CGI 2008 Conference Proceedings manuscript No.
(will be inserted by the editor)

Paul Rosenthal · Lars Linsen

Image-space Point Cloud Rendering

Abstract Point-based rendering approaches have gained a major interest in recent years, basically replacing global surface reconstruction with local surface estimations using, for example, splats or implicit functions. Crucial to their performance in terms of rendering quality and speed is the representation of the local surface patches. We present a novel approach that goes back to the original ideas of Grossman and Dally to avoid any object-space operations and compute high-quality renderings by only applying image-space operations.

Starting from a point cloud including normals, we render the lit point cloud to a texture with color, depth, and normal information. Subsequently, we apply several filter operations. In a first step, we use a mask to fill background pixels with the color and normal of the adjacent pixel with smallest depth. The mask assures that only the desired pixels are filled. Similarly, in a second pass, we fill the pixels that display occluded surface parts. The resulting piecewise constant surface representation does not exhibit holes anymore and is smoothed by a standard smoothing filter in a third step. The same three steps can also be applied to the depth channel and the normal map such that a subsequent edge detection and curvature filtering leads to a texture that exhibits silhouettes and feature lines. Anti-aliasing along the silhouettes and feature lines can be obtained by blending the textures. When highlighting the silhouette and feature lines during blending, one obtains illustrative renderings of the 3D objects. The GPU implementation of our approach achieves interactive rates for point cloud renderings without any pre-computation.

Keywords Point Cloud Rendering · GPU Rendering · Image-space Rendering

Paul Rosenthal
E-mail: p.rosenthal@jacobs-university.de

Lars Linsen
E-mail: l.linsen@jacobs-university.de

Jacobs University
School of Engineering and Science
Campus Ring 1
28759 Bremen
Germany
Tel.: +49-421-200-3188
Fax: +49-421-200-3103

1 Introduction

Surface reconstruction from point clouds has been a vivid area of research since 3D scanning devices emerged. Traditional approaches generate triangular meshes that connect the points to form a piecewise linear approximation of the sampled surface. Geometrically and topologically, the generated mesh has to represent a manifold, typically a closed manifold.

As the scanning devices improved, the number of samples increased significantly. Since the existing triangular mesh generation algorithms scaled superlinearly in the number of points, the desire arose to replace the global surface reconstruction with some local operations. It was the time when point-based rendering approaches came up. The idea was to render the points directly without generating global connectivity information.

Nowadays, different point-based rendering approaches exist. The most prominent ones are based on locally approximating the surface by fitting planes or polynomial surfaces to a subset of neighboring points. Technically, these are still (local) surface reconstructions. We propose an approach that gets back to the original idea of directly rendering points instead of surface parts surrounding the points. We do not perform any (pre-)computations in object space. Instead, all our processing steps are carried out in image space.

The first step of our processing pipeline, shown in Figure 1, is to project all points with normals of the (illuminated) point cloud to the screen (or image) space, see Section 3. In a second and a third step, the holes in the projected surface need to be filled. The second step fills holes that incorrectly exhibit background information (see Section 4), while the third step fills holes

that exhibit occluded (or hidden) surface parts (see Section 5). The fourth step smoothes the output to generate smoothly varying surface color shades, see Section 6.

Optionally, the silhouettes of the generated image can be anti-aliased in a post-processing step, see Section 7. The processing pipeline also enables the opportunity to perform certain illustrative rendering techniques like silhouette rendering and feature-line extraction, see Section 8.

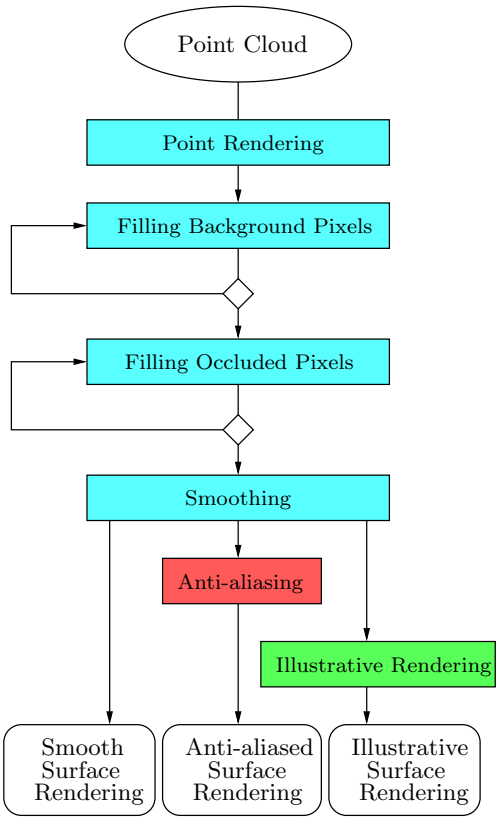

Fig. 1 Process pipeline for image-space point cloud rendering.

Since all operations are performed in image space, they are all implemented to operate on the GPU. Exploiting the capacities of modern GPUs in terms of speed and parallelism, we are capable of displaying point clouds with large numbers of points at interactive rates. Results are presented in Section 9 and discussed in Section 10.

2 Related Work

Point cloud representation of surfaces has gained increasing attention over the last decade. One milestone was the Michelangelo project [5], where huge amounts of surface points have been generated for the first time. As the number of points is steadily increasing, generating global triangular meshes is often unfeasible because of exploding computation times. Therefore, several local or approximating methods were proposed [3].

Alternatively, several efforts have been made to directly operate on point cloud surface representations. One direction of research is to use some kind of surface elements such as splats or triangle fans [6,7,9,11,16,17]. Although these approaches are fast, they require precomputations to generate local neighborhoods. The quality of the surface representation depends heavily on the chosen neighborhood computation and typically there is a trade-off between quality and speed.

A second direction in this field is the point set surface approach [1] and its follow-up papers, where an implicit surface is locally fit to the data. These approaches involve an iteration and are, generally, slower than the approaches using surface elements. For an overview over ongoing research in the field of point-based computer graphics, we refer to the tutorial by Gross et al. [2] or the article by Sainz et al. [12].

The above-mentioned approaches can mostly be regarded as local surface reconstruction algorithms. However, with growing number of surface points, sizes of rendering primitives nearly reduce to pixel size, and surface approximation algorithms should be transferred to image space. This was recently done by Marroquim et al. [8] and Schnabel et al. [14], who adapted splatting to image space using hierarchical framebuffers.

This idea of using only surface points as rendering primitives goes back to Grossman and Dally [4]. Our approach for image-space operations for rendering point cloud surfaces also builds upon this idea, but in contrast to the other approaches mentioned here omits any geometric considerations and calculations of the radius of influence of a pixel.

Meanwhile, there also exist some illustrative rendering techniques for point cloud surface representations. Xu et al. [15] use image-space operations to generate silhouette renderings of point-based models using small discs in image space.

3 Point Rendering

The goal of the presented approach is to efficiently produce high-quality renderings of objects and scenes, whose surfaces are given in point cloud representation. More precisely, a two-dimensional oriented manifold Γ is given by a finite set $\tilde{\Gamma}$ of points on the manifold. In addition to their position, the points on the surface should also have surface normals associated with them, i. e.

$$\tilde{\Gamma} := \left\{ (\mathbf{x}_i, \mathbf{n}_i) \in \Gamma \times T_{\mathbf{x}_i}\Gamma^{\perp} \;:\; \|\mathbf{n}_i\| = 1 \right\} ,$$

where $T_{\mathbf{x}_i}\Gamma^{\perp}$ denotes the orthogonal complement to the tangent plane $T_{\mathbf{x}_i}\Gamma$ to the manifold Γ in the point $\mathbf{x}_i$.

We propose a rendering pipeline for such point clouds $\tilde{\Gamma}$ that does not require additional object-space computations such as local (or even global) geometry or topol-

ogy estimations of the manifold Γ. Instead, all processing steps are performed in image (or screen) space. Consequently, the first processing step in our pipeline is to project point cloud $\tilde{\Gamma}$ into image space, including not only the points but also generating a normal map.

Before projecting the points, they are lit using the local Phong illumination model with ambient, diffuse, and specular lighting. The illuminated points and associated normals are projected onto the screen using perspective projection.

During projection we apply backface culling and depth buffering. The backface culling is performed by discarding back-facing points $(\mathbf{x}_i, \mathbf{n}_i)$, with respect to the surface normal $\mathbf{n}_i$. The depth test is performed by turning on the OpenGL depth buffering. Consequently, if two points are projected to the same pixel, the one closer to the viewer is considered. The colors as well as the normals of the projected points are stored in RGBA color textures using the RGB channels only. Figure 2 shows the result of our first processing steps. The data set used is the well-known skeleton hand data set consisting of 327k points. The illuminated points after projection in conjunction with backface culling and depth buffering are displayed.

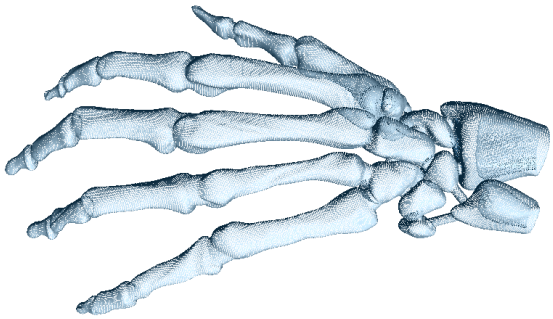

Fig. 2 Rendering of the illuminated surface points of the skeleton hand data set containing 327k surface points (Data set courtesy of Stereolithography Archive, Clemson University).

Besides color and position in image space, the depth of each projected point, i. e. the distance of the represented point $\mathbf{x}_i$ to the viewer's position $\mathbf{x}_v$, is required for our subsequent processing steps. The depth value calculated during the depth test is not linear in the distance d of the point to the viewer, as it is given by

$$f(d) := \frac{(d - z_{\text{near}})\, z_{\text{far}}}{(z_{\text{far}} - z_{\text{near}})\, d} ,$$

where z_{near} and z_{far} denote the viewer's distances to the near and far planes. Since this depth information is not suitable for our needs, we replace it by computing the depth values for each projected point by

$$f(d) := \frac{d}{z_{\text{far}}} ,$$

and storing this value at the respective position in the alpha channel of the RGBA textures.

4 Filling Background Pixels

If the sampling rate of surface Γ is high enough such that the projected distances of adjacent points of point cloud $\tilde{\Gamma}$ are all smaller or equal to the pixel size, then the projected illuminated points that pass backface culling and depth test produce the desired result of a smoothly shaded surface rendering. Obviously, this assumption does not hold in general, especially not when zooming closer to the object. As a consequence, the resulting surface rendering exhibits "holes" such that pixels that should be filled with object colors are filled with background colors, cf. Figure 2.

In a second pass, such background pixels need to be filled with the proper surface color and normal. Of course, one has to carefully choose, which background pixels are to be filled and which not. Figure 3 shows the issue and the two cases that need to be distinguished. All white pixels represent background pixels. While the pixels with a green frame are examples of holes in the surface that need to be filled, the pixels with a red frame lie beyond the silhouette (or border) of the object and should maintain their background color.

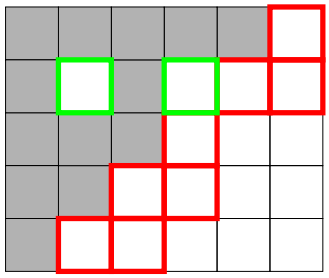

Fig. 3 Closeup view of the rendered surface's border. Pixels that have been filled in the first pass (point rendering) are shown in grey, background pixels in white. All background pixels close to a filled pixel are framed. Green frames indicate holes in the surface rendering and the respective pixels have to be filled. Red frames indicate pixels that are beyond the silhouette of the object and must not be filled.

To distinguish between background pixels that are to be filled and those that are not, we use a mask that is been applied in form of a filter using 3×3 pixels. When applying this filter, we only look at those background pixels, where some of the surrounding pixels are non-background pixels. In Figure 3, the considered pixels are the ones that are framed. To identify the ones that have a red frame, we use the eight masks shown in Figure 4, where the white pixels indicate background pixels and the dark pixels could be both background or non-background pixels. For each background pixel, we test

whether the 3×3 neighborhood of that pixel matches any of the cases. In case it does, the pixel is not filled. Otherwise, it is filled with the color, depth and normal information of the pixel with smallest depth out of the 3×3 neighborhood.

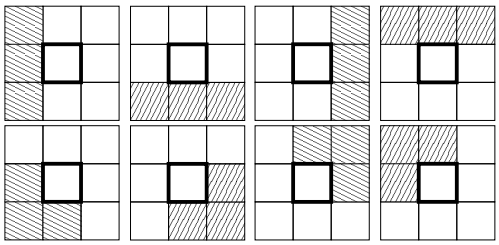

Fig. 4 Filters with size 3×3 for detecting whether a background pixel is beyond the projected silhouette of the object. If one of the eight masks matches the neighborhood of a background fragment, it is not filled. White cells indicate background pixels, dark cells may be background or non-background pixels.

The implementation of this complex test is extremely simple and can be done efficiently by a single test. Assuming that background pixels have depth zero, for each mask in Figure 4 the depth values of corresponding white pixels are summed up. If the product of all eight sums equals zero, at least one sum was zero, i. e. at least one of the eight filters detected that the observed background pixel is beyond the object's silhouette.

The process of filling background pixels using the filter is iterated, until no more background pixels need to be filled. The number of iterations depends on the point density and the viewing parameters. It is easy to see, that every hole in image space with a maximum diameter of n pixels is filled after at most n iterations.

When applied to the output of the first pass shown in Figure 2, the filling of background pixels leads to the result shown in Figure 5. Only one filter pass had to be applied to fill background pixels.

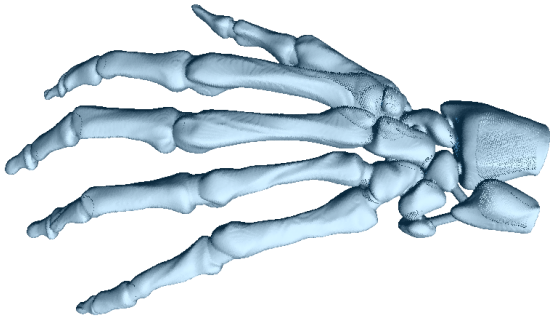

Fig. 5 Filling background pixels applied to the output of point rendering (Figure 2) of skeleton hand data set. Only one iteration of the filter had to be applied (Data set courtesy of Stereolithography Archive, Clemson University).

5 Filling Occluded Pixels

After having filled all background pixels that represent holes in the surface, all pixels within the projected silhouette of surface Γ are filled, see Figure 5. However, there are still holes in the surface caused by pixels that represent points of occluded surface parts. In a third processing step, such occluded pixels are replaced by the color, normal and depth values that represent the occluding surface part. This third processing step is very similar to the preceding one of filling background pixels, cf. Section 4.

Again, we first have to identify the pixels that represent parts of occluded surfaces by applying a border test with respect to a minimum distance $\tilde{d}$ between two consecutive surface layers, i. e. two front-facing surface parts that are projected to the same area in image space. For a candidate pixel with depth d, the used masks are similar to those in Figure 4, where, now, white pixels represent those pixels with depth values greater than $d - \tilde{d}$ and dark pixels may have any depth value. If the candidate pixel is identified as being occluded, its color, normal and depth values are replaced by the values of the pixel with minimum depth within the filter's stencil.

In Figure 6, we show the effect of filling occluded pixels when applied to the skeleton hand data set. The input to this third processing step is the output of the second step shown in Figure 5. For the filling of occluded pixels, again, only one iteration was needed.

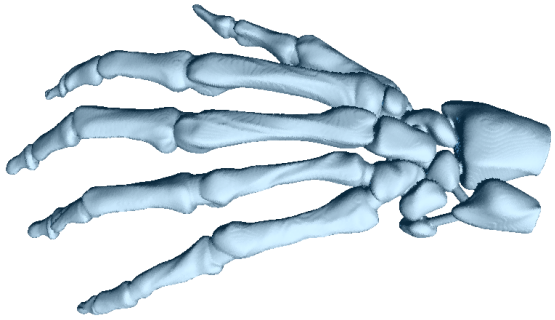

Fig. 6 Skeleton hand data set after point rendering, filling background pixels, and one step of filling occluded pixels (Data set courtesy of Stereolithography Archive, Clemson University).

6 Smoothing

The output of the third processing step is a point cloud rendering, where all holes in the surface have been appropriately filled. For the filling, we used neighboring pixels of minimum depth, which leads to a piecewise constant representation of the surface in image space. In order

to generate a smooth-looking representation we apply image-space smoothing as a last processing step.

For smoothing we apply a low-pass (or smoothing) filter of size 3×3 such as the ones shown in Table 1. Though both the box (or mean) filter and the Gaussian filter could be applied, we prefer an alleviated Gaussian filter, where the middle pixel is not weighted by $\frac{4}{16}$ but by $\frac{16}{28}$, to avoid blurring of the image. To not mix background colors with non-background colors, the filter is only applied to all non-background pixels.

$\frac{1}{9}$

1	1	1
1	1	1
1	1	1

Box filter

$\frac{1}{16}$

1	2	1
2	4	2
1	2	1

Gaussian filter

Table 1 Common low-pass filters of size 3×3. If a filter is applied to a pixel, it is assigned the weighted sum of all neighboring non-background pixels with the given weights.

Figure 7 shows the smoothed version of Figure 6. A single iteration of applying the smoothing filter suffices to produce the desired result.

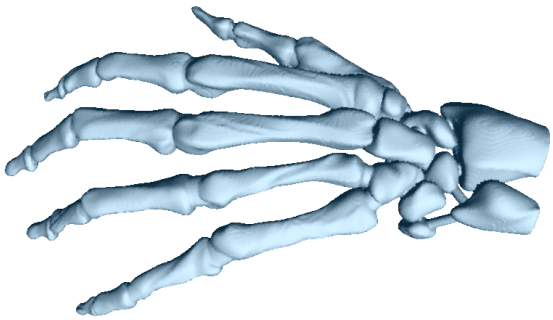

Fig. 7 Point cloud rendering of skeleton hand data set after applying the entire processing pipeline. For the final smoothing step, an alleviated Gaussian filter of size 3×3 has been applied (Data set courtesy of Stereolithography Archive, Clemson University).

7 Anti-aliasing

When having a close-up look at the results generated by the processing steps described in the previous four sections, one can observe aliasing artifacts. Figure 8(a) shows a close-up view of Figure 7. The staircase effects become particularly obvious along the silhouettes, since the smoothing filter is only applied to non-background pixels.

To detect silhouette pixels, i. e. pixels at the border of background pixels and non-background pixels as well as pixels at the border of a front surface layer and a back surface layer, we apply a high-pass filter to the depth values. Many high-pass filters exist and are commonly applied for edge detection. Any of these could be applied. We choose to apply a Laplace filter for our purposes, as one filter can simultaneously detect edges in all directions. Table 2 shows the Laplace filter of size 3×3 that we applied to our examples.

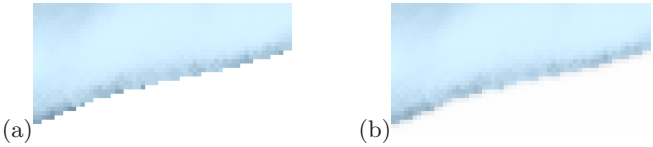

Fig. 8 (a) Close-up view on skeleton hand data set exhibits aliasing artifacts along the silhouette of the object. (b) Anti-aliasing by blending with high-pass-filtered depth buffer texture (Data set courtesy of Stereolithography Archive, Clemson University).

0	-1	0
-1	4	-1
0	-1	0

Table 2 Laplace filter used for edge detection.

Having applied the Laplace filter to the depth values, we obtain a texture with all the silhouette pixels. This resulting texture can be blended with the color texture to obtain an anti-aliased image. Before blending the two textures, one can apply a thresholding to the high-pass-filtered depth information in order to decide whether only the background-foreground transitions should be anti-aliased (high threshold) or whether the front-back surface layer transitions should also be further anti-aliased (low threshold). Figure 8(b) shows the anti-aliased image of Figure 8(a).

8 Illustrative Rendering

The detection of silhouettes described in the previous section and the concurrent processing of a normal map, throughout the whole pipeline easily opens up for applying some non-photorealistic rendering techniques to obtain illustrative drawings [13]. Rendering silhouettes and feature lines in an exaggerated fashion emphasizes the geometrical structure of the object. It is often been used to enhance depth perception, e. g. when two tube-like surface structures exhibit a crossing in image space.

Applying the Laplace filter of Table 2 to the depth values leads to a silhouette rendering. As described in the previous section, a thresholding can be used to adjust the amount of silhouettes that are rendered. Figure 9(a) shows such a silhouette rendering for the skeleton hand data set, while Figure 9(b) shows the blending of silhouette rendering with the photorealistic rendering of Figure 7.

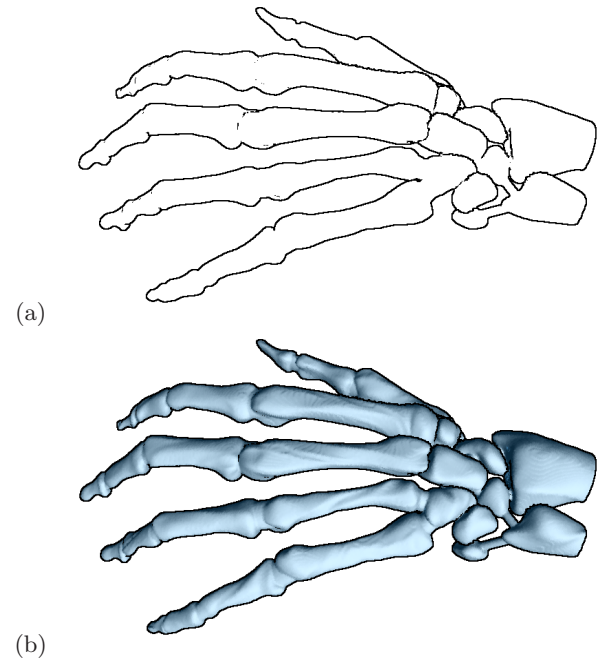

Fig. 9 (a) Silhouette rendering of skeleton hand data set. The silhouettes are detected by applying a 3×3 Laplace filter to the depth values. To make the silhouettes more visible, the lines have been thickened. (b) Combination of silhouette rendering with illuminated surface rendering of the skeleton hand data set (Data set courtesy of Stereolithography Archive, Clemson University).

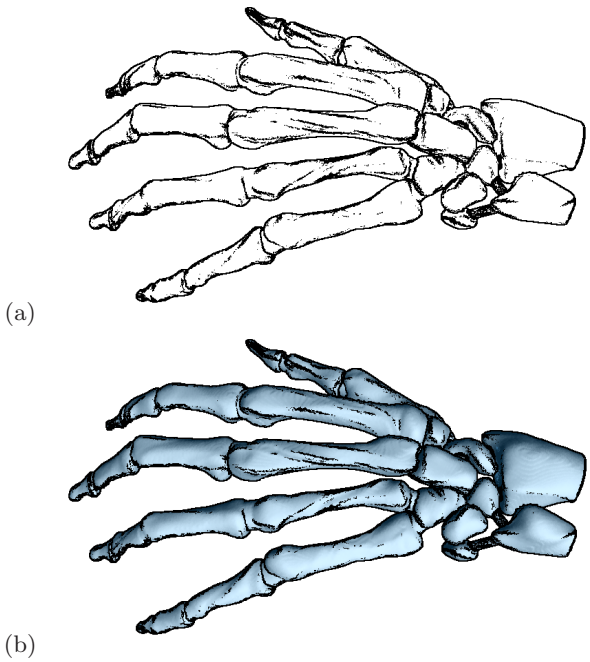

Fig. 10 (a)Rendering of the feature lines of skeleton hand data set. The feature lines are detected by applying a 3×3 curvature filter to the normal map. (b) Final rendering of the illuminated skeleton hand data set, after applying the entire processing pipeline, smoothing and illustrative rendering. (Data set courtesy of Stereolithography Archive, Clemson University).

The available normal map permits a yet much more illustrative type of rendering, by detecting regions with high surface curvature, i. e. feature lines. The surface curvature at a surface point can be easily obtained by exploring the cosine of the angle between surface normals of neighboring surface points. Hence, feature lines can be extracted from the final rendering, by applying a 3×3 filter on the normal map, highlighting regions with high curvature. An illustrative rendering of the feature lines of the skeleton hand data set is shown in Figure 10(a). Figure 10(b) shows the final rendering, including smooth surface rendering, silhouettes and feature lines.

9 Results

We applied our image-space point cloud rendering to two types of point cloud data. The first type of data was obtained by scanning 3D objects. Among the skeleton hand data set used throughout the paper, the data sets include the well-known Happy Buddha data set (courtesy of the Stanford University Computer Graphics Laboratory) and the dragon data set (courtesy of the Stanford University Computer Graphics Laboratory). Of course, polygonal models of these objects exist, but for testing our algorithm we neglected any further information beside the actual point cloud. The second type of data sets were obtained by extracting points on an isosurface of a volumetric scalar field [10]. We applied our methods to the Turbine Blade data set provided with the Visualization Toolkit.

All images have been generated on an Intel XEON 2.66 GHz processor, with a NVIDIA Quadro FX 4500 graphics card. The algorithm was implemented in C++ with OpenGL. For the GPU implementation, we used the OpenGL Shading Language.

We have been able to achieve interactive framerates even when applying the entire processing pipeline including anti-aliasing or illustrative rendering. A detailed listing of the computation times of the individual processing steps for three different data sets is given in Table 3. Note, that the overall framerate nearly doubles if no normal map has to be processed or if only the feature lines have to be extracted without showing the illuminated surface. All shown results are computed in a 1024×1024 viewport.

Figure 11 shows an image-space rendering of the Happy Buddha data set consisting of 544k points. The images are generated using three iterations for filling background pixels and one iteration each for filling occluded pixels and smoothing the image. Due to omitting the process of the normal map, we achieved a framerate of 35 fps.

To show the speedup achievable by omitting either the normal map or the color information, we applied the

data set (# points)	Dragon (437k)		Buddha (544k)		Blade (883k)	
viewport	512 × 512	1024 × 1024	512 × 512	1024 × 1024	512 × 512	1024 × 1024
point rendering	11.5 ms	12.0 ms	14.2 ms	14.4 ms	24.8 ms	25.2 ms
background fill. iter.	1.9 ms	8.6 ms	2.0 ms	9.3 ms	1.1 ms	8.8 ms
occluded fill. iter.	4.1 ms	14.7 ms	4.1 ms	15.8 ms	3.8 ms	14.2 ms
smoothing	0.9 ms	5.9 ms	1.0 ms	5.9 ms	1.5 ms	7.0 ms
anti-alias./illustrative	1.2 ms	1.9 ms	1.1 ms	1.9 ms	1.1 ms	2.1 ms
overall w. illustr. rend.	51 fps	20 fps	45 fps	18 fps	31 fps	16 fps
overall	**95 fps**	**38 fps**	**82 fps**	**35 fps**	**60 fps**	**29 fps**

Table 3 Computations times for individual processing steps for three different data sets with two sizes for the viewports. The time in milliseconds is given for each single computation step, applied to the color buffer and normal map. The overall framerate with illustrative rendering includes the complete processing pipeline applied to both, the color buffer and the normal map, with required number of iterations. In comparison, the overall framerate for only rendering (including anti-aliasing) the surface is shown in the last row.

Fig. 11 Image-space rendering of the Happy Buddha data set with 544k surface points. Background pixels are filled with three iterations, while occluded pixels are filled with one iteration and one smoothing step is applied. The average framerate for the final rendering is 35 fps.

Fig. 12 Rendering of the feature lines of the dragon data set with 437k surface points. The average framerate is 38 fps.

process pipeline only to the normal map of the dragon data set and rendered the feature lines in Figure 12.

Finally, we include a picture of an illustrative point cloud rendering of the Turbine Blade data set with 883k points in Figure 13. We applied two filter iterations each for filling background and occluded pixels. In this example, one can observe that the filling of occluded pixels works perfectly, even if consecutive layers of surfaces come close to each other. We obtained a framerate of 16 fps.

10 Discussion

The results document that we have been able to achieve interactive frame rates for all models without any precomputations while producing Gouraud-like surface shading. We want to emphasize that an increasing number of points would not slow down our algorithm. In fact, it will, in general, run faster the more points are to be processed, as increasing point number means increasing point density and thus less hole filling. The hole filling procedure is actually the most time-consuming step of our algorithm.

Our algorithm requires surface normals at the given surface points in order to apply the Phong illumination model and to process the normal map. Any surface rendering with non-constant shading does require surface normal information at some point. When the point cloud represents the isosurface of a volumetric scalar field, the normals are given in form of gradients of the scalar field. In case we are using scanned objects and no surface normals are available, they can quickly be computed by computing a least-squares fitting plane through the k-nearest neighbors. The k-nearest neighbors are fast to compute, but can typically not be used for local surface reconstruction, as they lead to gaps in the surface representation in case of non-equidistant point sampling. However, for estimating the surface normal, the k-nearest neighbors are sufficiently precise.

The distance $\tilde{d}$ that has been introduced in Section 5 to distinguish between different front-facing surface layers needs to be smaller than the distance between any of two such front-facing surface layers. Since this comparison is made in the range of the depth values, which have

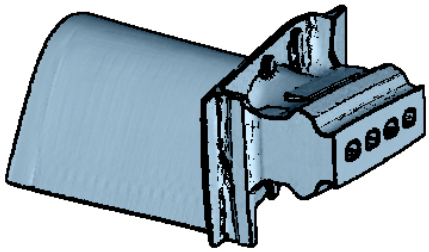

Fig. 13 Point cloud rendering of the Turbine Blade data set, consisting of 883k surface points. Filling background pixels just like filling occluded pixels is done with two filter iterations each. Additional one smoothing step and illustrative rendering is applied, resulting in an average framerate of 16 fps.

been scaled to the interval $[0, 1]$, we could use one distance $\tilde{d}$ for all our examples and did not encounter any problem. We chose a rather small distance $\tilde{d} = 0.0001$.

11 Conclusions

We have presented a novel approach to compute high-quality renderings of point clouds by only applying image-space operations. The lit point cloud attached with normals is projected to image space while depth values are calculated for each pixel. A sequence of filters are applied to this image to fill holes in the projected surface in form of pixels showing background or occluded surface parts. Finally, a smoothing step is applied to generate smooth renderings out of the piecewise constant images. In addition, an anti-aliasing procedure is introduced to avoid aliasing artifacts at the silhouettes of the surface. An illustrative rendering of the surface can be obtained by emphasizing such silhouettes in addition to feature lines that are obtained from the normal map.

The presented approach was implemented on the GPU and tested on several data sets. It achieves interactive rates for point cloud renderings without any pre-computation. The achieved results show correct surface renderings that are comparable to those obtained by Gouraud shading. The illustrative rendering approach can be used to improve the understanding of the data.

Acknowledgements This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under the project LI1530/6-1 "SmoothViz: Visualization of Smoothed Particle Hydrodynamics Simulation Data".

References

1. Alexa, M., Behr, J., Cohen-Or, D., Fleishman, S., Levin, D., Silva, C.T.: Point set surfaces. In: VIS '01: Proceedings of the conference on Visualization '01, pp. 21–28. IEEE Computer Society, Washington, DC, USA (2001)
2. Alexa, M., Gross, M., Pauly, M., Pfister, H., Stamminger, M., Zwicker, M.: Point-based computer graphics. In: SIGGRAPH 2004 Course Notes. ACM SIGGRAPH (2004)
3. Allègre, R., Chaine, R., Akkouche, S.: A flexible framework for surface reconstruction from large point sets. Comput. Graph. **31**(2), 190–204 (2007)
4. Grossman, J.P., Dally, W.J.: Point sample rendering. In: 9th Eurographics Workshop on Rendering, pp. 181–192 (1998). URL citeseer.ist.psu.edu/grossman98point.html
5. Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., Fulk, D.: The digital michelangelo project: 3d scanning of large statues. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp. 131–144. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
6. Linsen, L.: Point cloud representation. Tech. rep., Fakultät für Informatik, Universität Karlsruhe (2001)
7. Linsen, L., Müller, K., Rosenthal, P.: Splat-based ray tracing of point clouds. Journal of WSCG **13**(1–3) (2008)
8. Marroquim, R., Kraus, M., Cavalcanti, P.R.: Efficient point-based rendering using image reconstruction. In: Proceedings of Symposium on Point-Based Graphics, pp. 101–108 (2007)
9. Pfister, H., Zwicker, M., van Baar, J., Gross, M.: Surfels: surface elements as rendering primitives. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, pp. 335–342. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA (2000)
10. Rosenthal, P., Linsen, L.: Direct isosurface extraction from scattered volume data. In: Eurographics / IEEE VGTC Symposium on Visualization (EuroVis 2006) (2006)
11. Rusinkiewicz, S., Levoy, M.: QSplat: A multiresolution point rendering system for large meshes. In: K. Akeley (ed.) Siggraph 2000, Computer Graphics Proceedings, pp. 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman (2000)
12. Sainz, M., Pajarola, R., Lario, R.: Points reloaded: Point-based rendering revisited. In: Symposium on Point-Based Graphics (2004). URL http://www.graphics.ics.uci.edu/pdfs/PointsReloaded.pdf
13. Saito, T., Takahashi, T.: Comprehensible rendering of 3-d shapes. In: SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques, pp. 197–206. ACM Press, New York, NY, USA (1990)
14. Schnabel, R., Moeser, S., Klein, R.: A parallelly decodeable compression scheme for efficient point-cloud rendering. In: Proceedings of Symposium on Point-Based Graphics, pp. 214–226 (2007)
15. Xu, H., Nguyen, M.X., Yuan, X., Chen, B.: Interactive silhouette rendering for point-based models. In: Proceedings of Symposium On Point Based Graphics. ETH Zurich, Switzerland (2004)
16. Yuan, X., Chen, B.: Illustrating surfaces in volume. In: Proceedings of Joint IEEE/EG Symposium on Visualization (VisSym'04), pp. 9–16, color plate 337. the Eurographics Association (2004)
17. Zhang, Y., Pajarola, R.: Deferred blending: Image composition for single-pass point rendering. Comput. Graph. **31**(2), 175–189 (2007)