Interactive Image-space Point Cloud Rendering with Transparency and Shadows

Petar Dobrev Paul Rosenthal Lars Linsen

Jacobs University, Bremen, Germany

{p.dobrev, p.rosenthal, l.linsen}@jacobs-university.de

ABSTRACT

Point-based rendering methods have proven to be effective for the display of large point cloud surface models. For a realistic visualization of the models, transparency and shadows are essential features. We propose a method for point cloud rendering with transparency and shadows at interactive rates. Our approach does not require any global or local surface reconstruction method, but operates directly on the point cloud. All passes are executed in image space and no pre-computation steps are required. The underlying technique for our approach is a depth peeling method for point cloud surface representations. Having detected a sorted sequence of surface layers, they can be blended front to back with given opacity values to obtain renderings with transparency. These computation steps achieve interactive frame rates. For renderings with shadows, we determine a point cloud shadow texture that stores for each point of a point cloud whether it is lit by a given light source. The extraction of the layer of lit points is obtained using the depth peeling technique, again. For the shadow texture computation, we also apply a Monte-Carlo integration method to approximate light from an area light source, leading to soft shadows. Shadow computations for point light sources are executed at interactive frame rates. Shadow computations for area light sources are performed at interactive or near-interactive frame rates depending on the approximation quality.

Keywords: point-based rendering, shadows, transparency

1 INTRODUCTION

Ever since the emergence of 3D scanning devices, surface representation and rendering of the scanned objects has been an active area of research. Acquiring consistent renderings of the surfaces is not trivial as the output of the scanning processes are point clouds with no information about the connectivity between the points. Several techniques have been developed to remedy this problem, ranging from global and local surface reconstruction to methods entirely operating in image space. Traditional approaches involve the generation of a triangular mesh from the point cloud, e.g. [3],which represents a (typically closed) manifold, and the subsequent application of standard mesh rendering techniques for display. Such global surface reconstruction approaches, however, scale superlinearly in the number of points and are slow when applied to the large datasets that can be obtained by modern scanning devices.

This observation led to the idea of using local surface reconstruction methods instead. Local surface reconstruction methods compute for each point a subset of neighboring points and extend the point to a local surface representation based on plane or surface fitting to its neighborhood [1]. The point cloud rendering is, then, obtained by displaying the (blended) extensions. The local surface reconstruction itself is linear in the number of points, but it relies on a fast and appropriate computation of a neighborhood for each point in a pre-computation step. The speed and quality of the approach depends heavily on the choice of the neighborhood.

As the number of points increases, the surface elements tend to shrink and when projected to the image plane have nearly pixel size. This observation was already made by Grossman and Dally [6], who presented an approach just using points as rendering primitives and some image-space considerations to obtain surface renderings without holes. Recently, this image-space technique has been re-considered and improved [8, 11, 13]. This method has the advantage that no surface reconstruction is required and that all image-space operations can efficiently be implemented on the GPU, utilizing its speed and parallelism. It only assumes points (and a surface normal for appropriate illumination). Our approach builds upon the ideas of Rosenthal and Linsen [11]. The image-space operations for transforming a projected point cloud to a surface rendering include image filters to fill holes in the projected surface, which originate from pixels that exhibit background information or occluded/hidden surface parts, and smoothing filters. The contribution of this paper is to provide transparency and shadow capabilities for such point cloud renderings at high frame rates using a *depth peeling* technique.

Depth peeling is a multi-pass technique used to extract (or "peel") layers of surfaces with respect to a

given viewpoint from a scene with multiple surface layers. While standard depth testing in image space provides the nearest fragments of the scene (i.e., the closest layer), depth peeling with n passes extracts n such layers. We describe our depth peeling approach for point cloud surface representations in Section 3.

The information extracted by the depth peeling approach can be put to different applications. We exploit this information for enhancing the capabilities of interactive point cloud renderings with transparency and (soft) shadows. To achieve the first goal, we developed a method for order-independent transparency computation described in Section 4. Once the depth peeling approach has acquired the surface layers, they are blended with object-specific opacity values in the order of their acquisition. This approach allows for rendering of multiple surfaces in one scene using different opacity values for each.

Our second goal was the shadow computation in scenes with point cloud surface representations and the interactive rendering of such scenes. To determine lit and unlit regions of the scene, one has to determine, which points are visible from the light source and which are not. This can be done by rendering the scene with the viewpoint being the position of the light source. In this setting, all those points that are visible can be marked as lit. This approach assumes that we apply the image-space rendering approach with the filters that remove occluded surface parts. The result can be stored in form of a point cloud shadow texture. However, since the scene is typically composed of a large number of points, it is more than likely that multiple visible points project to the same pixel such that marking only one of those points as lit would result in an inconsistent shadow texture. To extract and mark multiple lit points that project to the same pixel, we apply the depth peeling technique, again. Once all lit points have been marked, the scene is rendered from the viewpoint of the observer, where the unlit points are rendered without diffuse or specular lighting, i.e., only using ambient light. To create soft shadows and alleviate aliasing artifacts, we use a Monte-Carlo integration method to approximate light intensity from an area light source. Details are given in Section 5.

The GPU implementation of the algorithms allows us to achieve interactive rates for layer extraction, transparent renderings, and renderings of scenes with (soft) shadows. Results of all steps are presented in Section 6.

2 RELATED WORK

An effective way to incorporate transparency and/or shadows to point-based rendering is the use of ray tracing methods as introduced by Schaufler and Jensen [12]. However, such approaches are typically far from achieving interactive frame rates. The only interactive ray tracing algorithm of point-based models was introduced by Wald and Seidel [14], but they restricted themselves to scenes with shadows, i.e., transparency is not supported. The original EWA splatting paper [16] presents a method for transparency utilizing a software multi-layered framebuffer with fixed number of layers per pixel. Zhang and Pajarola [15] introduced the *deferred blending* approach, which requires only one geometry pass for both visibility culling and blending. They also propose an extension how to use this approach to achieve order-independent transparency with one geometry pass.

An approach to incorporate shadows into interactive point-based rendering can be obtained in a straight-forward manner when first reconstructing the surface from the point cloud (globally or locally) and subsequently apply standard shadow mapping techniques [4]. Botsch et al. [2] applied shadow maps to EWA splatting using GPU implementation to achieve interactive rates. Guennebaud and Gross [7] presented another local surface reconstruction technique, employing moving least squares fitting of algebraic spheres, and also applied shadow mapping to it.

The shadow computation in our approach is similar to irradiance textures (also known as “pre-baked” lighting) in mesh-based rendering [10, 9]. Lit surfaces are determined and stored in a texture by rendering the scene with the viewpoint being the position of the light source. In the rendering pass this information is used to determine which surfaces should be drawn in shadow, and which not.

3 DEPTH PEELING

Depth peeling was introduced by Everitt [5] and is a technique to partition a static 3D scene into sorted layers of geometry. As the name suggests, the layers are extracted in an iterative fashion by “peeling” off one layer after another. The sorting is induced by the given viewpoint. Hence, in each iteration the fragments of the projected visible scene are determined, stored as a representation of the current layer, and removed to compute the subsequent layers. Figure 1 illustrates the depth peeling idea. The depth peeling technique is im-

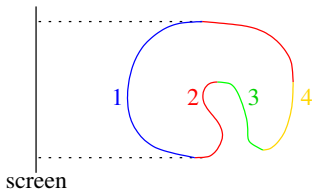


Figure 1: 2D illustration of depth peeling: visible layers of geometry are extracted from front to back. First layer is shown in blue, second in red, third in green, and fourth in yellow.

plemented in a multi-pass algorithm, i.e., to extract n layers the whole scene has to be rendered n times. Each rendering pass is performed with enabled depth testing such that the points closest to the viewpoint and their distances to the viewer are recorded. For the second up to the nth pass, only those points are rendered, whose distance to the viewer is greater than the distance recorded in the preceding pass.

As we want to avoid any (global or local) object-space surface reconstruction, we apply the depth peeling technique to scenes consisting of points only. Consequently, each layer is represented as a set of projected points. Depending on the sampling rate that has been used to acquire the surface, the screen resolution, and the distance to the viewer, it may happen that the points projected to the image plane do not cover all the screen pixels that a reconstructed surface would. Hence, the surface layer may exhibit holes where the background or points of hidden surface layers become visible. Figure 2 illustrates this effect for a 2D scene that is projected to a 1D screen consisting of five pixels. The projection of the first surface layer (blue points) should cover the entire screen. However, there are pixels to which no blue point is mapped. Instead, the second surface layer (red color) or even the background of the scene (grey color) is visible. These gaps in the surface representation of the first layer need to be filled appropriately. Of course, the same issue may arise for all other extracted layers. Hence, in each rendering pass, we apply image-space operations to the extracted layer to fill the gaps in the surface. The image-space operations are executed on the rendering texture using depth information stored in the depth buffer. The operations are executed in four steps: filling surface gaps in form of background pixels (grey pixel in Figure 2), filling surface gaps in form of occluded pixels (red pixel in Figure 2), smoothing the image for an improved rendering quality of the extracted layer, and anti-aliasing applied to the silhouettes and feature lines in the resulting image.

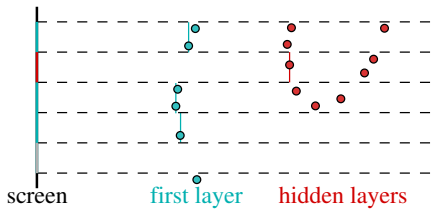


Figure 2: When projecting first layer (blue) in point cloud representation to the screen, the layer exhibits holes such that hidden layers (red) or the background (grey) become visible.

To fill holes caused by pixels exposing background information, one has to identify which background pixels represent holes in the surface layer and which do not. To determine reliably which pixels are to be filled and which not, we apply a filter that checks the 3×3 neighborhood of each background pixel against the set of masks shown in Figure 3. In Figure 3, the framed pixel is the candidate to be filled and the bright ones are neighboring background pixels. The dark pixels may be background or non-background pixels. If the neighborhood matches any of the configurations, the pixel is not filled. Otherwise, its color and depth information is replaced by the color and depth information of the pixel with smallest depth within the stencil of the mask, i.e., within the 3×3 neighborhood. The filters in Figure 3 have been proposed by Rosenthal and Linsen for image-space point cloud rendering. For a detailed discussion of the filters and their application, we refer to the literature [11]. The application of the gap filling step may have to be iterated to fill larger gaps. The operations are always executed on both the rendering texture and the depth texture simultaneously.

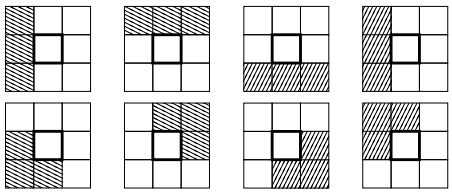


Figure 3: Masks of size 3×3 for detecting pixels exhibiting holes in the projected point cloud surface representation.

To fill pixels that exhibit occluded surface layers, we need to be able to distinguish between pixels from different surface layers. In order to decide whether two pixels belong to the same surface layer, we introduce a parameter d_{min} denoting the minimum distance between two consecutive layers. The parameter depends on the dataset and is typically determined empirically. The occluded pixel filling operation is analogous to the background pixel filling operation. The neighborhood of the candidate pixel is also checked against the masks in Figure 3, only that the bright and the dark pixels in the masks have a different meaning. If the candidate pixel's depth is d, bright pixels correspond to points that have depth values greater than $d + d_{min}$. Dark pixels may have any depth. If the neighborhood satisfies any of the masks, the pixel is not changed. Otherwise, its color and depth information is replaced by the color and depth information of the pixel with smallest depth within the stencil of the mask. Also this second gap filling step may have to be iterated.

To improve the quality of the surface rendering, two additional steps may be applied. The two gap filling steps always replace the gap with the information from the pixel closest to the viewer. A weighted average of the information of those neighboring pixels that belong to the same surface layer would have been preferable. As it would have been too cumbersome to detect all those neighbors, a more efficient way to obtain a simi-

lar result is to apply a subsequent smoothing filter. We apply a Gaussian filter of size 3×3. This smoothing step may be iterated.

However, the smoothing step does not smooth across the silhouette of the projected surface. The silhouettes and feature lines are treated in a separate step that has explicitly been introduced for anti-aliasing purposes. From the depth image, we can easily detect silhouettes and feature lines by checking the depth difference of neighboring pixels against parameter d_{min} (edge detection filtering). All those pixels whose neighborhood exhibit a significant jump in the depth values are marked as contributing to a feature line. To all these pixels, we apply a smoothing that reduces aliasing along the feature lines.

A result of the described pipeline may be seen in Figure 4. We used the Turbine Blade dataset (Data courtesy of Visualization Toolkit) and extracted the first three surface layers. The results have been obtained by applying in each depth peeling pass one iteration of the background pixel filling, occluded pixel filling, Gaussian smoothing, and anti-aliasing.

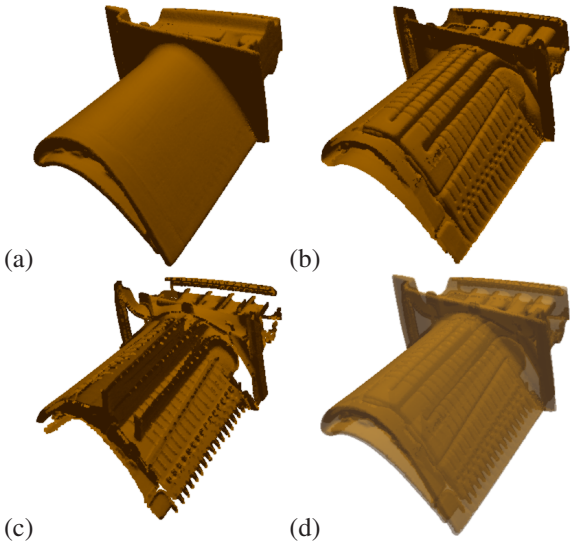


Figure 4: Depth peeling applied to the Blade dataset to extract the (a) first, (b) second, and (c) third layer. The layers are represented as point clouds.The gaps between projected points have been filled using only image-space operations. Blending the layers allows for transparent surface renderings (d).

4 TRANSPARENT SURFACES

Rendering of transparent surfaces is a direct application of depth peeling. It only requires to blend the acquired layers in the order of extraction. However, since point clouds are typically dense, it frequently happens that two or more adjacent points of one surface layer project to the same fragment. Without taking special care of this case, they would be recorded in separate layers by the depth peeling technique such that consecutive layers contain points that should belong to the same surface layer. Figure 5(a) illustrates this problem in the 2D case. Points of the first surface layer are depicted in blue and of the second surface layer in red. Multiple blue points are mapped to one pixel of the screen.

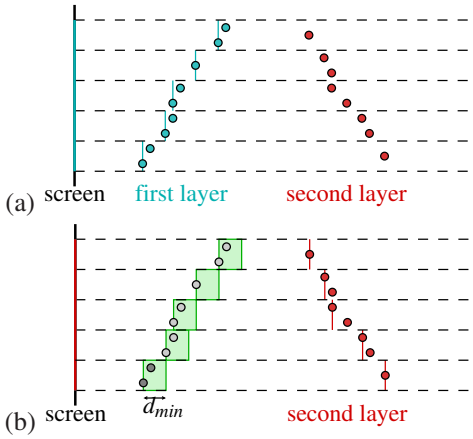


Figure 5: Depth peeling for transparent rendering: (a) first rendering pass records closest points and their depths; b) second rendering pass again records the closest points and their depths, but ignores points less than d_{min} away from the reference depths obtained in the preceding run.

We tackle this problem by using, again, parameter d_{min}, i.e., the minimum distance between two surface layers, to perform ε-z culling: in each rendering pass, depth peeling records the color of the closest point $\mathbf{p}$ for each pixel along with its depth d that serves as a reference for the next run. All points that project to the same pixel as point $\mathbf{p}$ and have a depth less than $d + d_{min}$ must belong to the same surface layer as $\mathbf{p}$. Figure 5(b) illustrates this idea for the example from Figure 5(a). The green boxes of width d_{min} indicate the area that is considered as one surface layer. Hence, the second depth peeling pass discards all points with depth less than $d + d_{min}$ and correctly detects only points belonging to the second (red) surface layer, see Figure 5(b).

This procedure of skipping points within depth range $[d, d + d_{min}]$ has already been used to generate the three layers of the Blade dataset shown in Figure 4. All that is left to do for point cloud rendering with transparency is to blend the layers front to back with an application-specific opacity value α. The result can be seen in Figure 4(d). The opacity value used for all layers was $\alpha = 0.5$.

5 SHADOW TEXTURES

Point cloud shadow textures are basically Boolean arrays that store which points are lit and which not. Once the shadow texture is determined, lit points are drawn

properly illuminated with ambient, diffuse, and specular reflection components using Phong's illumination model, while unlit points are only drawn using the ambient reflection component. This illumination creates the effect of shadows, as only those points are marked unlit where the light source is occluded by other surface parts.

To determine which points are visible from the light source, we render the scene with the light source's position being the viewpoint with depth testing enabled. All visible points are marked in an array. However, as in Section 4 we observe that, due to the high point density, it is not unusual that several adjacent points of one surface layer project to the same fragment position. The suggested procedure would only mark the closest point for each fragment as lit, which would lead to an inconsistent shadow textures. Figure 6 illustrates the problem for a scene with only one surface layer and no occlusion. The points of the entire surface should be marked as lit. However, due to the described issue, only the closest points (red) are marked as lit, while the others (blue) remain unlit. When observing the scene from a position different from the position of the light source, the unlit points become visible and the rendering exhibits strange shadow patterns.

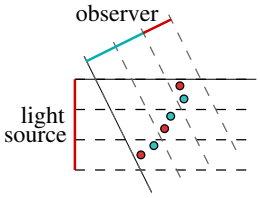


Figure 6: Inconsistent shadow texture in case of high point density: marking only the closest points to the light source as lit, leaves unlit points on the same surface part. The unlit points become visible when positions of observer and light source do not coincide.

Again, depth peeling is the key to solve this problem, but we apply it differently. While for transparent surface rendering our goal was to extract different surface layers, now we want to find all the points that belong to a single surface layer, namely the closest one.

To decide, which points belong to one layer, we consider again parameter d_{min}, i.e., the minimum distance between two surface layers. We render the point cloud from the position of the light source. Let d be the depth of the closest point $\mathbf{p}$ for a given pixel. Then, we consider all points that project to that pixel and have depth values less than $d + d_{min}$ as belonging to the same surface layer as $\mathbf{p}$. Therefore, we mark them as lit.

However, since depth is measured as the distance to the viewing plane, applying the same offset d_{min} for all points would result in an inconsistent shadow texture. The reason is that the depth of the lit layer should always be taken perpendicularly to the surface, and not along the viewing direction. In order to account for the change in the offset, we scale d_{min} by a factor that depends on the surface normal. Let $\mathbf{v}$ be the viewing direction and $\mathbf{n}$ be the surface normal in the light source domain. Then, the offset is given by $\Delta d = \frac{d_{min}}{<\mathbf{v},\mathbf{n}>}$. Given that the viewing direction in the light source domain is $(0,0,-1)$, we obtain that $< \mathbf{v},\mathbf{n} >= -n_z$. To avoid division by zero, this factor is truncated at some maximum value.

As a first step of the algorithm, we obtain the shadow map for the light source, i.e., we record the depth of the closest points as viewed from the light source. As some of the recorded depths might correspond to occluded surface parts, we apply the occluded pixel hole-filling filter on the shadow map. This way pixels, which belong to an occluded surface, will be overwritten in the shadow map and, hence, remain in shadow.

Then, we project all points from the dataset to the light domain and compare their depth values to the ones stored in the shadow map. The points, whose depth is less than the reference depth plus threshold Δd, are recorded as lit in the shadow texture. The rest are left unlit. This operation can very efficiently be implemented on the GPU by using a shader, which takes an array (a texture) of all point positions as input and outputs a boolean array of the same size. The values in the boolean array determine whether the respective point from the input array is lit or not. The shader reads the position of each point from the input texture and projects it in the light domain. Then it compares its depth with the one stored in the shadow map and outputs the result of the comparison to the same texture position as in the input texture.

Figure 7(a) shows a point cloud rendering with shadows applied to the Blade surface shown in Figure 4. It can be observed that the binary marking whether a point is lit or not results in hard shadows with crisp, sharp edges. To create more appealing renderings with softer shadows, we approximate the complex computation of illumination by an area light source using Monte-Carlo integration methods. A number of randomly chosen sample points, lying in the plane perpendicular to the light direction and within the area of the light source, are used as point light sources. A separate shadow texture is computed for each of them. The resulting binary decision values are averaged. The resulting shadow texture is the average of all the shadow textures for the different sample points. It contains no longer just zeros or ones, but floating-point numbers out of the interval $[0,1]$. These numbers determine to what extent the diffuse and specular components are taken into account.

Let k_a, k_d, and k_s denote the ambient, diffuse, and specular components of the illuminated surface at a specific point. Moreover, let $m \in [0,1]$ be the value in the shadow texture stored for that particular point.

Then, the surface color at that point is computed as: $c = k_a + m \cdot (k_d + k_s)$. Figure 7(b) shows the result of point cloud rendering with soft shadows using Monte-Carlo integration methods for the scene that has been shown in Figure 7(a). We have used 30 samples to compute the shadow texture. In the lower left of both figures, we provide a zoomed view into a shadow/no-shadow transition region. The shadows appear much softer in Figure 7(b) and their edges are much smoother.

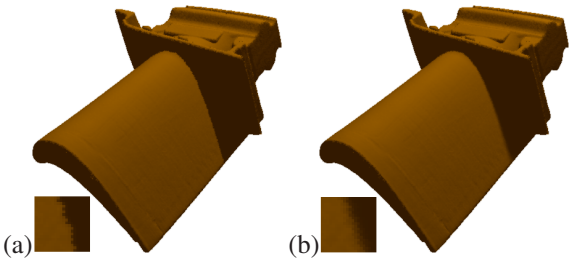


Figure 7: Point cloud rendering with shadows for the Blade dataset: (a) hard shadows using one point light source; (b) soft shadows using Monte-Carlo integration methods with 30 samples to compute the point cloud shadow texture.

6 RESULTS & DISCUSSION

We applied our approach to three types of point cloud data: The model of the Turbine Blade (883k points), given as an example throughout the paper, is from the category of scanned 3D objects. Other datasets from the same category that we have tested our approach on are the Dragon (437k points) and Happy Buddha (543k points) models[1]. Although polygonal representations of these objects exist, any information beside the point cloud was not considered. A synthetical dataset we applied our algorithm to is a set of three nested tori (each 2M points). Finally, we tested our method on two point clouds obtained from isosurface extraction: one from an electron spatial probability distribution field referred to as "Neghip"[2] (128k points) and the other from a hydrogen molecule field[3] (535k points for 3 nested isosurfaces).

All results have been generated on an Intel XEON 3.20GHz processor with an NVIDIA GeForce GTX260 graphics card. The algorithms were implemented in C++ with OpenGL and OpenGL Shading Language for shader programming. All images provided as examples or results in the paper have been captured from a 1024×1024 viewport. One iteration of each of the image-space operations described in Section 3, i.e., background pixels filling, occluded pixels filling, smoothing, and anti-aliasing, was used when producing each rendering. A detailed list of computation times for different datasets, number of layers, number of samples, and resolutions is given in Table 1.

The frame rates for point cloud rendering with local Phong illumination are between 102 fps and 7.8 fps for datasets of sizes between 128k and 6M points and a 1024×1024 viewport. The computation times exhibit a linear behavior in the number of points and a sub-linear behavior in the number of pixels. There is no pre-computation such as local surface reconstruction necessary. All methods directly operate on the point cloud. All operations are done in image space.

For rendering with transparency, the computation times depend linearly on the number of transparent layers. For three transparent surface layers, we obtained frame rates ranging from 28 fps to 2.7 fps. No pre-computations are required. Zhang and Pajarola [15] report better performance for their *deferred blending* approach than depth peeling, but it is only applicable to locally reconstructed surfaces using splats and requires pre-computations. Moreover, it relies on an approximate solution to compute transparency. The frame rates they achieve on an NVidia GeForce 7800GTX GPU are around 37fps for a 303k points dataset and 23 fps for a 1.1M points dataset. As a comparison, our approach renders a 437k points model with 3 layers of transparency at 35fps and a 883k points one at 17.6. Unfortunately, no information about the resolution of the view port used to capture their results is stated to be able to perform a fully adequate comparison.

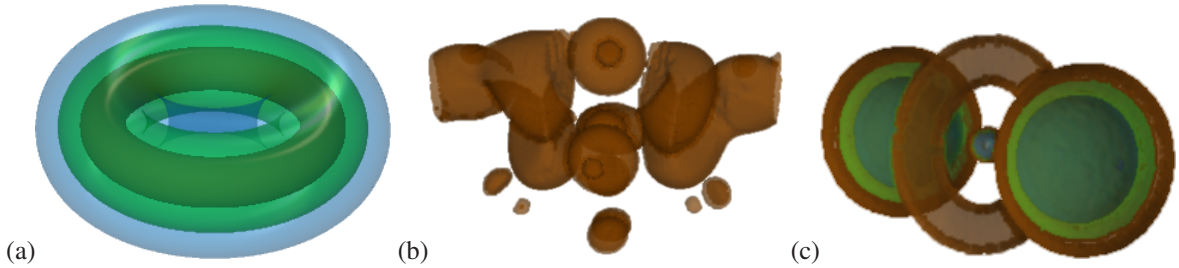
Figure 8(a) shows a transparent rendering of three nested tori, each drawn with a different color and having a different opacity value. The required number of layers to achieve this kind of rendering is six, such that all surface parts of all three tori are captured and blended. When rendering all six layers of this 6M point dataset, the frame rate drops to 1.3 fps. During navigation it may, therefore, be preferable to render just the first layer.

Figures 8(b) and (c) show examples of how our approach can be applied in the context of scientific visualization. When a scalar field is only known at unstructured points in space, an isosurface can be computed by interpolating between neighboring points. The result is given in form of an unstructured set of points on the isosurface, i.e., a point cloud. The datasets we used actually represent scalar fields defined over a structured grid, but for a proof of concept we re-sampled the datasets at uniform randomly distributed points in space. In Figure 8(b), we extracted an isosurface with many components and 128k points, whereas in Figure 8(c) we used three isovalues to extract multiple nested isosurfaces with a total of 535k points. Some surface parts are completely occluded by others. A transparent rendering helps the user to fully observe

[1] Data courtesy of Stanford University Computer Graphics Lab
[2] Data courtesy of VolVis distribution of SUNY Stony Brook
[3] Data courtesy of SFB 382 of the German Research Council

Dataset	Blade		Happy Buddha		Dragon		3 nested tori		Neghip		Hydrogen	
# points	883k		543k		437k		3×2M		128k		535k in total	
Resolution	512^2	1024^2	512^2	1024^2	512^2	1024^2	512^2	1024^2	512^2	1024^2	512^2	1024^2
Local illumination	52	52	83	64	103	68	8	8	235	82	72	48
Transparency (3 layers)	17.6	17.5	28	22	35	23	2.7	2.7	83	27	24	15
Transparency (6 layers)	8.8	8.8	14	11	18	12	1.4	1.4	43	14	12	8
Shadows (1 sample)	26	25	40	39	50	49	4	3.7	145	64	40	31
Shadows (5 samples)	9	9	14	14	18	17	1.3	1.1	62	35	14	14
Shadows (10 samples)	5	5	7	7	9.6	9	0.6	0.6	35	22	8	7.5

Table 1: Frame rates in frames per second (fps) for rendering of point clouds with local illumination only, with transparency (using 3 and 6 blending layers), and with shadows computed with 1, 5, and 10 samples used for approximation of an area light source. One step for each hole filling filter was applied. No pre-computations are necessary.



(a) (b) (c)

Figure 8: Image-space point cloud rendering with transparency: (a) Transparent rendering of three nested tori (2M points each) with six blended layers. Each of the tori is drawn in a different color (blue, green, brown) and with a different opacities ($\alpha = 0.3, 0.5, 1.0$). (b) Point cloud with 128k points obtained by isosurface extraction of the volumetric scalar field "Neghip" is rendered with transparency ($\alpha = 0.7$) at 25 fps. (c) Three nested isosurfaces are extracted from a hydrogen molecule scalar field in form of point clouds with a total of 535k points. The visualization (at 9.8 fps) with semi-transparently rendered surfaces ($\alpha = 0.3, 0.5, 1.0$) allows the user to observe surfaces that are entirely occluded by others.

the isosurface extraction results. The transparent point cloud renderings use four and six surface layers, respectively, and run at frame rates of 25 fps and 9.8 fps.

The frame rates for generating renderings with shadows by first computing a shadow texture are also presented in Table 1. For low number of samples for Monte-Carlo integration, we achieve interactive rates for most tested models. For comparable models, our frame rates are higher than what has been reported for interactive ray tracing on splats [14] and similar to the ones reported for using shadow maps on splats [2]. These approaches, however, require a local surface reconstruction from the point cloud representation in a pre-processing step. For large datasets such local surface reconstructions can have a substantial computation time. Wald and Seidel [14] report performance of about 5 frames per second for comparable models with shadows and Phong shading, using a view port of 512x512 on a 2.4GHz dual-Opteron PC. On modern day hardware their approach would still be slower than what we have achieved (26 fps), since it utilizes only the CPU. The GPU accelerated EWA splatting approach of Botsch et al. [2] achieved a frame rate of about 23 fps on a GeForce 6800 Ultra GPU for rendering a model of 655k points with shadows. For comparison, our approach renders a 543k points model at 40 fps with one sample for shadows computation. On today's GPUs, their approach would achieve similar performance, but it still requires a pre-processing step to compute the splats. Moreover, for objects and light sources that do not change their relative position our approach also allows the shadow texture to be precomputed and loaded along the point cloud. This way soft shadows, computed with lots of samples, can be rendered at highly interactive rates, imposing almost no load on the rendering pipeline.

A limitation of our approach comes from the resolution of the shadow map used to generate the shadow texture. If the resolution is chosen high, it is likely that the shadow texture will contain more "holes" and hence require more steps of the hole-filling filter to be applied. If the resolution is chosen lower, such that a couple of steps suffice, the edges of the shadow appear crisp and jaggy. This problem can be alleviated by using more samples for the area light source integration, which will provide soft anti-aliased shadows. If the scene cannot be rendered with multiple samples at interactive rates, an interactive rendering mode can be used: while navigating through the scene, i.e., rotating, translating or zooming, only one sample is used for shadow computation to provide high responsiveness. When not interacting, soft shadows are computed with a given number of samples.

A rendering of the Dragon dataset with shadows is shown in Figure 9. Ten samples were used for the shadow texture computation. The frame rate for that

rendering is 9.6 fps, which allows for smooth interaction.

Figure 9: Interactive rendering of the Dragon point cloud model with soft shadows at 9.6 fps. 10 samples are taken for the Monte-Carlo integration over the area light source.

Although all operations were executed without any computations in object space, we only introduced one intuitive parameter, namely the minimum distance d_{min} between two consecutive surface layers. This parameter was used at multiple points within our rendering pipeline. An improper choice of this parameter can produce severe rendering artifacts. For many datasets there is a wide range of values from which a suitable value for d_{min} can be chosen. Only when consecutive layers happen to get close to each other as, for example, for the Blade dataset, one has to choose d_{min} carefully. However, as the impact of the choice becomes immediately visible, an empirical choice was quickly made for all our examples.

7 CONCLUSION

We presented an approach for interactive rendering of surfaces in point cloud representation that supports transparency and shadows. Our approach operates entirely in image space. In particular, no object-space surface reconstructions are required. Rendering with transparency is achieved by blending surface layers that have been computed by a depth peeling algorithm. The depth peeling approach is also applied to compute point cloud shadow textures. A Monte-Carlo integration step was applied to create soft shadows. We have demonstrated the potential of our approach to achieve high frame rates for large point clouds. To our knowledge, this is the first approach that computes point cloud rendering with transparency and shadows without local surface reconstruction.

ACKNOWLEDGEMENTS

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under project grant LI-1530/6-1.

REFERENCES

[1] Marc Alexa, Markus Gross, Mark Pauly, Hanspeter Pfister, Marc Stamminger, and Matthias Zwicker. Point-based computer graphics. In *SIGGRAPH 2004 Course Notes*. ACM SIGGRAPH, 2004.

[2] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Eurographics Symposium on Point-Based Graphics*, pages 17–24, 2005.

[3] Frédéric Cazals and Joachim Giesen. Delaunay triangulation based surface reconstruction. In Jean-Daniel Boissonnat and Monique Teillaud, editors, *Effective Computational Geometry for Curves and Surfaces*. Springer-Verlag, Mathematics and Visualization, 2006.

[4] Florent Duguet and George Drettakis. Flexible point-based rendering on mobile devices. *IEEE Comput. Graph. Appl.*, 24(4):57–63, 2004.

[5] Cass Everitt. Introduction interactive order-independent transparency. White Paper, NVIDIA, 2001.

[6] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques '98*, pages 181–192. Springer, 1998.

[7] Gaël Guennebaud and Markus Gross. Algebraic point set surfaces. In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, page 23, New York, NY, USA, 2007. ACM.

[8] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In *Proceedings Symposium on Point-Based Graphics*, pages 101–108, 2007.

[9] Tom Mertens, Jan Kautz, Philippe Bekaert, Frank Van Reeth, and Hans-Peter Seidel. Efficient rendering of local subsurface scattering. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, page 51, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Ravi Ramamoorthi and Pat Hanrahan. An efficient representation for irradiance environment maps. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 497–500, New York, NY, USA, 2001. ACM.

[11] Paul Rosenthal and Lars Linsen. Image-space point cloud rendering. In *Proceedings of Computer Graphics International (CGI) 2008*, pages 136–143, 2008.

[12] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 319–328, London, UK, 2000. Springer-Verlag.

[13] R. Schnabel, S. Moeser, and R. Klein. A parallelly decodeable compression scheme for efficient point-cloud rendering. In *Symposium on Point-Based Graphics 2007*, pages 214–226, September 2007.

[14] Ingo Wald and Hans-Peter Seidel. Interactive ray tracing of point based models. In *Proceedings of 2005 Symposium on Point Based Graphics*, pages 9–16, 2005.

[15] Yanci Zhang and Renato Pajarola. Deferred blending: Image composition for single-pass point rendering. *Comput. Graph.*, 31(2):175–189, 2007.

[16] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM.