Surface Splatting

Matthias Zwicker * Hanspeter Pfister † Jeroen van Baar† Markus Gross*

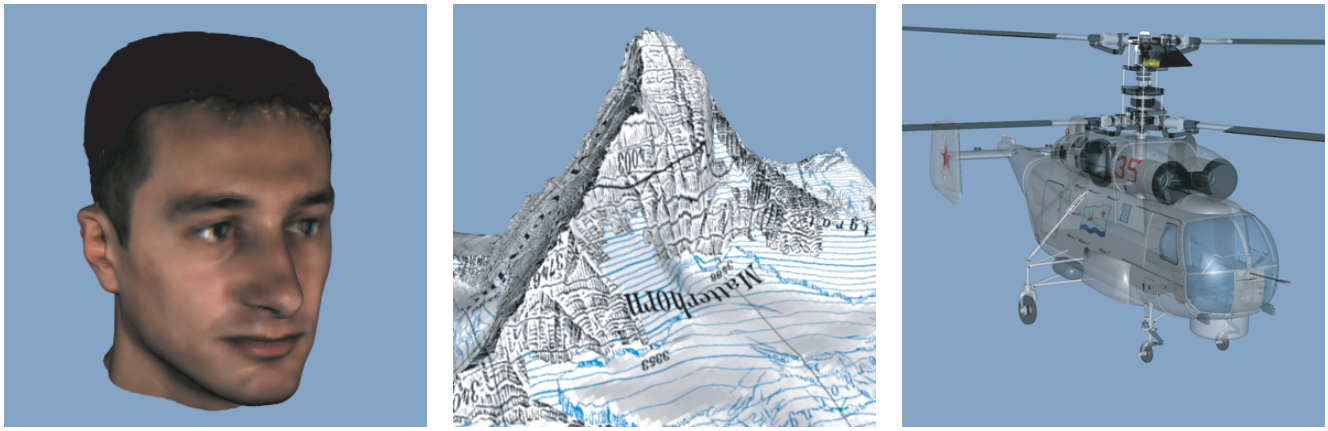

Figure 1: *Surface splatting of a scan of a human face, textured terrain, and a complex point-sampled object with semi-transparent surfaces.*

Abstract

Modern laser range and optical scanners need rendering techniques that can handle millions of points with high resolution textures. This paper describes a point rendering and texture filtering technique called *surface splatting* which directly renders opaque and transparent surfaces from point clouds without connectivity. It is based on a novel screen space formulation of the Elliptical Weighted Average (EWA) filter. Our rigorous mathematical analysis extends the texture resampling framework of Heckbert to irregularly spaced point samples. To render the points, we develop a surface splat primitive that implements the screen space EWA filter. Moreover, we show how to optimally sample image and procedural textures to irregular point data during pre-processing. We also compare the optimal algorithm with a more efficient view-independent EWA pre-filter. Surface splatting makes the benefits of EWA texture filtering available to point-based rendering. It provides high quality anisotropic texture filtering, hidden surface removal, edge anti-aliasing, and order-independent transparency.

Keywords: Rendering Systems, Texture Mapping, Antialiasing, Image-Based Rendering, Frame Buffer Algorithms.

1 Introduction

Laser range and image-based scanning techniques have produced some of the most complex and visually stunning models to date [9]. One of the challenges with these techniques is the huge volume of point samples they generate. A commonly used approach is generating triangle meshes from the point data and using mesh reduction techniques to render them [7, 2]. However, some scanned meshes are too large to be rendered interactively [9], and some applications cannot tolerate the inherent loss in geometric accuracy and texture fidelity that comes from polygon reduction.

Recent efforts have focused on direct rendering techniques for point samples without connectivity [16, 4, 15]. These techniques use hierarchical data structures and forward warping to store and render the point data efficiently. One important challenge for point rendering techniques is to reconstruct continuous surfaces from the irregularly spaced point samples while maintaining the high texture fidelity of the scanned data. In addition, the point rendering should correctly handle hidden surface removal and transparency.

In this paper we propose a new point rendering technique called *surface splatting*, focusing on high quality texture filtering. In contrast to previous point rendering approaches, surface splatting uses a novel screen space formulation of the Elliptical Weighted Average (EWA) filter [3], the best anisotropic texture filtering algorithm for interactive systems. Extending the framework of Heckbert [6], we derive a screen space form of the EWA filter for irregularly spaced point samples without global texture parameterization. This makes surface splatting applicable to high-resolution laser range scans, terrain with high texture detail, or point-sampled geometric objects (see Figure 1). A modified A-buffer [1] provides hidden surface removal, edge anti-aliasing, and order-independent transparency at a modest increase in computation efforts.

The main contribution of this paper is a rigorous mathematical formulation of screen space EWA texture filtering for irregular point data, presented in Section 3. We show how the screen space EWA filter can be efficiently implemented using surface splatting in Section 4. If points are used as rendering primitives for complex geometry, we want to apply regular image textures to point samples during conversion from geometric models. Hence, Section 5 introduces an optimal texture sampling and pre-filtering method for irregular point samples. Sections 6 and 7 present our modified A-buffer method for order-independent transparency and edge anti-aliasing, respectively. Finally, we discuss implementation, timings, and image quality issues in Section 8.

*ETH Zürich, Switzerland. Email: [zwicker,grossm]@inf.ethz.ch
†MERL, Cambridge, MA. Email: [pfister,jeroen]@merl.com

2 Previous Work

Texture mapping increases the visual complexity of objects by mapping functions for color, normals, or other material properties onto the surfaces [5]. If these texture functions are inappropriately band-limited, texture aliasing may occur during projection to raster images. For a general discussion of this problem see [21]. Although we develop our contributions along similar lines to the seminal work of Heckbert [6], our approach is fundamentally different from conventional texture mapping. We present the first systematic analysis for representing and rendering texture functions on irregularly point-sampled surfaces.

The concept of representing objects as a set of points and using these as rendering primitives has been introduced in a pioneering report by Levoy and Whitted [10]. Due to the continuing increase in geometric complexity, their idea has recently gained more interest. QSplat [16] is a point rendering system that was designed to interactively render large data sets produced by modern scanning devices. Other researchers demonstrated the efficiency of point-based methods for rendering geometrically complex objects [4, 15]. In some systems, point-based representations are temporarily stored in the rendering pipeline to accelerate rendering [11, 17]. Surprisingly, nobody has systematically addressed the problem of representing texture functions on point-sampled objects and avoiding aliasing during rendering. We present a surface splatting technique that can replace the heuristics used in previous methods and provide superior texture quality.

Volume splatting [19] is closely related to point rendering and surface splatting. A spherical 3D reconstruction kernel centered at each voxel is integrated along one dimension into a 2D "footprint function." As each voxel is projected onto the screen, the 2D footprints are accumulated directly into the image buffer or into image-aligned sheet buffers. Some papers [18, 14] address aliasing caused by insufficient resampling rates during perspective projections. To prevent aliasing, the 3D reconstruction kernels are scaled using a heuristic. In contrast, surface splatting models both reconstructing and band-limiting the texture function in a unified framework. Moreover, instead of pre-integrating isotropic 3D kernels, it uses oriented 2D kernels, providing anisotropic filtering for surface textures.

3 The Surface Splatting Framework

The basis of our surface splatting method is a model for the representation of continuous texture functions on the surface of point-based graphics objects, which is introduced in Section 3.1. Since the 3D points are usually positioned irregularly, we use a weighted sum of radially symmetric basis functions. With this model at hand, we look at the task of rendering point-based objects as a concatenation of warping, filtering, and sampling the continuous texture function. In Section 3.2 we extend Heckbert's resampling theory [6] to process point-based objects and develop a mathematical framework of the rendering procedure. In Section 3.3 we derive an alternative formulation of the EWA texture filter that we call *screen space* EWA, leading to the surface splatting algorithm discussed in Section 4. In Section 5, we describe how to acquire the texture functions, which can be regarded as a scattered data approximation problem. A continuous approximation of the unknown original texture function needs to be computed from an irregular set of samples. We distinguish between scanned objects with color per point and regular textures that are explicitly applied to point-sampled geometry.

3.1 Texture Functions on Point-Based Objects

In conventional polygonal rendering, texture coordinates are usually stored per vertex. This enables the graphics engine to combine the mappings from 2D texture space to 3D object space and from there to 2D screen space into a compound 2D to 2D mapping between texture and screen space. Using this mapping, pixel colors are computed by looking up and filtering texture samples in 2D texture space at rendering time. There is no need for a sampled representation of the texture in 3D object space. By contrast, the compound mapping function is not available with point-based objects at rendering time. Consequently, we must store an explicit texture representation in object space.

We represent point-based objects as a set of irregularly spaced points $\{\mathbf{P}_k\}$ in three dimensional object space without connectivity. A point $\mathbf{P}_k$ has a position and a normal. It is associated with a radially symmetric basis function r_k and coefficients w_k^r, w_k^g, w_k^b that represent continuous functions for red, green, and blue color components. Without loss of generality, we perform all further calculations with scalar coefficients w_k. Note that the basis functions r_k and coefficients w_k are determined in a pre-processing step, described in Section 5.

We define a continuous function on the surface represented by the set of points as illustrated in Figure 2. Given a point $\mathbf{Q}$ any-

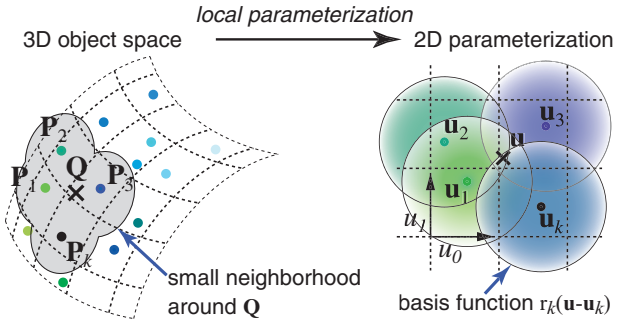

Figure 2: *Defining a texture function on the surface of a point-based object.*

where on the surface, shown left, we construct a local parameterization of the surface in a small neighborhood of $\mathbf{Q}$, illustrated on the right. The points $\mathbf{Q}$ and $\mathbf{P}_k$ have *local coordinates* $\mathbf{u}$ and $\mathbf{u}_k$, respectively. We define the continuous surface function $f_c(\mathbf{u})$ as the weighted sum:

$$f_c(\mathbf{u}) = \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{u} - \mathbf{u}_k). \tag{1}$$

We choose basis functions r_k that have *local* support or that are appropriately truncated. Then $\mathbf{u}$ lies in the support of a small number of basis functions. Note that in order to evaluate (1), the local parameterization has to be established in the union of these support areas only, which is very small. Furthermore, we will compute these local parameterizations on the fly during rendering as described in Section 4.

3.2 Rendering

Heckbert introduced a general resampling framework for texture mapping and the EWA texture filter in [6]. His method takes a regularly sampled input function in *source space*, reconstructs a continuous function, warps it to *destination space*, and computes the properly sampled function in destination space. Properly sampled means that the Nyquist criterion is met. We will use the term *screen space* instead of destination space.

We extend this framework towards a more general class of input functions as given by Equation (1) and describe our rendering process as a resampling problem. In contrast to Heckbert's regular setting, in our representation the basis functions r_k are irregularly spaced. In the following derivation, we adopt Heckbert's notation.

Given an input function as in Equation (1) and a mapping $\mathbf{x} = \mathbf{m}(\mathbf{u}) : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ from source to screen space, rendering involves the three steps illustrated in Figure 3:

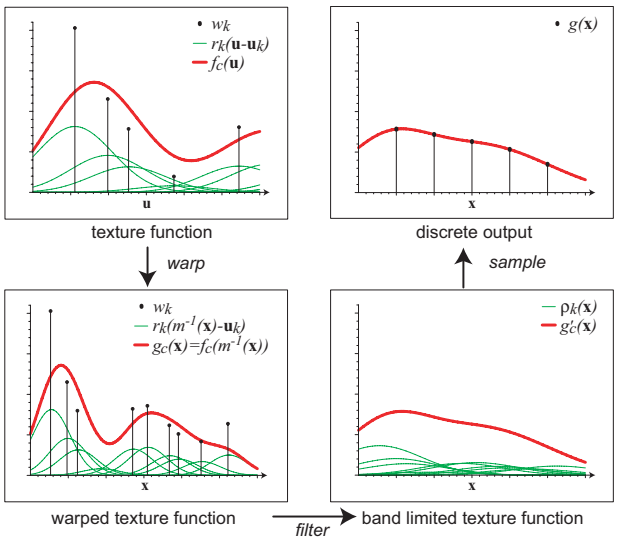

Figure 3: *Warping, filtering, and sampling the texture function.*

1. Warp $f_c(\mathbf{u})$ to screen space, yielding the warped, continuous screen space signal $g_c(\mathbf{x})$:

$$g_c(\mathbf{x}) = (f_c \circ \mathbf{m}^{-1})(\mathbf{x}) = f_c(\mathbf{m}^{-1}(\mathbf{x})),$$

where $\circ$ denotes function concatenation.

2. Band-limit the screen space signal using a prefilter h, resulting in the continuous output function $g'_c(\mathbf{x})$:

$$g'_c(\mathbf{x}) = g_c(\mathbf{x}) \otimes h(\mathbf{x}) = \int_{\mathbb{R}^2} g_c(\xi) h(\mathbf{x} - \xi) d\xi,$$

where $\otimes$ denotes convolution.

3. Sample the continuous output function by multiplying it with an impulse train i to produce the discrete output $g(\mathbf{x})$:

$$g(\mathbf{x}) = g'_c(\mathbf{x}) i(\mathbf{x}).$$

An explicit expression for the warped continuous output function can be derived by expanding the above relations in reverse order:

$$\begin{aligned} g'_c(\mathbf{x}) &= \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) \sum_{k \in \mathbb{N}} w_k r_k(\mathbf{m}^{-1}(\xi) - \mathbf{u}_k) d\xi \\ &= \sum_{k \in \mathbb{N}} w_k \rho_k(\mathbf{x}), \end{aligned} \tag{2}$$

$$\text{where} \quad \rho_k(\mathbf{x}) = \int_{\mathbb{R}^2} h(\mathbf{x} - \xi) r_k(\mathbf{m}^{-1}(\xi) - \mathbf{u}_k) d\xi. \tag{3}$$

We call a warped and filtered basis function $\rho_k(\mathbf{x})$ a *resampling kernel*, which is expressed here as a screen space integral. Equation (2) states that we can first warp and filter each basis function r_k individually to construct the resampling kernels ρ_k and then sum up the contributions of these kernels in screen space. We call this approach *surface splatting*, as illustrated in Figure 4. In contrast to Heckbert, who transformed the screen space integral of Equation (2) to a source space integral and formulated a *source space resampling kernel*, we proceed with (3) to derive a *screen space resampling kernel*.

In order to simplify the integral for $\rho_k(\mathbf{x})$ in (3), we replace a general mapping $\mathbf{m}(\mathbf{u})$ by its local affine approximation $\mathbf{m}_{\mathbf{u}_k}$ at a point $\mathbf{u}_k$,

$$\mathbf{m}_{\mathbf{u_k}}(\mathbf{u}) = \mathbf{x_k} + \mathbf{J}_{\mathbf{u_k}} \cdot (\mathbf{u} - \mathbf{u_k}), \tag{4}$$

$$\text{where} \quad \mathbf{x_k} = \mathbf{m}(\mathbf{u_k}) \quad \text{and the Jacobian} \quad \mathbf{J_{u_k}} = \frac{\partial \mathbf{m}}{\partial \mathbf{u}}(\mathbf{u_k}).$$

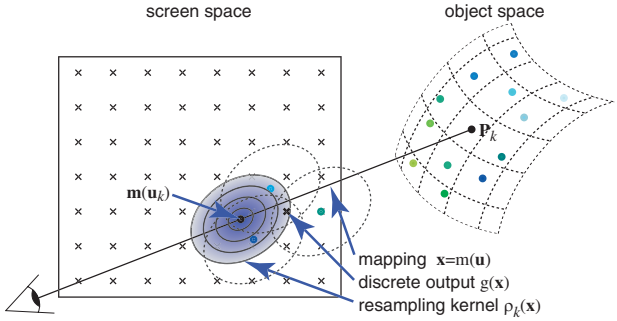

Figure 4: *Rendering by* surface splatting, *resampling kernels are accumulated in screen space.*

Heckbert relied on the same approximation in his derivation [6]. Since the basis functions r_k have local support, $\mathbf{m}_{\mathbf{u}_k}$ is used only in a small neighborhood around $\mathbf{u}_k$ in (3). Moreover, the approximation is most accurate in the neighborhood of $\mathbf{u}_k$ and so it does not cause visual artifacts. We use it to rearrange Equation (3), and after a few steps we find:

$$\begin{aligned} \rho_k(\mathbf{x}) &= \int_{\mathbb{R}^2} h(\mathbf{x} - \mathbf{m}_{\mathbf{u}_k}(\mathbf{u}_k) - \xi) r'_k(\xi) d\xi \\ &= (r'_k \otimes h)(\mathbf{x} - \mathbf{m}_{\mathbf{u}_k}(\mathbf{u}_k)), \end{aligned} \tag{5}$$

where $r'_k(\mathbf{x}) = r_k(\mathbf{J}^{-1}_{\mathbf{u}_k}\mathbf{x})$ denotes a warped basis function. Thus, although the texture function is defined on an irregular grid, Equation (5) states that the resampling kernel in screen space, $\rho_k(\mathbf{x})$, can be written as a *convolution* of a warped basis function r'_k and the low-pass filter kernel h. This is essential for the derivation of screen space EWA in the next section. Note that from now on we are omitting the subscript $\mathbf{u}_k$ for $\mathbf{m}$ and $\mathbf{J}$.

3.3 Screen Space EWA

Like Greene and Heckbert [3], we choose elliptical Gaussians both for the basis functions and the low-pass filter, since they are closed under affine mappings and convolution. In the following derivation we apply these mathematical properties to the results of the previous section. This enables us to express the resampling kernel as a single Gaussian, facilitating efficient evaluation during rendering.

An elliptical Gaussian $\mathcal{G}_{\mathbf{V}}(\mathbf{x})$ with variance matrix $\mathbf{V}$ is defined as:

$$\mathcal{G}_{\mathbf{V}}(\mathbf{x}) = \frac{1}{2\pi|\mathbf{V}|^{\frac{1}{2}}} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{V}^{-1}\mathbf{x}},$$

where $|\mathbf{V}|$ is the determinant of $\mathbf{V}$. We denote the variance matrices of the basis functions r_k and the low-pass filter h with $\mathbf{V}^r_k$ and $\mathbf{V}^h$, respectively. The warped basis function and the low-pass filter are:

$$\begin{aligned} r'_k(\mathbf{x}) &= r(\mathbf{J}^{-1}\mathbf{x}) = \mathcal{G}_{\mathbf{V}^r_k}(\mathbf{J}^{-1}\mathbf{x}) = \frac{1}{|\mathbf{J}^{-1}|}\mathcal{G}_{\mathbf{J}\mathbf{V}^r_k\mathbf{J}^T}(\mathbf{x}) \\ h(\mathbf{x}) &= \mathcal{G}_{\mathbf{V}^h}(\mathbf{x}). \end{aligned}$$

The resampling kernel ρ_k of (5) can be written as a single Gaussian with a variance matrix that combines the warped basis function and the low-pass filter. Typically $\mathbf{V}^h = \mathbf{I}$, yielding:

$$\begin{aligned} \rho_k(\mathbf{x}) &= (r'_k \otimes h)(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)) \\ &= \frac{1}{|\mathbf{J}^{-1}|}(\mathcal{G}_{\mathbf{J}\mathbf{V}^r_k\mathbf{J}^T} \otimes \mathcal{G}_{\mathbf{I}})(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)) \\ &= \frac{1}{|\mathbf{J}^{-1}|}\mathcal{G}_{\mathbf{J}\mathbf{V}^r_k\mathbf{J}^T + \mathbf{I}}(\mathbf{x} - \mathbf{m}(\mathbf{u}_k)). \end{aligned} \tag{6}$$

We will show how to determine $\mathbf{J}^{-1}$ in Section 4, and how to compute $\mathbf{V}^r_k$ in Section 5. Substituting the Gaussian resampling kernel

(6) into (2), the continuous output function is the weighted sum:

$$g'_c(\mathbf{x}) = \sum_{k\in\mathbb{N}} w_k \frac{1}{|\mathbf{J}^{-1}|}\mathcal{G}_{\mathbf{J}\mathbf{V}_k^r\mathbf{J}^T+\mathbf{I}}(\mathbf{x}-\mathbf{m}(\mathbf{u}_k)). \qquad (7)$$

We call this novel formulation *screen space* EWA. Note that Equation (7) can easily be converted to Heckbert's original formulation of the EWA filter by transforming it to source space. Remember that $\mathbf{m}$ denotes the local affine approximation (4), hence:

$$\mathbf{x}-\mathbf{m}(\mathbf{u}_k) = \mathbf{m}(\mathbf{m}^{-1}(\mathbf{x})-\mathbf{u}_k) = \mathbf{J}\cdot(\mathbf{m}^{-1}(\mathbf{x})-\mathbf{u}_k).$$

Substituting this into (7) we get:

$$g'_c(\mathbf{x}) = \sum_{k\in\mathbb{N}} w_k \mathcal{G}_{\mathbf{V}_k^r+\mathbf{J}^{-1}\mathbf{J}^{-1^T}}(\mathbf{m}^{-1}(\mathbf{x})-\mathbf{u}_k). \qquad (8)$$

Equation (8) states the well known *source space* EWA method extended for irregular sample positions, which is mathematically equivalent to our screen space formulation. However, (8) involves backward mapping a point $\mathbf{x}$ from screen to the object surface, which is impractical for interactive rendering. It amounts to ray tracing the point cloud to find surface intersections. Additionally, the locations $\mathbf{u}_k$ are irregularly positioned such that the evaluation of the resampling kernel in object space is laborious. On the other hand, Equation (7) can be implemented efficiently for point-based objects as described in the next section.

4 The Surface Splatting Algorithm

Intuitively, screen space EWA filtering (7) starts with projecting a radially symmetric Gaussian basis function from the object surface onto the image plane, resulting in an ellipse. The ellipse is then convolved with a Gaussian low-pass filter yielding the resampling filter, whose contributions are accumulated in screen space. Algorithmically, surface splatting proceeds as follows:

```
for each point P[k] {
  project P[k] to screen space;
  determine the resampling kernel rho[k];
  splat rho[k];
}
for each pixel x in the frame buffer {
  shade x;
}
```

We describe these operations in detail:

Determining the resampling kernel The resampling kernel $\rho_k(\mathbf{x})$ in Equation (6) is determined by the Jacobian $\mathbf{J}$ of the 2D to 2D mapping that transforms coordinates of the local surface parameterization to viewport coordinates. This mapping consists of a concatenation of an affine viewing transformation that maps the object to camera space, a perspective projection to screen space, and the viewport mapping to viewport coordinates.

Note that in the viewing transformation we do not allow non-uniform scaling or shearing. This means we preserve the rotation invariance of our basis functions in camera space. Hence, the Jacobian of this transformation can be written as a uniform scaling matrix with scaling factor s_{mv}. Similarly, since we restrict the viewport mapping to translations and uniform scaling, we can describe its Jacobian with a scaling factor s_{vp}.

To compute the Jacobian $\mathbf{J}_{pr}$ of the perspective projection, we have to compute the local surface parameterization. After the viewing transformation, objects are given in camera coordinates that can be projected simply by division by the z coordinate. The center of projection is at the origin of camera space and the projection plane is the plane $z = 1$. The following explanations are illustrated in Figure 5. We construct a local parameterization of the object surface around a point $\mathbf{P}_k$ by approximating the surface with its tangent plane given by the normal $\mathbf{n}_k$ (transformed to camera space) at $\mathbf{P}_k$. We define the parameterization by choosing two orthogonal basis vectors $\mathbf{u}_0$ and $\mathbf{u}_1$ in the tangent plane. Since our basis functions are radially symmetric, the orientation of these vectors is arbitrary. Note that the tangent plane approximation leads to the same inconsistencies of the local parameterizations as in conventional rendering pipelines. There, the perspective mapping from texture space to screen space is determined per triangle. However, when the EWA kernel of a pixel near a triangle edge is warped to texture space, it may overlap a region of the texture that is mapped to several triangles, leading to slightly incorrect filtering of the texture. Yet, for both rendering methods, the error introduced is too small to cause visual artifacts.

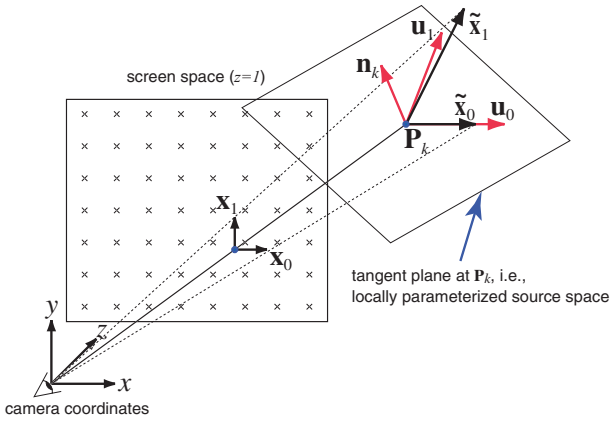

Figure 5: *Calculating the Jacobian* $\mathbf{J}_{pr}^{-1}$.

The mapping of coordinates of the local parameterization to screen coordinates is given by the perspective projection of the tangent plane to screen space. We find the Jacobian $\mathbf{J}_{pr}^{-1}$ of the inverse mapping at the point $\mathbf{P}_k$ by projecting the basis vectors of screen space $\mathbf{x}_0$ and $\mathbf{x}_1$ along the viewing ray that connects the center of projection with $\mathbf{P}_k$ onto the tangent plane. This results in the vectors $\tilde{\mathbf{x}}_0$ and $\tilde{\mathbf{x}}_1$. Specifically, we choose $\mathbf{u}_0 = \tilde{\mathbf{x}}_0/\|\tilde{\mathbf{x}}_0\|$ and construct $\mathbf{u}_1$ such that $\mathbf{u}_0, \mathbf{u}_1$, and the normal $\mathbf{n}_k$ form a right-handed orthonormal coordinate system. This simplifies the Jacobian, since $\tilde{\mathbf{x}}_0 \cdot \mathbf{u}_1 = 0$ and $\tilde{\mathbf{x}}_0 \cdot \mathbf{u}_0 = \|\tilde{\mathbf{x}}_0\|$, which is then given by:

$$\mathbf{J}_{pr}^{-1} = \begin{pmatrix} \tilde{\mathbf{x}}_0\cdot\mathbf{u}_0 & \tilde{\mathbf{x}}_1\cdot\mathbf{u}_0 \\ \tilde{\mathbf{x}}_0\cdot\mathbf{u}_1 & \tilde{\mathbf{x}}_1\cdot\mathbf{u}_1 \end{pmatrix} = \begin{pmatrix} \|\tilde{\mathbf{x}}_0\| & \tilde{\mathbf{x}}_1\cdot\mathbf{u}_0 \\ 0 & \tilde{\mathbf{x}}_1\cdot\mathbf{u}_1 \end{pmatrix},$$

where $\cdot$ denotes the vector dot product.

Concatenating the Jacobians of viewing transformation, projection, and viewport mapping, we finally get $\mathbf{J}$:

$$\mathbf{J} = s_{vp}\cdot\mathbf{J}_{pr}\cdot s_{mv}.$$

Splatting the resampling kernel First, each point $\mathbf{P}_k$ is mapped to the position $\mathbf{m}(\mathbf{u}_k)$ on screen. Then the resampling kernel is centered at $\mathbf{m}(\mathbf{u}_k)$ and is evaluated for each pixel. In other words, the contributions of all points are splatted into an *accumulation buffer*. The projected normals of the points are filtered in the same way. Besides color and normal components, each frame buffer pixel contains the sum of the accumulated contributions of the resampling kernels and camera space z values as well (see Table 1).

Although the Gaussian resampling kernel has infinite support in theory, in practice it is computed only for a limited range of the exponent $\beta(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T(\mathbf{I}+\mathbf{J}^{-1}\mathbf{J}^{-1^T})\mathbf{x}$. We choose a cutoff radius c, such that $\beta(\mathbf{x}) < c$. Bigger values for c increase image quality but also the cost of splatting the kernel. A typical choice is $c = 1$, providing good image quality at moderate splatting cost [6, 13].

Data	Storage
RGBA color components	4×4 Bytes
XYZ normal components	3×4 Bytes
Accumulated contributions	4 Bytes
Camera space z value	4 Bytes
Material index	2 Bytes
Total per pixel:	38 Bytes

Table 1: *Data storage per frame buffer pixel.*

Because the resampling kernels are truncated to a finite support, an additional normalization by the sum of the accumulated contributions is required, yielding the final pixel value:

$$g(\mathbf{x}) = \sum_{k \in \mathbb{N}} w_k \frac{\rho_k(\mathbf{x})}{\sum_{j \in \mathbb{N}} \rho_j(\mathbf{x})}. \tag{9}$$

Since the pixel grid in screen space is regular, the kernel can be evaluated efficiently by forward differencing in a rectangular bounding box and using lookup tables.

In general, the depth complexity of a scene is greater than one, thus a mechanism is required that separates the contributions of different surfaces when they are splatted into the frame buffer. Consequently, the z value of the tangent plane at $\mathbf{P}_k$ is computed at each pixel that is covered by the kernel, which can be done by forward differencing as well. This is similar to the visibility splatting approach of [15]. To determine whether a new contribution belongs to the same surface as is already stored in a pixel, the difference between the new z value and the z value stored in the frame buffer is compared to a threshold. If the difference is smaller than the threshold, the contribution is added to the pixel. Otherwise, given that it is closer to the eye-point, the data of the frame buffer is replaced by the new contribution.

Deferred shading The frame buffer is shaded after all points of a scene have been splatted. This avoids shading invisible points. Instead, each pixel is shaded using the filtered normal. Parameters for the shader are accessed via an index to a table with material properties (see Table 1). Advanced pixel shading methods, such as reflection mapping, can be easily implemented as well.

5 Texture Acquisition

In this section we address the problem of *pre-computing* the texture coefficients w_k and the basis functions r_k of the continuous texture function in (1).

Determining the basis functions As in Section 3.2, the basis functions r_k are Gaussians with variance matrices $\mathbf{V}_k^r$. For each point $\mathbf{P}_k$, this matrix has to be chosen appropriately in order to match the local density of points around $\mathbf{P}_k$. In some applications, we can assume that the sampling pattern in the local planar area around $\mathbf{u}_k$ is a jittered grid with sidelength h in object space. Then a simple solution to choose $\mathbf{V}_k^r$ is:

$$\mathbf{V}_k^r = \begin{pmatrix} \frac{1}{h^2} & 0 \\ 0 & \frac{1}{h^2} \end{pmatrix},$$

which scales the Gaussian by h. For example in the Surfel system [15], h is globally given by the object acquisition process that samples the positions $\mathbf{u}_k$. Another possibility is to choose h as the maximum distance between points in a small neighborhood. This value can be pre-computed and stored in a hierarchical data structure as in [16].

Computing the coefficients We distinguish between two different settings when computing the coefficients w_k:

1. *Objects with per point color.* The object acquisition method provides points with per point color samples.

2. *Texture mapping point-based objects.* Image or procedural textures from external sources are applied to a given point-sampled geometry.

Objects with per point color Many of today's imaging systems, such as laser range scanners or passive vision systems [12], acquire range and color information. In such cases, the acquisition process provides a color sample c_k with each point. We have to compute a continuous approximation $f_c(\mathbf{u})$ of the unknown original texture function from the irregular set of samples c_k.

A computationally reasonable approximation is to normalize the basis functions r_k to form a partition of unity, i.e., to sum up to one everywhere. Then we use the samples as coefficients, hence $w_k = c_k$, and build a weighted sum of the samples c_k:

$$f_c(\mathbf{u}) = \sum_{k \in \mathbb{N}} c_k \hat{r}_k(\mathbf{u} - \mathbf{u}_k) = \sum_{k \in \mathbb{N}} c_k \frac{r_k(\mathbf{u} - \mathbf{u}_k)}{\sum_{j \in \mathbb{N}} r_j(\mathbf{u} - \mathbf{u}_j)},$$

where $\hat{r}_k$ are the normalized basis functions. However, these are rational functions, invalidating the derivation of the resampling kernel in Section 3.2. Instead, we normalize the resampling kernels, which are warped and band-limited basis functions. This normalization does not require additional computations, since it is performed during rendering, as described in Equation (9).

Texture mapping of point-based objects When an image or procedural texture is explicitly applied to point-sampled geometry, a mapping function from texture space to object space has to be available at pre-processing time. This allows us to warp the continuous texture function from texture space with coordinates $\mathbf{s}$ to object space with coordinates $\mathbf{u}$. We determine the unknown coefficients w_k of $f_c(\mathbf{u})$ such that $f_c(\mathbf{u})$ optimally approximates the texture.

From the samples c_i and the sampling locations $\mathbf{s}_i$ of the texture, the continuous texture function $c_c(\mathbf{s})$ is reconstructed using the reconstruction kernel $n(\mathbf{s})$, yielding:

$$c_c(\mathbf{s}) = \sum_i c_i n(\mathbf{s} - \mathbf{s}_i) = \sum_i c_i n_i(\mathbf{s}).$$

In our system, the reconstruction kernel is a Gaussian with unit variance, which is a common choice for regular textures. Applying the mapping $\mathbf{u} = \mathbf{t}(\mathbf{s})$ from texture space to object space, the warped texture function $\tilde{f}_c(\mathbf{u})$ is given by:

$$\tilde{f}_c(\mathbf{u}) = c_c(\mathbf{t}^{-1}(\mathbf{u})) = \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})).$$

With $\tilde{f}_c(\mathbf{u})$ in place, our goal is to determine the coefficients w_k such that the error of the approximation provided by $f_c(\mathbf{u})$ is minimal. Utilizing the L_2 norm, the problem is minimizing the following functional:

$$\begin{aligned} F(\mathbf{w}) &= \|\tilde{f}_c(\mathbf{u}) - f_c(\mathbf{u})\|_{L_2}^2 \\ &= \|\textstyle\sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})) - \sum_k w_k r_k(\mathbf{u} - \mathbf{u_k})\|_{L_2}^2, \end{aligned} \tag{10}$$

where $\mathbf{w} = (w_j)$ denotes the vector of unknown coefficients. Since $F(\mathbf{w})$ is a quadratic function of the coefficients, it takes its minimum at $\nabla F(\mathbf{w}) = 0$, yielding a set of linear equations. After some algebraic manipulations, detailed in Appendix A, we find the linear

system $\mathbf{Rw} = \mathbf{c}$. The elements of the matrix $\mathbf{R}$ and the vector $\mathbf{c}$ are given by the inner products:

$$(\mathbf{R})_{kj} =< r_k, r_j > \quad \text{and}$$
$$(\mathbf{c})_k = \sum_i c_i \langle r_k, n_i \circ \mathbf{t}^{-1} \rangle. \tag{11}$$

We compare this optimization method with a view-independent EWA approach, similar as proposed in [4, 15]. Our novel technique is a generalization of view-independent EWA, which can be derived from (11) by means of the simplifying assumption that the basis functions r_k are orthonormal. In this case, the inner products are given by $\langle r_k, r_j \rangle = \delta_{kj}$, where $\delta_{kj} = 1$ if $k = j$ and $\delta_{kj} = 0$ otherwise. Consequently, $\mathbf{R}$ is the identity matrix and the coefficients are determined as in EWA filtering by:

$$w_k = \sum_i c_i \langle r_k, n_i \circ \mathbf{t}^{-1} \rangle.$$

In Figure 6, we show a checkerboard texture that was sampled to an irregular set of points. On the left, we applied our optimized texture sampling technique. On the right, we used view-independent EWA. In the first row, the textures are rendered under minification, which does not reveal the difference between the continuous texture functions, since the filter used for rendering is dominated by the band-limiting step. In the second row, however, magnification clearly illustrates that optimized texture sampling produces a much sharper approximation of the original texture. In the third row, we use extreme magnification to visualize the irregular pattern of points, depicted in the middle.

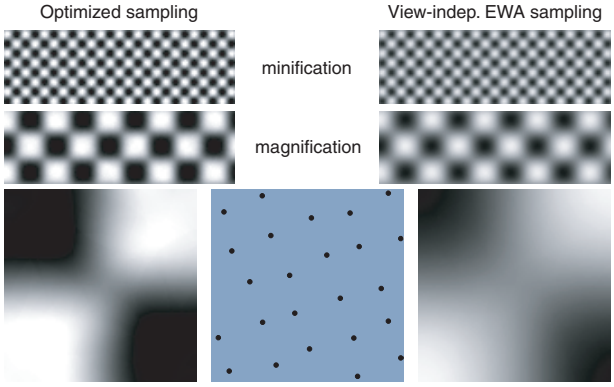

Figure 6: *Left: optimized texture sampling. Right: view-independent EWA. Bottom middle: Irregular grid of points in the area shown on the left and right.*

6 Transparency

The basic algorithm described in Section 4 can be easily extended to handle transparent surfaces as well. Our approach provides order-independent transparency using a single rendering pass and a fixed amount of frame buffer memory.

The general idea is to use a frame buffer that consists of several layers, each containing the data listed in Table 1. A layer stores a *fragment* at each pixel. The purpose of a fragment is to collect the contributions of a single surface to the pixel. After all points have been splatted, the fragments are blended back-to-front to produce the final pixel color.

We adopt the strategy presented in [8], which avoids the disadvantages of both multi-pass (e.g., [20]) and basic A-buffer (e.g., [1]) algorithms. Providing a small fixed number l of fragments per pixel, fragments are merged whenever the number of fragments exceeds the preset limit l. We apply the same rendering procedure as described in Section 4, where the *splatting* is extended as follows:

Splatting the resampling kernel In contrast to the single layered frame buffer of Section 4, the frame buffer now contains several layers, each storing a fragment per pixel. Each contribution that is splatted into a pixel is processed in three steps:

1. *Accumulate-or-Separate Decision.* Using a z threshold as described in Section 4, all fragments of the pixel are checked to see if they contain data of the same surface as the new contribution. If this is the case, the contribution is added to the fragment and we are done. Otherwise, the new contribution is treated as a separate surface and a temporary fragment is initialized with its data.

2. *New Fragment Insertion.* If the number of fragments including the temporary fragment is smaller than the limit l, the temporary fragment is copied into a free slot in the frame buffer and we are done.

3. *Fragment Merging.* If the above is not true, then two fragments have to be merged. Before merging, the fragments have to be shaded.

When fragments are merged, some information is inevitably lost and visual artifacts may occur. These effects are minimized by using an appropriate merging strategy. Unfortunately, the situation is complicated by the fact that a decision has to be taken as the scene is being rendered, without knowledge about subsequent rendering operations. The main criterion for merging fragments is the difference between their z values. This reduces the chance that there are other surfaces, which are going to be rendered later, that lie between the two merged surfaces. In this case, incorrect back-to-front blending may introduce visual artifacts.

Before fragments can be merged, their final color has to be determined by shading them. Shaded fragments are indicated by setting their accumulated weight (see Table 1) to a negative value to guarantee that they are shaded exactly once. The color and alpha values of the front and back fragment to be merged are $\mathbf{c}_f, \alpha_f$ and $\mathbf{c}_b, \alpha_b$, respectively. The color and alpha values $\mathbf{c}_o, \alpha_o$ of the merged fragment are computed using:

$$\mathbf{c}_o = \mathbf{c}_f \alpha_f + \mathbf{c}_b \alpha_b \left(1 - \alpha_f\right)$$
$$\alpha_o = \alpha_f + \alpha_b \left(1 - \alpha_f\right). \tag{12}$$

Similar to Section 4, fragments are shaded if necessary in a second pass. After shading, the fragments of each pixel are blended back-to-front as described in Equation (12) to produce the final pixel color.

Figure 7 shows a geometric object consisting of semi-transparent, intersecting surfaces, rendered with the extended surface splatting algorithm. In most areas, the surfaces are blended flawlessly back-to-front. The geometry of the surfaces, however, is not reconstructed properly around the intersection lines, as illustrated in the close-up on the right. In these regions, contributions of different surfaces are mixed in the fragments, which can cause visual artifacts. On the other hand, in areas of high surface curvature the local tangent plane approximation poorly matches the actual surface. Hence, it may happen that not all contributions of a surface are collected in a single fragment, leading to similar artifacts. Essentially, both cases arise because of geometry undersampling. We can avoid the problem by increasing the geometry sampling rate or by using a higher order approximation of the surface. However, the latter is impracticable to compute for interactive rendering.

7 Edge Antialiasing

In order to perform edge antialiasing, information about partial coverage of surfaces in fragments is needed. For point-based representations, one way to approximate coverage is to estimate the density

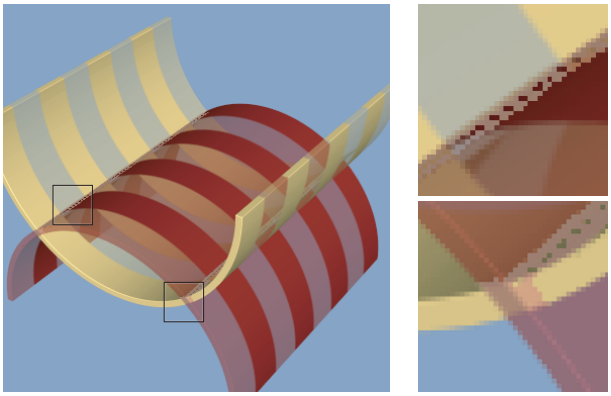

Figure 7: *Geometric object with intersecting, semi-transparent surfaces, rendered with the extended surface splatting algorithm and edge-antialiasing.*

of projected points per pixel area [10]. Coverage is then computed by measuring the actual density of points in a fragment and dividing the measured value by the estimated value.

Rather than explicitly calculating this estimation, we make the simplifying assumption that the Gaussian basis functions are located on a regular grid and have unit variance. In this case, they approximate a partition of unity. In other words, we assume that they sum up to one at any point. Warping and band-limiting this constant function results in a constant function again. Therefore the sum q of the resampling kernels is approximately one at any point, specifically in all fragments $\mathbf{x}$:

$$q = \sum_{k \in \mathbb{N}} \rho_k(\mathbf{x}) \approx 1.$$

If q is smaller than one in a fragment, this indicates that the texture does not completely cover the pixel and q can be used as a coverage coefficient.

For an irregular grid, the approximation of the partition of unity becomes less reliable. Furthermore, the Gaussians are truncated to a finite support. With a cutoff radius $c = 1$ (see Section 4), we found that a threshold $\tau = 0.4$ for indicating full coverage produces good results in general. The coverage q' of a pixel is $q' = q/\tau$. We implement edge antialiasing by multiplying the alpha value α of a fragment with its coverage coefficient. The final alpha value α' of the fragment is $\alpha' = \alpha \cdot q'$ if $q' < 1$, otherwise $\alpha' = \alpha$.

8 Results

We implemented a point-sample rendering pipeline based on surface splatting in software. Furthermore, we can convert geometric models into point-based objects in a pre-processing step. Our sampler generates a hierarchical data structure similar to [15], facilitating multiresolution and progressive rendering. It applies view-independent EWA texture filtering to sample image textures onto point objects. We implemented the optimized texture sampling technique discussed in Section 5 in Matlab.

Figure 1, left, shows a face that was rendered using a point cloud acquired by a laser range scanner. Figure 1, middle and right, show point-sampled geometric objects. We illustrate high quality texturing on terrain data and semi-transparent surfaces on the complex model of a helicopter.

Table 2 shows the performance of our unoptimized C implementation of surface splatting. The frame rates were measured on a 1.1 GHz AMD Athlon system with 1.5 GByte memory. We rendered to a frame buffer with three layers and a resolution of 256×256 and 512×512 pixels, respectively. A pixel needs $3 \times 38 = 114$ bytes of storage. The entire frame buffer requires 6.375 MB and 25.5 MB of memory, respectively.

Data	**# Points**	**256×256**	**512×512**
Scanned Head	429075	1.3 fps	0.7 fps
Matterhorn	4782011	0.2 fps	0.1 fps
Helicopter	987552	0.6 fps	0.3 fps

Table 2: *Rendering performance for frame buffer resolutions 256 $\times$ 256 and 512 $\times$ 512.*

The texture quality of the surface splatting algorithm is equivalent to conventional source space EWA texture quality. Figure 8, top and second row, compare screen space EWA and source space EWA on a high frequency texture with regular sampling pattern. Note that screen space EWA is rendered with edge antialiasing and there was no hierarchical data structure used. Moreover, the third row illustrates splatting with circular Gaussians, similar to the techniques of Levoy [10] and Shade [17]. We use the major axis of the screen space EWA ellipse as the radius for the circular splats, which corresponds to the choices of [10] and [17]. This leads to overly blurred images in areas where the texture is magnified.

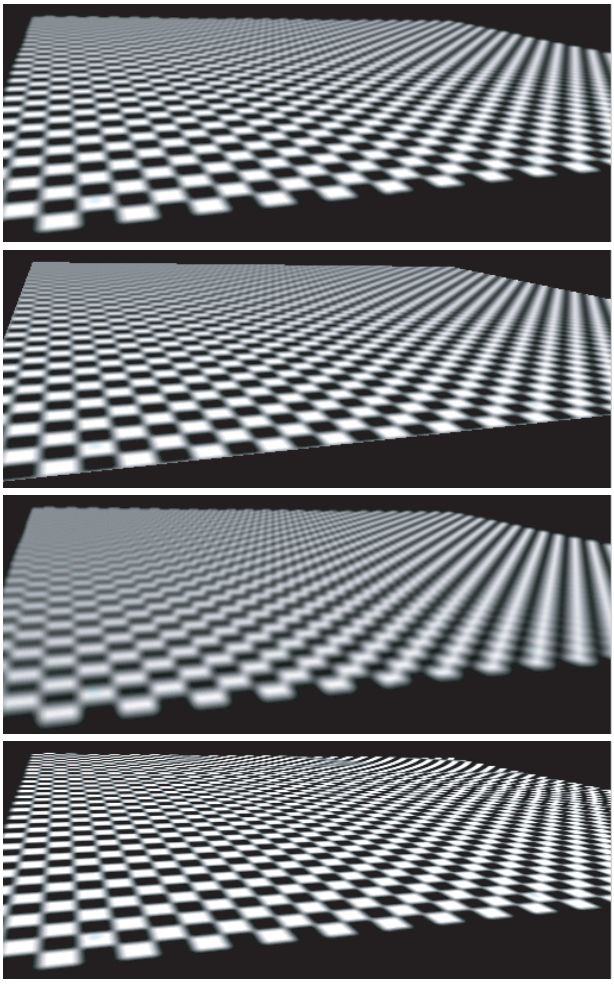

Figure 8: *Top row: screen space EWA. Second row: source space EWA. Third row: circular splats. Bottom: elliptical splats.*

In case of minification, the circular splats approximate the screen space EWA ellipses more closely, leading to better filtering. The bottom row shows splatting with elliptical Gaussians that are determined using the normal direction of the surface as discussed in [16]. This amounts to omitting the band-limiting step of EWA, which causes aliasing artifacts in regions where the texture is minified. In contrast to these methods, screen space EWA provides a continuous transition between minification and magnification and renders high quality textures in both cases.

9 Conclusions

Surface splatting is a new algorithm that makes the benefits of EWA texture filtering accessible to point-based surface representations and rendering techniques. We have provided a thorough mathematical analysis of the process of constructing a continuous textured image from irregular points. We have also developed an optimized texture sampling technique to sample image or procedural textures onto point-based objects. Surface splatting provides stable textures with no flickering during animations. A modified A-buffer and simple merging strategy provides transparency and edge-antialiasing with a minimum of visual artifacts.

We will apply surface splatting to procedurally generated objects, such as parametric surfaces or fractal terrain. Rendering the generated points in the order of computation should yield high performance. We think it is possible to extend surface splatting to render volumetric objects, such as clouds, fire, and medical CT scans. This will require extending the screen space EWA framework to 3D spherical kernels. By rendering voxels in approximate front-to-back order we could use our modified A-buffer without undue increase of the number of fragments to be merged per pixel. Because of the simplicity of the surface splatting algorithm we are investigating an efficient hardware implementation. Increasing processor performance and real-time hardware will expand the utility of this high quality point-rendering method.

10 Acknowledgments

We would like to thank Ron Perry for many helpful discussions, and Jennifer Roderick and Martin Roth for proof-reading the paper. The textured terrain data set in Figure 1 is courtesy of *Bundesamt für Landestopographie*, Bern, Switzerland.

References

[1] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics*, volume 18 of *SIGGRAPH 84 Proceedings*, pages 103–108. July 1984.

[2] B. Curless and M. Levoy. A Volumetric Method for Building Complex Models from Range Images. In *Computer Graphics*, SIGGRAPH 96 Proceedings, pages 303–312. New Orleans, LA, August 1996.

[3] N. Greene and P. Heckbert. Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter. *IEEE Computer Graphics & Applications*, 6(6):21–27, June 1986.

[4] J. P. Grossman and W. Dally. Point Sample Rendering. In *Rendering Techniques '98*, pages 181–192. Springer, Wien, Vienna, Austria, July 1998.

[5] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics & Applications*, 6(11):56–67, November 1986.

[6] P. Heckbert. *Fundamentals of Texture Mapping and Image Warping*. Master's thesis, University of California at Berkeley, Department of Electrical Engineering and Computer Science, June 17 1989.

[7] H. Hoppe, T. DeRose, T. Duchampt, J. McDonald, and W. Stuetzle. Surface Reconstruction from Unorganized Points. In *Computer Graphics*, SIGGRAPH 92 Proceedings, pages 71–78. Chicago, IL, July 1992.

[8] N. Jouppi and C. Chang. $\mathbb{Z}^3$: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. In *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware 1999*, pages 85–93. Los Angeles, CA, August 1999.

[9] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D Scanning of Large Statues. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 131–144. Los Angeles, CA, July 2000.

[10] M. Levoy and T. Whitted. The Use of Points as Display Primitives. Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985.

[11] T. W. Mark, L. McMillan, and G. Bishop. Post-Rendering 3D Warping. In *1997 Symposium on Interactive 3D Graphics*, pages 7–16. ACM SIGGRAPH, April 1997.

[12] W. Matusik, C. Buehler, R. Raskar, S. Gortler, and L. McMillan. Image-Based Visual Hulls. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 369–374. Los Angeles, CA, July 2000.

[13] J. McCormack, R. Perry, K. Farkas, and N. Jouppi. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. In *Computer Graphics*, SIGGRAPH '99 Proceedings, pages 243–250. Los Angeles, CA, August 1999.

[14] K. Mueller, T. Moeller, J.E. Swan, R. Crawfis, N. Shareef, and R. Yagel. Splatting Errors and Antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):178–191, April-June 1998.

[15] H. Pfister, M. Zwicker, J. van Baar, and M Gross. Surfels: Surface Elements as Rendering Primitives. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 335–342. Los Angeles, CA, July 2000.

[16] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Computer Graphics*, SIGGRAPH 2000 Proceedings, pages 343–352. Los Angeles, CA, July 2000.

[17] J. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered Depth Images. In *Computer Graphics*, SIGGRAPH 98 Proceedings, pages 231–242. Orlando, FL, July 1998.

[18] J. E. Swan, K. Mueller, T. Moeller, N. Shareef, R. Crawfis, and R. Yagel. An Anti-Aliasing Technique for Splatting. In *Proceedings of the 1997 IEEE Visualization Conference*, pages 197–204. Phoenix, AZ, October 1997.

[19] L. Westover. Footprint Evaluation for Volume Rendering. In *Computer Graphics*, Proceedings of SIGGRAPH 90, pages 367–376. August 1990.

[20] S. Winner, M. Kelley, B. Pease, B. Rivard, and A. Yen. Hardware Accelerated Rendering of Antialiasing Using a Modified A-Buffer Algorithm. pages 307–316, August 1997.

[21] G. Wolberg. *Digital Image Warping*. IEEE Computer Society Press, Los Alamitos, California, 1990.

Appendix A: Mathematical Framework

The L_2 Norm of equation (10) can be computed using inner products, and exploiting the linearity of the operator we obtain:

$$\begin{aligned} F(\mathbf{w}) &= \| \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})) - \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j)\|^2_{L_2} \\ &= \langle \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})), \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u}))\rangle \\ &+ \langle \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j), \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j)\rangle \\ &- 2\langle \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u})), \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j)\rangle. \end{aligned}$$

We minimize $F(\mathbf{w})$ by computing the roots of the gradient, i.e.:

$$\nabla F(\mathbf{w}) = \left(\dots \frac{\partial F}{\partial w_k} \dots \right)^T = 0.$$

The partial derivatives $\frac{\partial F}{\partial w_k}$ are given by:

$$\begin{aligned} \frac{\partial F(\mathbf{w})}{\partial w_k} &= \frac{\partial}{\partial w_k} \sum_i \sum_j w_i w_j \langle r_i(\mathbf{u}-\mathbf{u}_i), r_j(\mathbf{u}-\mathbf{u}_j)\rangle \\ &- 2\frac{\partial}{\partial w_k} \langle \sum_j w_j r_j(\mathbf{u}-\mathbf{u}_j), \sum_i c_i n_i(\mathbf{t}^{-1}(\mathbf{u}))\rangle \\ &= 2\sum_j w_j \langle r_j(\mathbf{u}-\mathbf{u}_j), r_k(\mathbf{u}-\mathbf{u}_k)\rangle \\ &- 2\sum_i c_i \langle r_k(\mathbf{u}-\mathbf{u}_k), n_i(\mathbf{t}^{-1}(\mathbf{u}))\rangle = 0. \end{aligned}$$

This set of linear equations can be written in matrix form:

$$\underbrace{\begin{pmatrix} \ddots & & \\ & \langle r_k, r_j \rangle & \\ & & \ddots \end{pmatrix}}_{\mathbf{R}} \underbrace{\begin{pmatrix} \vdots \\ w_j \\ \vdots \end{pmatrix}}_{\mathbf{w}} = \underbrace{\begin{pmatrix} \vdots \\ \sum_i c_i \langle r_k, n_i \cdot \mathbf{t}^{-1} \rangle \\ \vdots \end{pmatrix}}_{\mathbf{c}}.$$