performing illumination splatting, the false pixels beyond the object contour are culled in a depth test. As Figure 6 (right) shows, this simple scheme is very effective for improving line drawings.

4 Splatting Lines from Polygonal Meshes

It is very natural to extend the above point-based splatting lines to polygonal meshes. Let M be the input polygonal mesh. Our algorithm converts M into a tessellation-independent point cloud. Although white noise can be generated in a highly efficient manner [Osada et al. 2002], the samples (hence the splats) are not dis-



$\lambda = 1$ $\lambda = 3$ $\lambda = 5$

Figure 5: *The parameter λ controls the splat sizes, which in turn controls the 'level-of-details' of the resulting splatting lines. Increasing or decreasing λ reveals coarser or finer shape cues respectively.*

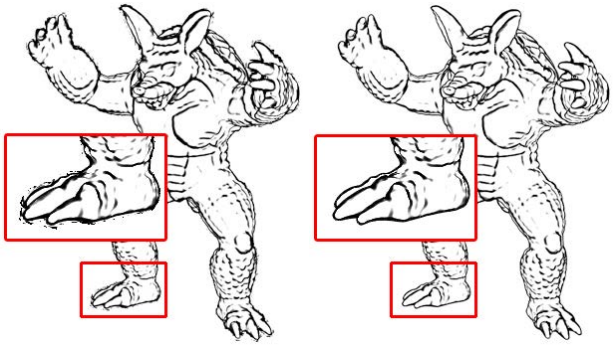


Figure 6: *The artifacts near the contours (left) can be reduced dramatically by our simple-yet-effective scheme (right).*

tributed uniformly, which produces non-smooth splatting lines (see Figure 7 left). To obtain a high-quality line drawing, we adopt the blue noise sampling [Ying et al. 2013], which distributes all samples uniformly and randomly on the surface. The algorithm is intrinsic, parallel and accurate, which enables it to generate bias-free blue noise samples interactively on commonly used models. We approximate the normal $\mathbf{n}_i$ of the blue noise sample $\mathbf{p}_i$ by Barycentric interpolation. The splat radius should be large enough to avoid any "hole" in the resulting diffuse shading images. We estimate a minimal splat radius by $r_{\min} = A/n$, where A is the total surface area of the mesh, and n is the number of samples. Then we determine the splat radius as $r = \lambda r_{\min}$, where λ is the scaling factor as described in Section 3. Thanks to its excellent spatial properties, the blue noise sampling generates smooth and visually pleasing splatting lines (see Figure 7 right).

5 Splatting Lines from Volumes

Our algorithm can also generate line drawings from a volume model, which consist of a 3D array of voxels. Each voxel has a scalar voxel value f. We can also compute a normal vector $\mathbf{n}$ for each voxel by computing the gradient of voxel values $\mathbf{n} = -\bigtriangledown f/||\bigtriangledown f||$. Similar to previous approaches [Burns et al. 2005; Zhang et al. 2011], our method takes a user-specified iso-value f_0 as input, and generates line drawings for conveying the corresponding iso-surface $\mathcal{S}_{f_0}$ (see Figure 8).

5.1 Sampling the iso-surface

To generate point samples on the iso-surface, a straightforward approach is to generate a single sample in the center of each cube whose voxel values cover the user-specified iso-value f_0, i.e., $f_0 \in [f_{\min}, f_{\max}]$, where $f_{\min}$ and $f_{\max}$ are the minimal and maximal

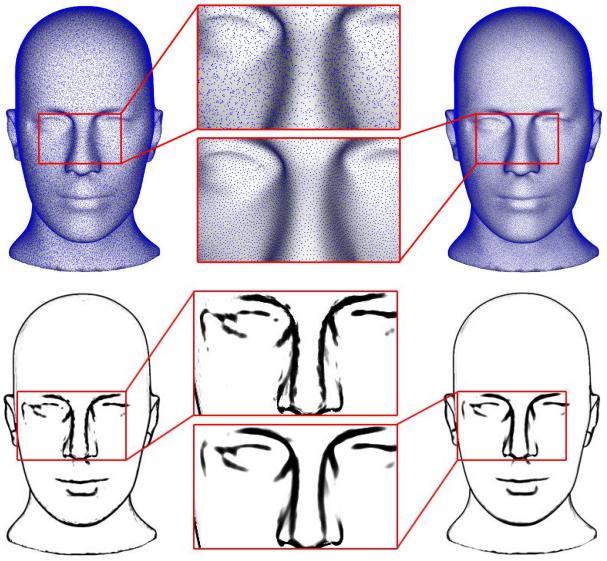
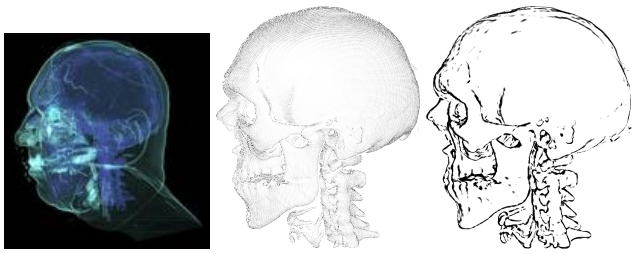


Figure 7: *120K white noise samples (left) and blue noise samples (right) on the mannequin model. Thanks to its nice spatial distribution, the blue noise sampling produces smoother splatting lines than the white noise sampling.*



(a) (b) (c)

Figure 8: *The splatting line algorithm for volume models. Given a volume model (a) and the user-specified iso-value f_0 as input, we first generate a set of point samples on the iso-surface $\mathcal{S}_{f_0}$, and then draw splatting lines (c) using the point-based method described in Section 3.*

values of eight voxels of the cube. Unfortunately, many samples produced by this method are not on the iso-surface, leading to various artifacts in the resulting line drawing as shown in Figure 10 (a).

We propose a simple-yet effective approach to sample the iso-surface. Similar to the näive approach, we generate a single sample for each cube C_i whose voxel values cover the current iso-value f_0. We check for each edge $e_j \in C_i$ of the cube C_i whether e_j intersects the iso-surface $\mathcal{S}_{f_0}$. Let $\mathbf{p}_1$ and $\mathbf{p}_2$ be the two endpoints of the edge e_j, and f_1, f_2 are their voxel values. If $(f_1 - f_0)(f_2 - f_0) < 0$, the intersection is given by $(1-\alpha)\mathbf{p}_1 + \alpha\mathbf{p}_2$, where $\alpha = (f_1 - f_0)/(f_1 - f_2)$. If $f_1 = f_2 = f_0$, we set $\alpha = 0.5$. Then we compute the average of all intersection points in C_i, denoted by $\overline{\mathbf{p}}$, and place a sample point at $\overline{\mathbf{p}}$. The normal of the point sample $\overline{\mathbf{p}}$ is approximated by trilinear interpolation of the normal vectors of eight voxels of C_i.

Despite its simplicity, our method works very well in practice. As Figure 9 shows, for a cube containing one polygonal facet, the point sample generated using our method is the centroid of the polygon. For a cube containing multiple separated facets, the point sample generated using our method is the average of the centroids of all facets. The latter scenario is rare in our experimentation. There-

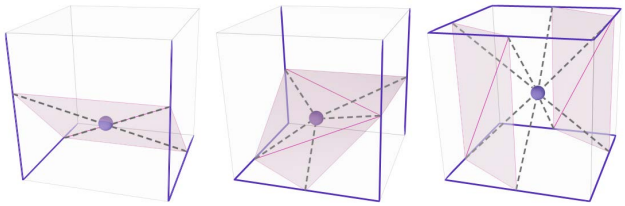
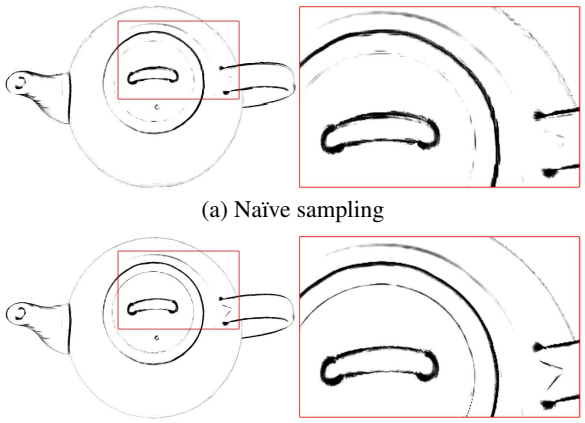


Figure 9: *Sampling the iso-surface. The shaded polygons represent the part of iso-surface within the cube and the dots are the samples generated by our method. The cube edges containing the intersection points are colored in blue. If the iso-surface $\mathcal{S}_{f_0}$ contains only one polygonal facet in the cube, the sample $\overline{\mathbf{p}}$ is indeed the centroid of the polygon; otherwise, $\overline{\mathbf{p}}$ is the average of the centroids of all facets.*

fore, our method generates accurate samples for most of the cubes, and the errors in those cubes containing multiple facets can be negligible. Figure 10 compares the line drawings based on the naïve sampling method and our method.



(a) Naïve sampling

(b) Our approach

Figure 10: *Comparison of line drawings using naïve sampling and our sampling scheme.*

Our sampling scheme can be fully implemented on the GPU. For each cube, we draw a point, resulting in a pixel by rasterization. The computation of the position and normal of the point sample in the cube is performed in the fragment shader. The result is packed in a 2D frame buffer and later copied to an OpenGL vertex buffer object (VBO). The samples stored in the vertex buffer object are then used to generate line drawings using the splatting lines algorithm described in Section 3. An alternative solution is to use the marching cubes algorithm [Lorensen and Cline 1987; Jose 2010] to extract the iso-surface in each cube and then draw samples on that iso-surface. Our approach is easy to implement and efficient, which leads to reasonably good results. In our experiments, it takes less than 50 ms to generate point samples for volume models with resolution up to $384 \times 384 \times 240$ (34M voxels).

5.2 Visibility

Given an iso-value f_0, it is very common that the corresponding iso-surface $\mathcal{S}_{f_0}$ has multiple connected components so that some components are partially or completely occluded by the other. As a result, the occlusion issue in the volume line drawings is more complicated than the surface counterpart. To convey both the exterior and the interior of the object, we adopt the depth peeling to draw splatting lines on the iso-surface $\mathcal{S}_{f_0}$.

Given a viewing direction, assume the iso-surface $\mathcal{S}_{f_0}$ has K layers, D_i, $i = 1, \cdots, K$, where D_1 is the front most (i.e., visible) layer. For the i-th layer, we generate a depth image $\mathcal{D}_i$ by using depth peeling based on the depth image of the previous layer $\mathcal{D}_{i-1}$. When performing illumination splatting, we apply a customized depth test that removes all splats whose depth values differ from that stored in the same position of the depth image $\mathcal{D}_i$. Such a depth test is performed in the geometry shader on a per-splat basis.

Obviously, the cost of drawing splatting lines on multiple layers is linearly proportional to the number of layers. In our experiments, we have observed that the iso-surfaces of most real-world models contain no more than three layers. An advantage of our method is that it can tell which line is on which layer, instead of just a Boolean value indicating that it is visible or invisible. As a result, we can distinguish lines on different levels using different drawing styles (such as colors, see Figure 11), which helps revealing the spatial structure of the volume model.

6 Experimental Results

We have implemented our splatting line algorithm in C++ and tested it on a low-end PC with an Intel i3 CPU, 2GB memory, an NVIDIA GTX650Ti graphics card with 1GB video memory. The screen resolution is of 1024×768.

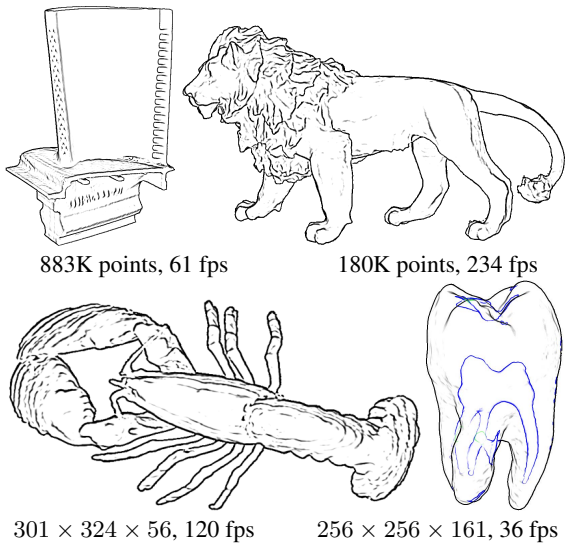
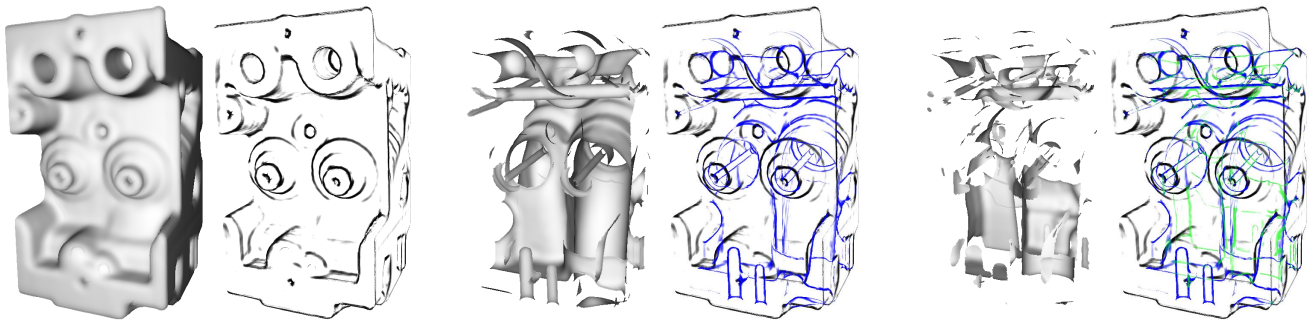


Figure 12: *Splatting lines from point models and volume models.*

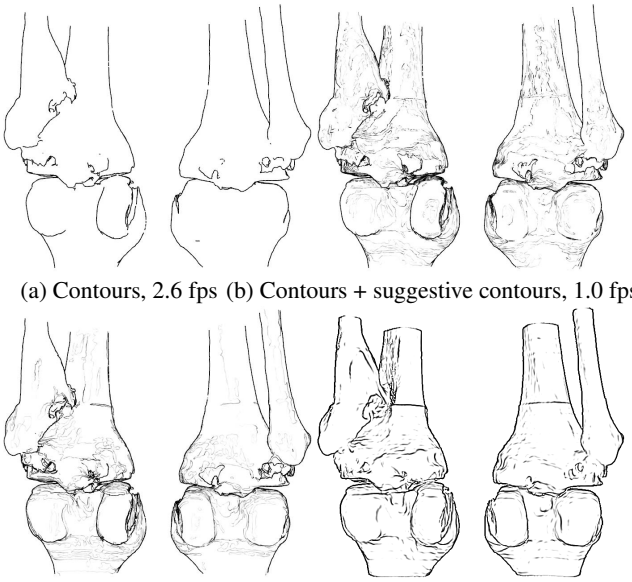
Figure 12 shows splatting lines from point models and volumes. The point models are generated using the QSplat software and the volume models are courtesy of volvis.org. As Figure 12 shows, the proposed splatting lines can depict most of the salient features of the models and convey the shape effectively. Figure 13 compares splatting lines with contours, suggestive contours and Laplacian lines on the Knee model. Due to the noise in the volume data, the other techniques produce many short and broken lines. Since our algorithm performs like a difference-of-Gaussian operator, the generated splatting lines are smoother than the other approaches.

We also tested our method on a wide range of 3D meshes, including both the scanned models and the artists-created models. The former usually contains various types of defects, such as noise, degenerate triangles, non-manifold configuration, etc, while the latter may contain skinny triangles and high anisotropy. Figure 14 compares splatting lines with other popular line drawings on artists-made meshes, which usually have skinny and/or large triangles. We observe that all the object-space algorithms produce line drawings with various artifacts such as noises, short and zigzag lines, since the computation of geometric properties in these methods heavily depends on



One layer Two layers Three layers

Figure 11: *Conveying the interior of the volume model by incorporating the depth peeling with splatting lines. The shaded images show the corresponding iso-surfaces. The splatting lines of different layers are drawn in different colors.*



(a) Contours, 2.6 fps (b) Contours + suggestive contours, 1.0 fps

(c) Contours + Laplacian lines, 1.2 fps (d) Splatting lines, 46 fps

Figure 13: *Comparison to other line drawings on the Knee model (*$379 \times 229 \times 305$*).*

the mesh tessellation. The DoG line is also poor on large triangles, since it interpolates some function defined on the vertex. Thanks to the point based nature, our method is robust to the mesh tessellation and generates smooth line drawings on these models.

Thanks to its good performance, our method is well suited for generating line drawing animations from deformable models (see Figure 15). We sampled only the first frame: for each sample we record the face to which it belongs and its Barycentric coordinates. Then we computed the new location of each sample in the subsequent frames by using the Barycentric interpolation. Thanks to the smooth nature of splatting lines, we have observed that the animation generated using splatting lines are temporally coherent. Refer to the accompanying video demo for further details. Some advanced sampling algorithm (e.g.,[Wand and Straßer 2002]) can certainly lead to a better sample distribution, thus, improves the quality of splatting lines.

Table 1 lists the performance of splatting lines from point clouds and volumes. We have observed that the bottleneck of the splatting line algorithm lies in pixel processing. The algorithm must perform splatting three times (one for depth splatting and two for illumination splatting) and in each splatting step, multiple pixels may be generated for each point of the input model. Accordingly, the performance of the splatting line depends on three factors - the number of points, the scaling factor of splat sizes (i.e., λ), and the screen resolution - the latter two of which determine how many pixels are generated for each point. Our approach achieves real-time frame rates for point models with millions of points. The cost of drawing splatting lines from volumes is similar to that from the point models. The main difference lies in that drawing splatting lines from volume models has an extra step for generating samples on the iso-surface. However, this step is only necessary when the user changes the iso-value. Also, the cost of this extra step is usually negligible compared to that of splatting.

Figure 15: *Splatting line animation generated from a deformable model (5K triangles). 200K samples are used in each frame.*

Table 1: *The rendering speed (in frame-per-second) of splatting lines in terms of the parameter* λ*.*

Model	size	$\lambda = 1$	$\lambda = 3$	$\lambda = 6$
Bunny	32K	458	127	40.3
Lion	179K	348	120	40.2
Buddha	1.01M	109	77	37.3
Dragon	1.22M	86.2	46.5	18
Lucy	9.6M	14.3	13.5	9.5
Lobster	$301 \times 324 \times 56$	380	117	38.3
Engine	$256 \times 256 \times 110$	209	59	18.8
Teapot	$256 \times 256 \times 178$	183	46	13.8
Bonsai	$256 \times 256 \times 256$	360	161	46
Carp	$256 \times 256 \times 512$	220	75	24.3
Chest	$384 \times 384 \times 240$	151	77.2	29.6

The rendering time of splatting lines on polygonal meshes is irrelevant to the input mesh complexity. Instead, it depends on the number of samples. In our experiments, the typical number of samples n ranges from 100K to 1M for the test models. We have observed that if the number of samples is very large (say $n \geq$ 500K), there is little visual difference of the resulting splatting lines between the white noise sampling and blue noise sampling. Therefore, we use blue noise sampling for $n <$ 500K and white noise sampling for $n \geq$ 500K. The parallel sampling algorithm [Ying et al. 2011] takes 0.1 to 0.5 seconds, depending on the mesh complexity. The white noise generation algorithm [Osada et al. 2002] can generate 1M white noise in 0.1 seconds. As the sampling is performed only once

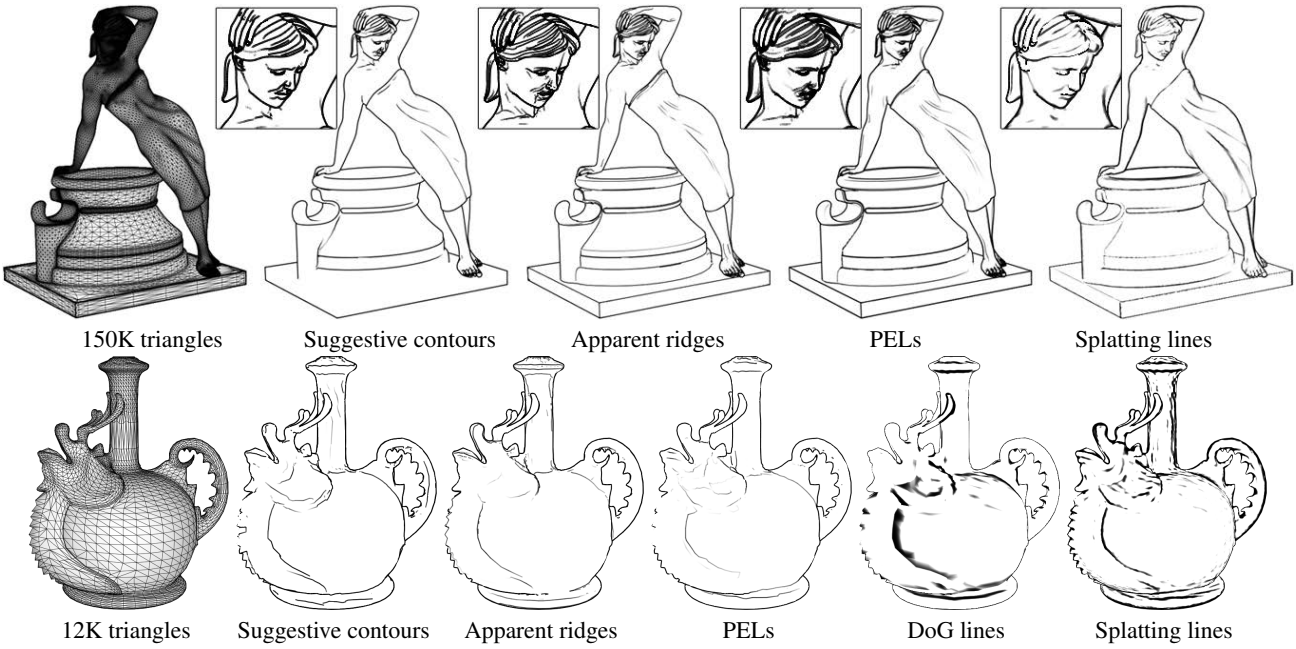


Figure 14: *Comparison to object-space algorithms on artist-created meshes. While all methods produce fairly good results on the smooth regions (e.g., the cloth in the Girl model), the object space approaches are not able to effective convey the fine details which contain large number of anisotropic triangles. This is because the existing approaches usually utilize the local differential operator to calculate the feature lines. The close-up views reveal the artifacts on the head of the Girl model. In contrast, our splatting lines are insensitive to the tessellation and produce visually pleasing results.*

when the mesh is loaded, the time for blue/white noise generation can be ignored. Figure 16 shows the rendering speed of splatting lines under different parameters. Table 2 compares the performance of splatting lines with other popular feature lines.

Table 2: *Mesh complexity and performance. The 3rd to 7th columns are the fps for suggestive contours (SC), apparent ridges (AR), photic extremum lines (PEL), Laplacian lines (LL) and our splatting lines (SL), respectively.*

Model	# faces	SC	AR	PEL	LL	SL
Grog (Fig. 3)	1.8M	4.6	1.3	1.3	4.8	23
Dragon (Fig. 4)	1.0M	8.9	2.2	3.0	9.2	24
Landscape	1.5M	6.3	1.6	2.1	6.3	18
Nicolo (Fig. 1)	1.8M	4.7	1.5	1.8	5.2	24

Our method has two limitations. Firstly, unlike the object-space algorithms, which extract the feature lines, our method renders the splatting lines in the image space, which makes it difficult to stylize splatting lines. Secondly, our method requires fairly dense sampling on the input polygonal mesh, even for a mesh with simple geometry and low resolution. For such models, the splatting line algorithm is slower than other object or image space algorithms. Also, our method requires that the point samples are uniformly distributed. While the blue noise sampling for polygonal meshes and the proposed iso-surface sampling for volumes can guarantee this property, our method cannot be directly applied to raw point clouds obtained by a range camera.

7 Conclusions

We have presented splatting lines, a uniform algorithm framework for generating high-quality line drawings from various types of 3D models, including point models, volumes, and static and dynamic deformable polygonal meshes. We first introduced splatting lines for point models. We applied point splatting twice with different parameters, resulting in two diffuse shading images, and then rendered splatting lines by processing the two images. In order to generate splatting lines from the volume data, we proposed a simple-yet-efficient method to sample the iso-surface. We also showed how to incorporate depth peeling with splatting lines to depict the inner structure of volumes. For polygonal meshes, we employed white-noise and blue-noise sampling methods to generate a set of dense point samples on the surface and then rendered splatting lines from the samples. The experimental results demonstrated the advantages of splatting lines, including high performance, robust to mesh tessellation, coherent results and avoiding the need for preprocessing.

Acknowledgments Ying He is partially supported by an MOE Tier1 grant RG40/12. Long Zhang is partially supported by NSFC 60903085 and NSFC 61170150. This research was done for Fraunhofer IDM@NTU, which is funded by the National Research Foundation (NRF) and managed through the multi-agency Interactive & Digital Media Programme Office (IDMPO) hosted by the Media Development Authority of Singapore (MDA). We thank the anonymous reviewers for their constructive comments. The Armadillo, Bunny, Dragon, and Lion models are courtesy of Stanford University, the volume datasets are courtesy of volvis.org, and all the other models are courtesy of Aim@Shape Project.

References

BUCHANAN, J. W., AND SOUSA, M. C. 2000. The edge buffer: a data structure for easy silhouette rendering. In *NPAR '00*, 39–42.

BURNS, M., KLAWE, J., RUSINKIEWICZ, S., FINKELSTEIN, A., AND DECARLO, D. 2005. Line drawings from volume data. In *SIGGRAPH '05*, 512–518.

DANIELS, J., HA, L. K., OCHOTTA, T., AND SILVA, C. T. 2007. Robust smooth feature extraction from point clouds. In *IEEE SMI'07*, 123–136.

DANIELS II, J., OCHOTTA, T., HA, L. K., AND SILVA, C. T. 2008. Spline-based feature curves from point-sampled geometry. *The Visual Computer 24*, 6, 449–462.

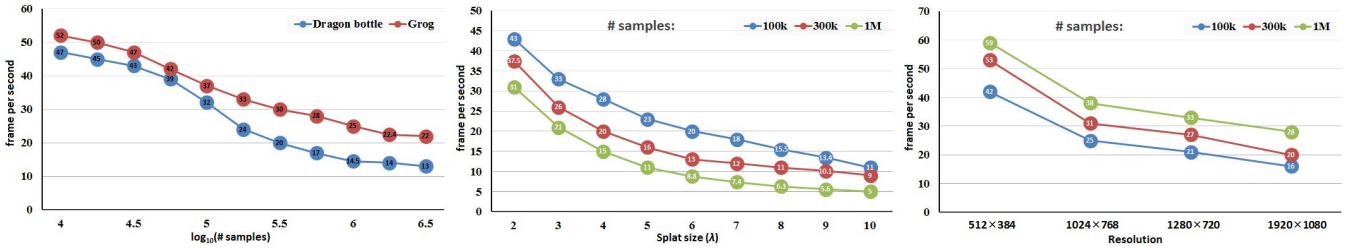


Figure 16: *Performance. Left: fps vs n with fixed splat size $\lambda = 2$ and image resolution 1024×768 for Grog and Bottle. Middle: fps vs splat size with fixed image resolution 1024×768 for Nicolo. Right: fps vs image resolution with fixed splat size $\lambda = 2$ for Nicolo.*

DeCarlo, D., and Rusinkiewicz, S. 2007. Highlight lines for conveying shape. In *NPAR '07*, 63–70.

DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., and Santella, A. 2003. Suggestive contours for conveying shape. *ACM Transactions on Graphics 22*, 848–855.

Jose, S. 2010. Gpu-accelerated data expansion for the marching cubes algorithm. In *GPU Technology Conference 2010*.

Judd, T., Durand, F., and Adelson, E. H. 2007. Apparent ridges for line drawing. *ACM TOG 26*, 3, Article No 19.

Kang, H., Lee, S., and Chui, C. K. 2007. Coherent line drawing. In *NPAR '07*, 43–50.

Kim, Y., Yu, J., Yu, X., and Lee, S. 2008. Line-art illustration of dynamic and specular surfaces. *ACM TOG 27*, 5.

Kolomenkin, M., Shimshoni, I., and Tal, A. 2008. Demarcating curves for shape illustration. *ACM TOG 27*, 5.

Kolomenkin, M., Shimshoni, I., and Tal, A. 2009. On edge detection on surfaces. In *CVPR'09*, 2767–2774.

Kyprianidis, J. E., and Kang, H. 2011. Image and video abstraction by coherence-enhancing filtering. *Computer Graphics Forum 30*, 2, 593–602.

Lee, Y., Markosian, L., Lee, S., and Hughes, J. F. 2007. Line drawings via abstracted shading. *ACM Trans. Graph. 26*, 3.

Linsen, L., Müller, K., and Rosenthal, P. 2007. Splat-based ray tracing of point clouds. In *WSCG '07*.

Lorensen, W. E., and Cline, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87*, vol. 21, 163–169.

Ohtake, Y., Belyaev, A., and Seidel, H. 2004. Ridge-valley lines on meshes via implicit surface fitting. In *Proceedings of ACM SIGGRAPH '04*, 609–612.

Osada, R., Funkhouser, T., Chazelle, B., and Dobkin, D. 2002. Shape distributions. *ACM TOG 21*, 4, 807–832.

Pauly, M., Keiser, R., and Gross, M. 2003. Multi-scale feature extraction on point-sampled surfaces. In *Computer Graphics Forum*, vol. 22, 281–289.

Raskar, R., Tan, K.-H., Feris, R., Yu, J., and Turk, M. 2005. Non-photorealistic camera: depth edge detection and stylized rendering using multi-flash imaging. *ACM Transactions on Graphics 23*, 3, 679–688.

Saito, T., and Takahashi, T. 1990. Comprehensible rendering of 3-D shapes. In *SIGGRAPH '90*, 197–206.

Sun, Q., Zhang, L., and He, Y. 2013. Splatting lines for 3D mesh illustration. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '13)*, 193.

Thirion, J.-P., and Gourdon, A. 1996. The 3D marching lines algorithm. *Graphical Models and Image Processing 58*, 6, 503–509.

Vergne, R., Barla, P., Granier, X., and Schlick, C. 2008. Apparent relief: a shape descriptor for stylized shading. In *NPAR '08*, 23–29.

Vergne, R., Pacanowski, R., Barla, P., Granier, X., and Schlick, C. 2009. Light warping for enhanced surface depiction. *ACM Transactions on Graphics 28*, 3, 25:1–25:8.

Wand, M., and Strasser, W. 2002. Multi-resolution rendering of complex animated scenes. *CGF 21*, 3, 483–491.

Whelan, J. C., and Visvalingam, M. 2003. Formulated silhouettes for sketching terrain. In *Proceedings of Theory and Practice of Computer Graphics 2003*, 90–96.

Winnemöller, H., Olsen, S. C., and Gooch, B. 2006. Real-time video abstraction. *ACM TOG 25*, 3, 1221–1226.

Xie, X., He, Y., Tian, F., Seah, H. S., Gu, X., and Qin, H. 2007. An effective illustrative visualization framework based on photic extremum lines (PELs). *IEEE Transactions on Visualization and Computer Graphics 13*, 6, 1328–1335.

Xu, H., Nguyen, M. X., Yuan, X., and Chen, B. 2004. Interactive silhouette rendering for point-based models. In *PBG '04*, 13–18.

Ying, X., Xin, S.-Q., Sun, Q., and He, Y. 2011. Parallel and accurate poisson disk sampling on arbitrary surfaces. In *SIGGRAPH Asia 2011 Sketches*, 18:1–18:2.

Ying, X., Xin, S.-Q., Sun, Q., and He, Y. 2013. An intrinsic algorithm for parallel poisson disk sampling on arbitrary surfaces. *IEEE Transactions on Visualization and Computer Graphics 19*, 9, 1425–1437.

Zakaria, N., and Seidel, H.-P. 2004. Interactive stylized silhouette for point-sampled geometry. In *Proceedings of GRAPHITE '04*, 242–249.

Zhang, L., He, Y., Xie, X., and Chen, W. 2009. Laplacian lines for real-time shape illustration. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games (I3D '09)*, 129–136.

Zhang, L., He, Y., Xia, J., Xie, X., and Chen, W. 2011. Real-time shape illustration using Laplacian lines. *IEEE Transactions on Visualization and Computer Graphics 17*, 7, 993–1006.

Zhang, L., Xia, J., Ying, X., He, Y., Mueller-Wittig, W., and Seah, H. S. 2012. Efficient and robust 3D line drawings using difference-of-Gaussian. *Graphical Models 74*, 4, 87–98.

Zwicker, M., Pfister, H., van Baar, J., and Gross, M. 2001. Surface splatting. In *SIGGRAPH '01*, 371–378.