An Image-space Approach to Interactive Point Cloud Rendering Including Shadows and Transparency

Petar Dobrev[1] Paul Rosenthal[1,2] Lars Linsen[1]

[1] Jacobs University, Bremen, Germany
{p.dobrev, l.linsen}@jacobs-university.de
[2] Chemnitz University of Technology, Germany
paul.rosenthal@informatik.tu-chemnitz.de

Abstract

Point-based rendering methods have proven to be effective for the display of large point cloud surface models, basically replacing global surface reconstruction with local surface estimations, for example, via 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 an approach that avoids any object-space operations and computes high-quality renderings by only applying image-space operations. The image-space operations apply a pipeline of filters to a point cloud after being projected to image space. Those filtering operations appropriately fill background pixels and occluded pixels and produce visually pleasing results using smoothing and anti-aliasing. 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. Again, 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 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. [5],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 [3]. 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 [8], 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 [14, 20, 24]. 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 [20]. 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. The idea of this image-space approach is described in Section 3.

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. In each rendering pass, the depth information of the fragments of the current layer is recorded and used to determine the next layer. In the context of point clouds, each layer consists of one or multiple surface parts in point cloud representation. Each layer is a subset of the entire point cloud, which can be displayed using an image-space point cloud rendering technique. Extracting all the layers in a scene leads to a partition of the point cloud. We describe our depth peeling approach for point cloud surface representations in Section 4.

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 5. 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. One application is the transparent rendering of complex scanned surfaces with multiple layers. Another application is the rendering of multiple, possibly nested isosurfaces (in point cloud representation) extracted from volumetric data fields in the context of scientific visualization.

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 6.

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 7.

2 Related Work

Research in the field of point cloud rendering has gained momentum especially after the Michelangelo project [10] when models with huge amount of points became available for use. Applying traditional global surface reconstruction approaches on the huge datasets was no longer feasible, which led to the development of a range of local surface reconstruction methods [29, 2, 12, 16, 22, 9, 11]. They operate on the point clouds by extending the points with some implicit or explicit surface elements. 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 precomputation 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 [8], 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 [14, 20, 24]. These approaches have an advantageous time complexity and obviously even more so when dealing with very large point cloud models. Our approach follows the ideas of Rosenthal and Linsen [20] avoiding any geometric observations in object space. The authors propose using filters on the rendered image of the lit point cloud and the respective depth buffer to appropriately fill pixels. In particular, pixels that incorrectly exhibit background information or occluded surface parts are corrected. The subsequent application of smoothing filters leads to a smooth surface rendering. Moreover, anti-aliasing and illustrative rendering techniques are supported. However, none of these approaches support the rendering of transparent surfaces or the rendering of scenes with shadows.

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 [23]. Their ray-tracing technique for point clouds is based on sending out rays with a certain width which can geometrically be described as cylinders. Wand and Straßer [26] introduce a similar concept by replacing the cylinders with cones. Adamson and Alexa [1] proposed a method for ray tracing point set surfaces, while Linsen et al. [13] used ray tracing in combination with splatting. 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 [25], but they restricted themselves to scenes with shadows, i.e., transparency is not supported. The original EWA splatting paper [29] presents a method for transparency utilizing a software multi-layered framebuffer with fixed number of layers per pixel. Several follow-up papers [18, 4] discuss how GPUs can be used to speed up the EWA splatting computation. Zhang and Pajarola [28] introduced the *deferred blending* approach, which requires only one geometry pass for both visibility culling and blending. It does so by rendering groups of non-overlapping splats to different images and then operating only in the image domain. They also propose an extension how to use this approach to achieve order-independent transparency with one geometry pass.

Another 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 applying standard shadow mapping techniques [6]. Botsch et al. [4] applied shadow maps to EWA splatting using GPU implementation to achieve interactive rates. The idea of shadow mapping goes back to the approach of Williams [27].

The shadow computation in our approach is similar to irradiance textures (also known as "pre-baked" lighting) in mesh-based rendering [17, 15]. Lit surfaces are determined and stored in a texture by rendering the scene with the view-point 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 Image-space Rendering Using Filters

The goal of the image-space point cloud rendering 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 $\mathbf{x}_i$.

3.1 Point Rendering

Image-space point cloud rendering uses 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 topology 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 1 shows the result of our first processing steps. The data set used is the well-known skeleton hand data set consisting of 327k points.[1] The illuminated points after projection in conjunction with backface culling and depth buffering are displayed.

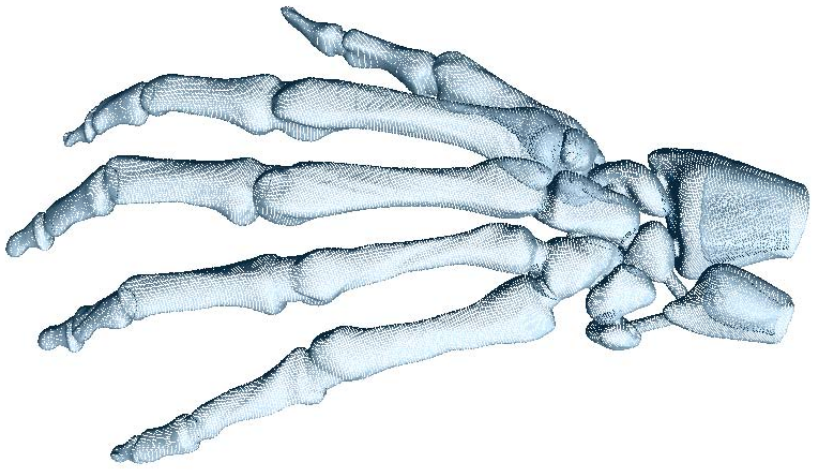


Figure 1: Rendering of the illuminated surface points of the skeleton hand data set containing 327k surface points.

Besides color and position in image space, the depth of each projected

[1](Data set courtesy of Stereolithography Archive, Clemson University.

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.

3.2 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 condition is not met, in general. Especially, when zooming closer to the object, the neighboring points of the point cloud will eventually no longer be projected to adjacent pixels. 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 1.

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 2 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.

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 2, 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 3, 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

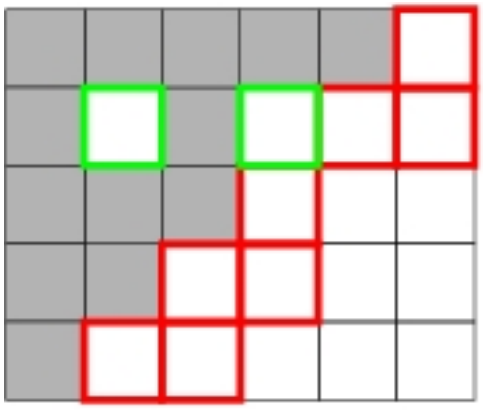


Figure 2: Closeup view of the rendered surface's border. Pixels that have been filled in the first pass (point rendering) are shown in grey, background pixels in white. All background pixels close to a filled pixel are framed. Green frames indicate holes in the surface rendering and the respective pixels have to be filled. Red frames indicate pixels that are beyond the silhouette of the object and must not be filled.

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.

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 3 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 1, the filling of background pixels leads to the result shown in Figure 4. Only one filter pass had to be applied to fill background pixels.

3.3 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 4. 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

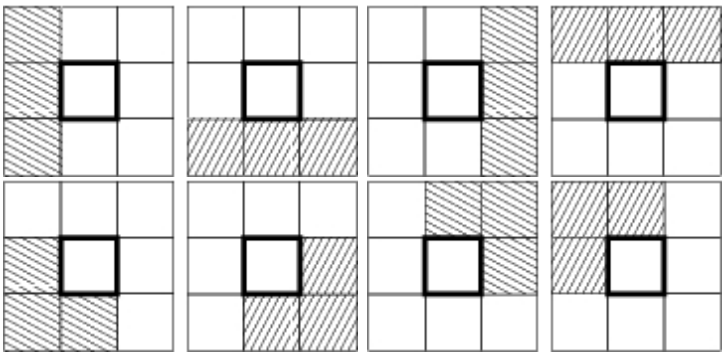


Figure 3: Filters with size 3×3 for detecting whether a background pixel is beyond the projected silhouette of the object. If one of the eight masks matches the neighborhood of a background fragment, it is not filled. White cells indicate background pixels, dark cells may be background or non-background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

the occluding surface part. This third processing step is very similar to the preceding one of filling background pixels.

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 3, 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 5, 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 4. For the filling of occluded pixels, again, only one iteration was needed.

3.4 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-based 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

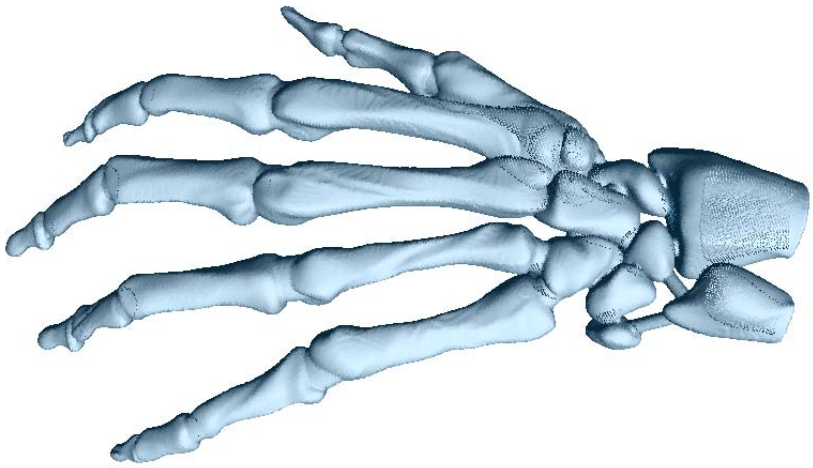


Figure 4: Filling background pixels applied to the output of point rendering (Figure 1) of skeleton hand data set. Only one iteration of the filter had to be applied.

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 $\frac{16}{28}$. The stronger emphasis on the middle pixel avoids blurring of the image. To not mix background colors with non-background colors, the filter is only applied to all non-background pixels.

$$\frac{1}{9}\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} \qquad \frac{1}{16}\begin{array}{|c|c|c|}\hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

Box filter 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 6 shows the smoothed version of Figure 5. A single iteration of applying the smoothing filter suffices to produce the desired result.

3.5 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 7(a) shows a close-up view of Figure 6. The staircase effects become particularly obvious along the silhouettes, since the smoothing filter is only applied to non-background pixels.

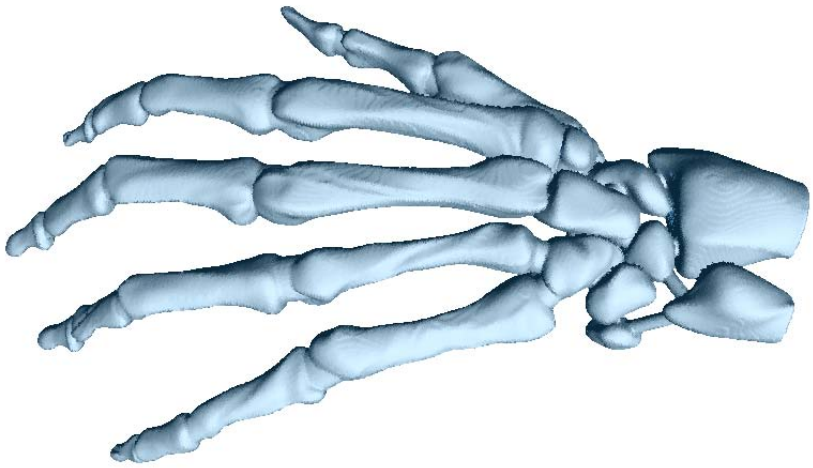


Figure 5: Skeleton hand data set after point rendering, filling background pixels, and one step of filling occluded pixels.

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.

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 7(b) shows the anti-aliased image of Figure 7(a).

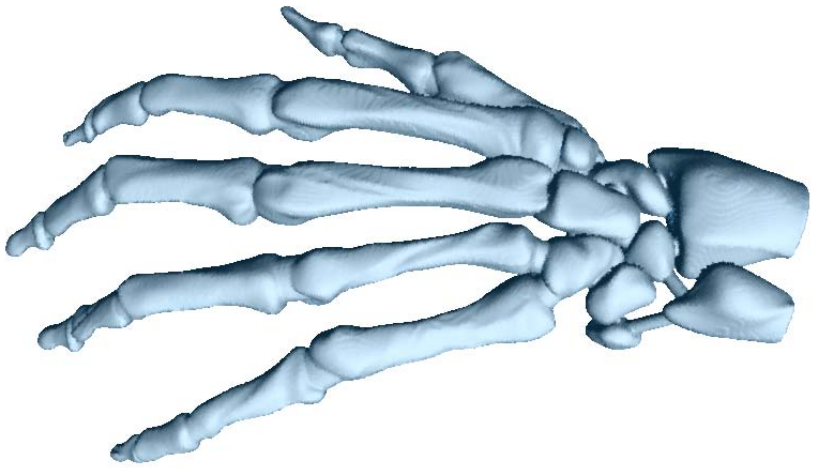
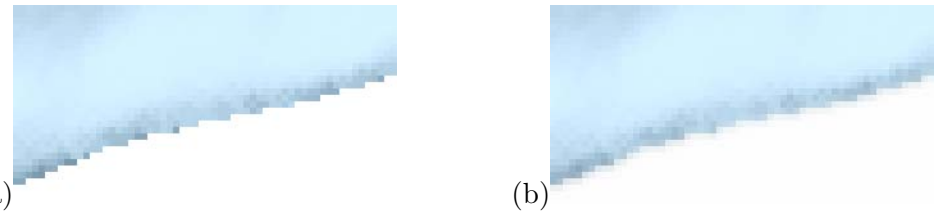


Figure 6: Point cloud rendering of skeleton hand data set after applying the entire processing pipeline. For the final smoothing step, an alleviated Gaussian filter of size 3×3 has been applied.



(a) (b)

Figure 7: (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.

4 Depth Peeling

Depth peeling was introduced by Everitt [7] 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 8 illustrates the depth peeling idea. For illustration purposes, we use depictions of the 2D case. Figure 8 shows a scene consisting of a planar curve that is rendered to a 1D screen. In a first step, the curve is projected onto the screen. The visible parts - depicted in blue - represent the first layer. The first layer is recorded and removed. Then, the remaining parts are projected. Whatever is visible when

projected to the screen is recorded in the second layer - depicted in red - which subsequently is also removed. The process continues until the scene is empty or a desired number of layers has been reached. In the example, four layers - depicted in blue, red, green, and yellow, respectively - are extracted. The generalization to 3D scenes consisting of surfaces projected to a 2D screen is straight forward. In fact, the 2D case can be interpreted as a slice of a 3D scene orthogonal to the screen.

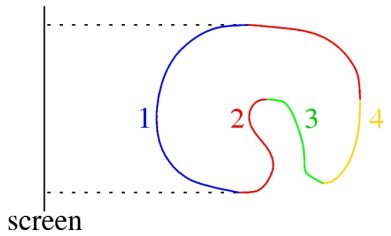


Figure 8: 2D illustration of depth peeling: visible layers of geometry are extracted from front to back. First layer is shown in blue, second in red, third in green, and fourth in yellow.

The depth peeling technique is implemented 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. The first pass of depth peeling is essentially the same as extracting the visible parts of a scene with the help of a depth buffer test. 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 a result, we do not only obtain the visible surface parts, but a sequence of n layers sorted by distance to the viewer. Of course, the projection that is used during rendering has to be the same for all n passes. Both perspective and orthogonal projection are supported.

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 9 illustrates this effect for a 2D scene that is projected to a 1D screen consisting of five pixels. The blue points represent the first surface layer. The projection of the first surface layer should cover the entire screen. However, when the blue points are projected onto the screen, there are pixels to which no blue point is mapped. The surface representation of the

first layer exhibits holes. What becomes visible is the second surface layer (red color) or even the background of the scene (grey color). 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. In each rendering pass, we apply image-space operations to the extracted layer to fill the gaps in the surface.

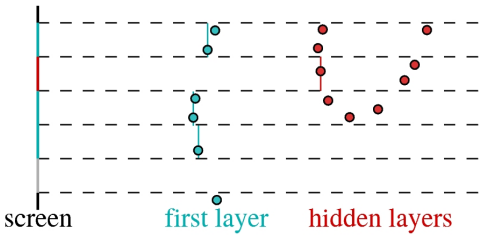


Figure 9: When projecting first layer (blue) in point cloud representation to the screen, the layer exhibits holes such that hidden layers (red) or the background (grey) become visible.

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 9), filling surface gaps in form of occluded pixels (red pixel in Figure 9), 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, see Section 3.

A result of the described pipeline may be seen in Figure 10. We used the Turbine Blade dataset[2] and extracted the first four 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.

5 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 11(a) illustrates this problem in the 2D case. Points of the first surface layer are depicted in

[2]Data courtesy of Visualization Toolkit.

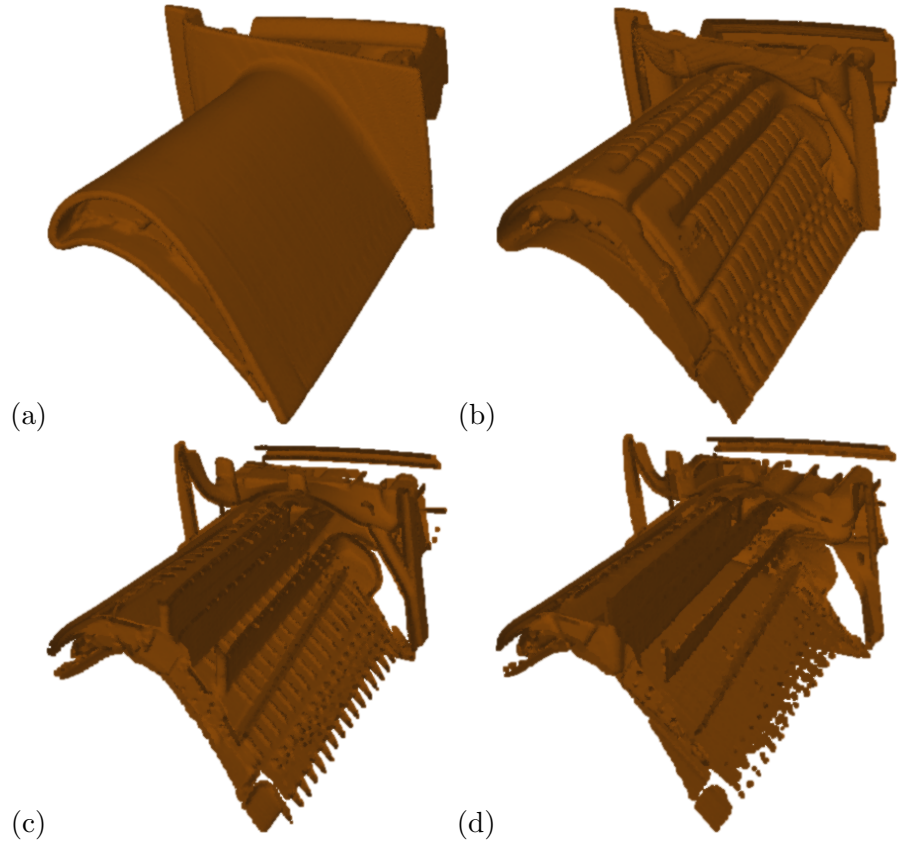


Figure 10: Depth peeling applied to the Blade dataset to extract the (a) first, (b) second, (c) third, and (d) fourth layer. The layers are represented as point clouds. The gaps between projected points have been filled using only image-space operations.

blue and of the second surface layer in red. Multiple blue points are mapped to one pixel of the screen.

We tackle this problem by using, again, parameter $\tilde{d}$, i.e., the minimum distance between two surface layers. 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. If the minimum distance between two surface layers is $\tilde{d}$, all points that project to the same pixel as point $\mathbf{p}$ and have a depth less than $d + \tilde{d}$ must belong to the same surface layer as $\mathbf{p}$. Figure 11(b) illustrates this idea for the example from Figure 11(a). The green boxes of width $\tilde{d}$ indicate the area that is considered as one surface layer. Hence, the second depth peeling pass discards all points with depth less than $d + \tilde{d}$ and correctly detects only points belonging to the second (red) surface layer, see Figure 11(b).

This procedure of skipping points within depth range $[d, d+\tilde{d}]$ has already

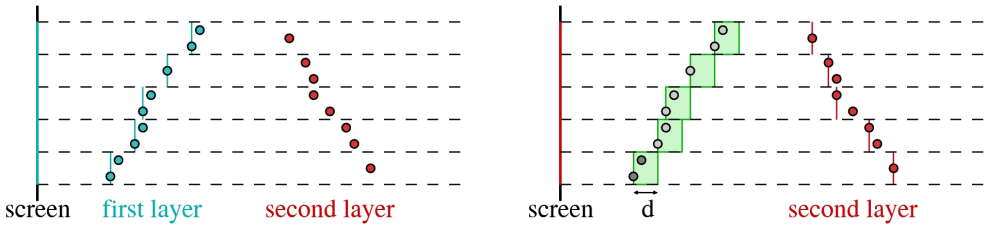


Figure 11: Depth peeling for transparent rendering: (a) first rendering pass records closest points and their depths; b) second rendering pass again records the closest points and their depths, but ignores points less than $\tilde{d}$ away from the reference depths obtained in the preceding run.

been used to generate the four layers of the Blade dataset shown in Figure 10. 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 12.

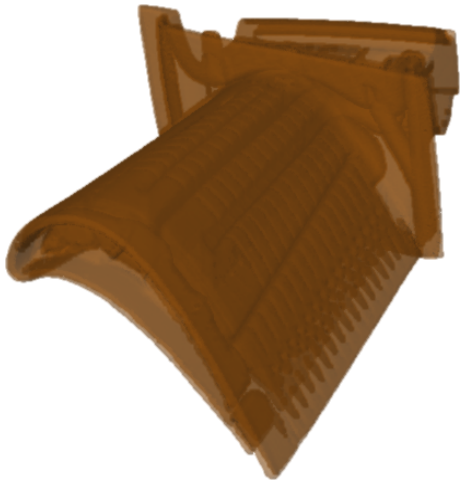


Figure 12: Transparent point cloud rendering by blending the four surface layers of Figure 10 obtained by depth peeling. Blending is executed front to back with opacity value $\alpha = 0.8$.

6 Shadow Textures

To determine which parts of a surface are directly lit by a light source and which parts fall into the shadow of the light source, shadow maps have proven to be efficient. Our approach to generate point cloud shadow textures is related yet different in one important aspect. The standard shadow

map approach records depth information of the scene as viewed from the light source. These depth values are then compared during runtime to the distance of each visible surface point to the light source plane. We, instead, determine and mark all points that are visible from the light source similar to "pre-baking" irradiance textures for polygonal mesh scenes. We will see that this approach provides us with more flexibility in the shadow computation.

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 5 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 13 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.

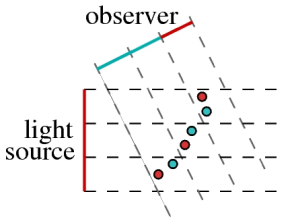


Figure 13: Inconsistent shadow map in case of high point density: marking only the closest points to the light source as lit, leaves unlit points on the same surface part. The unlit points become visible when positions of observer and light source do not coincide.

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 $\tilde{d}$, i.e., the minimum distance between two surface layers. We render the point cloud with the light source's position being the viewpoint. 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 $d + \tilde{d}$ 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 $\tilde{d}$ 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. Figure 14 illustrates the problem. In order to account for the change in the offset, we scale $\tilde{d}$ by a factor that depends on the surface normal. Let $\mathbf{v}$

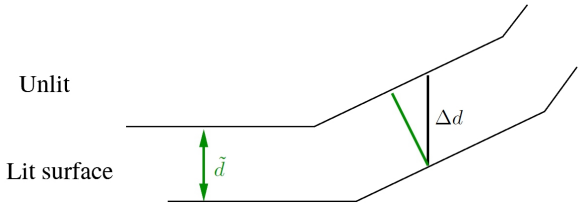


Figure 14: Adjusting the depth offset: as the depth of the lit layer $\tilde{d}$ should be measured perpendicularly to the surface, and not along the viewing direction, $\tilde{d}$ should be scaled accordingly to get the proper offset Δd.

be the viewing direction and $\mathbf{n}$ be the surface normal in the light source domain. Then, the offset is given by

$$\Delta d \quad = \quad \frac{1}{\langle \mathbf{v}, \mathbf{n} \rangle} \cdot \tilde{d}$$

Given that the viewing direction in the light source domain is $(0, 0, -1)$, we obtain that $\langle \mathbf{v}, \mathbf{n} \rangle = -n_z$. To avoid division by zero, this factor is truncated at some maximum value.

This idea is illustrated in Figure 15, where all blue dots belong to the first surface layer. They are extracted by considering all those points that fall in the green boxes of width Δd

As a first step of the algorithm, we perform a rendering pass to record the depth d of the closest point for each pixel in the shadow texture. Here we again have the issue that some pixels of the shadow texture might correspond to holes in the surface, which would result in surface parts being falsely marked as lit. To solve this problem, we apply the occluded pixel hole-filling filter on the shadow texture. This way pixels, which belong to an occluded surface, will be overwritten in the shadow texture and, hence, remain in shadow.

Once the shadow texture is acquired, we project all points from the dataset to the light domain and compare their depth values to the ones stored

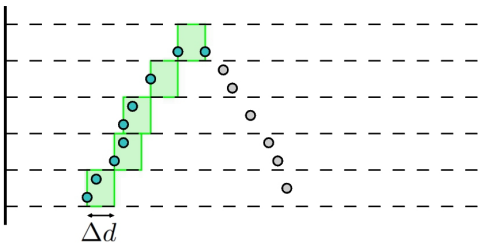


Figure 15: Shadow map generation: using the depth peeling approach, all points within the green boxes of width Δd are marked as lit.

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 remaining points are left unlit. This operation can very efficiently be implemented on the GPU by using a fragment 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 texture and outputs the result of the comparison to the same texture position as in the input texture.

During the first depth peeling rendering pass, the depth information is stored in a texture, which is used as reference for all subsequent runs. All points that are farther from the light source than the reference distance plus threshold $\tilde{d}$ must be discarded. During the rendering pass, we first project the point cloud to the rendering plane and subsequently run the occluded pixel hole-filling filter to make sure that no points, otherwise visible through the holes, are falsely marked as lit. The result is then read back and the respective points are marked as lit in the shadow map. In each iteration, the marked points are excluded from further investigations.

Since the purpose of rendering the scene from the light source is to mark points, the information that is rendered is not color and opacity values, but rather an index number that is passed as an attribute and uniquely identifies each point in the dataset. For an effective implementation, we make use of the integer rendering pipeline, which allows us to directly store and read (unsigned) integer numbers to and from textures. In each rendering pass, we run the occluded pixel hole-filling filter. The determined points within distance $\tilde{d}$ to the reference depth are marked as lit in the shadow map. The process is iterated until there are no more pixels to be marked as lit, i.e., when the read back texture is empty.

In Figure 15, the first rendering pass sets the reference depth for all five fragments and marks the closest points as lit. The second rendering pass reports further points within distance $\tilde{d}$ to the reference depth for four of

the five fragments. These points are also marked as lit. The third rendering pass does not report back any further points. The procedure halts.

Figure 16(a) shows a point cloud rendering with shadows applied to the Blade surface shown in Figure 10. 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 diffusive and specular components are taken into account.

Let k_a, k_d, and k_s denote the ambient, diffusive, 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 16(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 16(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 16(b) and their edges are much smoother.

This flexibility to extend the shadow computation to other models such as the presented soft shadows let to the decision to develop the described point cloud shadow mapping technique.

Since our shadow textures store information per point, they can be exported and reused for scenes, in which the relative position of the light source and the models has not changed. For such applications, there is no need to recompute the shadow texture every time. It can be loaded along with the point cloud.

7 Results & Discussion

We applied our approach to three types of point cloud data: synthetic data, data acquired by scanning the boundary surface of 3D objects, and data obtained by extracting points on an isosurface of a volumetric scalar field. The model of the Turbine Blade (883k points), given as an example throughout

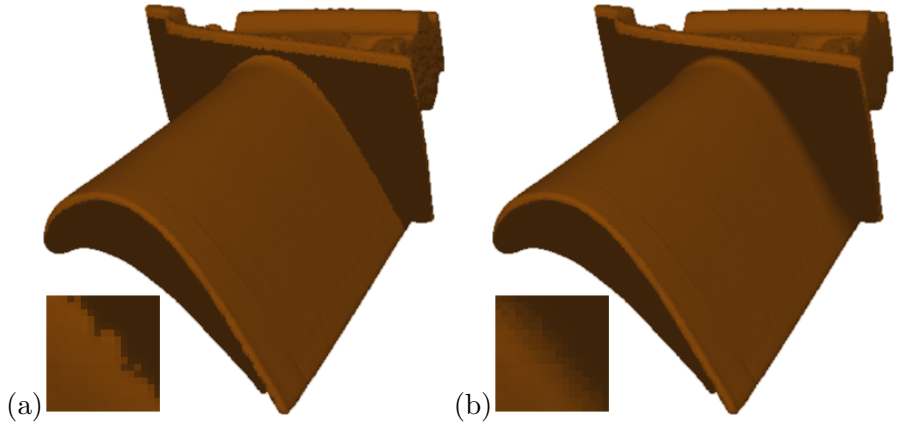


Figure 16: Point cloud rendering with shadows for the Blade dataset: (a) hard shadows using one point light source; (b) soft shadows using Monte-Carlo integration methods with 30 samples to compute the point cloud shadow texture.

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[3]. 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" (128k points)[4] and the other from a hydrogen molecule field (535k points for 3 nested isosurfaces)[5].

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 4, 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 3.

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

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

Dataset	Blade		Happy Buddha		Dragon	
# points	883k		543k		437k	
Resolution	512^2	1024^2	512^2	1024^2	512^2	1024^2
Local illumination	52	52	83	64	103	68
Transparency (3 layers)	17.6	17.5	28	22	35	23
Transparency (6 layers)	8.8	8.8	14	11	18	12
Shadows (1 sample)	26	25	40	39	50	49
Shadows (5 samples)	9	9	14	14	18	17
Shadows (10 samples)	5	5	7	7	9.6	9

Dataset	3 nested tori		Neghip		Hydrogen	
# points	3×2M		128k		535k in total	
Resolution	512^2	1024^2	512^2	1024^2	512^2	1024^2
Local illumination	8	8	235	82	72	48
Transparency (3 layers)	2.7	2.7	83	27	24	15
Transparency (6 layers)	1.4	1.4	43	14	12	8
Shadows (1 sample)	4	3.7	145	64	40	31
Shadows (5 samples)	1.3	1.1	62	35	14	14
Shadows (10 samples)	0.6	0.6	35	22	8	7.5

Table 3: 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.

behavior in the number of points and a sublinear 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. for datasets of sizes between 128k and 6M points and a 1024×1024 viewport. No pre-computations are required. Zhang and Pajarola [28] report better performance than depth peeling, but their approach is only applicable to locally reconstructed surfaces using splats and relies on an approximate solution. We are not aware of any other comparable point-based technique for transparent surface rendering that achieves interactive frame rates.

Figure 17(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 18 and 19 show examples of how our approach can be applied in the context of scientific visualization. When a scalar field is only known at

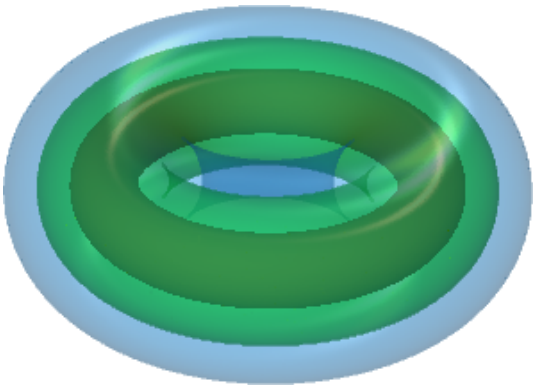


Figure 17: Transparent rendering of three nested tori (2M points each) with six blended layers. Each of the tori is drawn in a different color and with a different opacity value: the innermost is brown and completely opaque, the middle one is green with opacity $\alpha = 0.5$, and the outermost is blue with opacity $\alpha = 0.3$.

Figure 18: Point cloud obtained by isosurface extraction of the volumetric scalar field "Neghip" is rendered with transparency at 25 fps. The surface is represented by 128k points. Four layers are extracted and blended with opacity $\alpha = 0.7$.

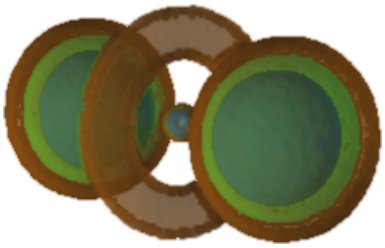


Figure 19: Three nested isosurfaces are extracted from a hydrogen molecule scalar field in form of point clouds with a total of 535k points. The visualization with semi-transparently rendered surfaces (at 9.8 fps) allows the user to observe surfaces that are entirely occluded by others. Each isosurface is blended with a different opacity values, namely $\alpha = 0.5$ for the outermost, $\alpha = 0.3$ for the middle, and $\alpha = 1.0$ for the innermost isosurface.

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 [19, 21]. 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 18, we extracted an isosurface with many components and 128k points, whereas in Figure 19 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 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 3. 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 [25] and similar to the ones reported for using shadow maps on splats [4]. 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 [25] 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. [4] 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

Figure 20: Interactive rendering of the Dragon point cloud model with soft shadows at 9.6 fps. Ten samples are taken for the Monte-Carlo integration over the area light source.

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 pre-computed 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 texture 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 20. Ten samples were used for the shadow texture computation. The frame rate for that rendering is 9.7 fps, which allows for smooth interaction.

Another result of our rendering approach with soft shadows when applied to the Happy Buddha is given in Figure 21. The frame rates are 7.7 fps. The

Figure 21: Interactive rendering of the Happy Buddha point cloud model with soft shadows at 7.7 fps. Ten samples were taken for the Monte-Carlo integration over the area light source to compute the shadow texture.

shadow texture for the Buddha dataset was computed with ten samples.

Although all operations were executed without any computations in object space, we only introduced one intuitive parameter, namely the minimum distance $\tilde{d}$ 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 $\tilde{d}$ 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 $\tilde{d}$ carefully. However, as the impact of the choice becomes immediately visible, an empirical choice was quickly made for all our examples.

8 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] Anders Adamson and Marc Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, page 272, Washington, DC, USA, 2003. IEEE Computer Society.

[2] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Point set surfaces. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 21–28, Washington, DC, USA, 2001. IEEE Computer Society.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] Zhiying He and Xiaohui Liang. A novel simplification algorithm based on mls and splats for point models. In *Proceedings of the 2009 Computer Graphics International Conference*, CGI '09, pages 45–52, New York, NY, USA, 2009. ACM.

[10] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In *Proceedings of ACM SIGGRAPH 2000*, pages 131–144, July 2000.

[11] Bao Li, Ruwen Schnabel, Reinhard Klein, Zhiquan Cheng, Gang Dang, and Shiyao Jin. Technical section: Robust normal estimation for point clouds with sharp features. *Comput. Graph.*, 34:94–106, April 2010.

[12] Lars Linsen. Point cloud representation. Technical report, Fakultät für Informatik, Universität Karlsruhe, 2001.

[13] Lars Linsen, Karsten Müller, and Paul Rosenthal. Splat-based ray tracing of point clouds. In *Proceedings of Fifteenth International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision - WSCG 2007*, pages 15(1–3), 51–58, 2007.

[14] 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.

[15] 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.

[16] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: surface elements as rendering primitives. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[17] 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.

[18] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum*, pages 461–470, 2002.

[19] Paul Rosenthal and Lars Linsen. Direct isosurface extraction from scattered volume data. In Beatriz Sousa Santos, Thomas Ertl, and Kenneth I. Joy, editors, *EuroVis06: Proceedings of the Eurographics/IEEE-VGTC Symposium on Visualization*, pages 99–106. Eurographics Association, 2006.

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

[21] Paul Rosenthal and Lars Linsen. Smooth surface extraction from unstructured point-based volume data using PDEs. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1531–1546, 2008.

[22] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, July 2000.

[23] 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.

[24] 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.

[25] 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.

[26] Michael Wand and Wolfgang Straßer. Multi-resolution point-sample raytracing. In *Graphics Interface*, pages 139–148, 2003.

[27] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, 1978.

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

[29] 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.