# Hardware Accelerated Rendering of Points as Surfaces

Fritz M. Gschwantner*
Vienna University of Technology

## Abstract

*Over the last 7 years point-based geometries gained more and more interest as a research area, since it is a valuable alternative to surface representations based on polygonal meshes. Especially hardware accelerated methods evolved over the years with increasing efficiency and image quality, along with the current development in graphics hardware.*

*This paper presents the recent development of hardware accelerated point based rendering like surface splatting, ray-tracing and level of detail culling. It focuses on the use of surface splatting, explaining the basic principles of every step to fast, high-quality splatting on today's GPUs.*

**Keywords:** splatting, raycasting, point sets

## 1 Introduction

The use of points as display primitives was first introduced by [Levoy and Whitted 1985] as an intermediate representation between modelling and rendering of objects. Instead of customizing rendering algorithms for different geometric representations of objects, point sets can be retrieved from those surfaces and those point set surfaces can be rendered through a standardized algorithm.

[Levoy and Whitted 1985] developed an algorithm based on texture mapping to render such point sets as a continuous, three dimensional surface. In their pioneer work for point based rendering they used different textures as their data source for *spatial* and *non-spatial* attributes. Points described by those textures are sent through the rendering pipeline. For every transformed point the contribution to each display pixel is calculated. This is done by overlaying each pixel with a Gaussian filter.

In Texture Mapping the resulting color through each texel's contribution to one pixel is divided through the weighted sums. [Levoy and Whitted 1985] introduced a method to calculate a factor for the density of source points in order to compute the normalizing factor for arbitrary source points in any viewing transformation. This factor is computed through the *tangent plane* of a surface on a specific point. The tangent plane of a point $\mathbf{p}_0$ after the viewing transformation is spanned by two vectors $\mathbf{u}$ and $\mathbf{v}$ wich are defined by two points $\mathbf{p}_u$ and $\mathbf{p}_v$ which are one unit away from $\mathbf{p}_0$ in $u$ and $v$ (cf. Fig. 1).

The area of parallelogram spanned by these vectors is inversely proportional to the densitiy of source points and can be computed through the absolute value of the determinant of the Jacobian matrix.

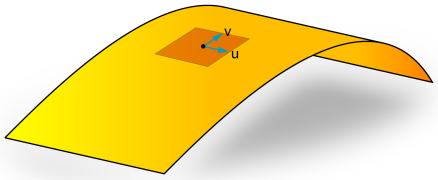


Figure 1: Tangent plane to surface in small neighborhood.)

So the algorithm is taking a point and a tangent plane and transforming them into image space, which results in a point in screen space and a measurement of the area which the point would cover if it were a *surface element*. This principle is also used to a further extent in the *Surface Splatting* approach, introduced by [Zwicker et al. 2001], which is described in section 2.1 of this paper.

Over the years, different rendering methods have been evolved, such as various forms and implementations of *splatting* and *raycasting*.

Due to the increasing complexity of polygonal meshes in various real time graphic applications, the development of hardware accelerated rendering of point set surfaces got more attention in the recent past. Conventional hardware accelerated rendering methods of triangles get increasingly inefficient when more than one whole triangle falls into a single raster element.

This paper offers a step-by-step introduction to hardware accelerated rendering methods of point set surfaces.

## 2 Exposition

### 2.1 Surface Splatting

#### 2.1.1 Principle

Surface Splatting has first been introduced by [Zwicker et al. 2001]. For each point elliptical disks are rendered in object space. By making these splats overlap each other, a hole-free representation of the point set surface is guaranteed. In order to prevent aliasing artefacts and shading discontinuities, [Zwicker et al. 2001] used a modified implementation of EWA texture filtering by [Greene and Heckbert 1986] for high quality anisotropic filtering and anti-aliasing.

While [Levoy and Whitted 1985] acted on the assumption of regulary spaced points, the algorithm in [Zwicker et al. 2001] can be used for irregularly spaced point sets. Therefore each point is represented by its position in three-dimensional space and at least a normal vector, alongside other non-spatial attributes like color.

The idea is to recreate the surface using a weighted sum of radially symmetric basis functions. Each point is associated with a radially

*e-mail: email@spikx.net

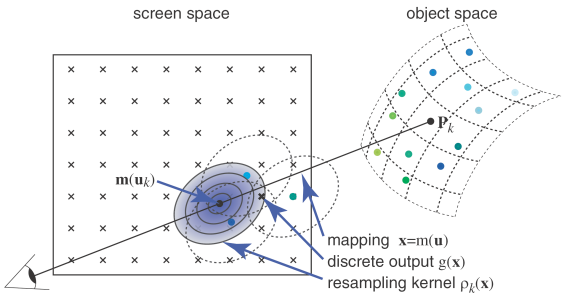


Figure 2: Rendering by surface splatting. Resampling kernels are accumulated in screen space.)

symmetric basis function $r_k$ and coefficients $w_k$ that represent continuous functions for red, green, and blue color components. Those functions are determined in a pre-processing step. The continuous surface function $f_c(u)$ is defined as follows:

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

[Zwicker et al. 2001] are describing the rendering process as a resampling problem in order to use Heckbert's EWA texture filter. Heckbert's annotation is adopted so it can be applied for irregularly spaced basis functions. The function is warped to screen space, convolved with a filter $h$ for band limitations and then sampled to produce a discrete output. This represents the *surface splatting* approach (cf. Fig. 2). The resulting equation is further derived to a screen space resampling kernel, where $p_k(x)$ for a mapped point $m_{uk}$ is

$$p_k(x) = (r'_k \otimes h)(x - m_{uk}(u_k)). \tag{2}$$

Then an elliptical Gaussian $Gv(x)$ with a variance matrix V is introduced. Substituting the Gaussian resampling kernel with equation (1) yields this equation:

$$g'_c(x) = \sum_{k \in \mathbb{N}} w_k \frac{1}{|J^{-1}|} G_{JV'_k J^T + I}(x - m(u_k))$$

To determine the resampling kernel for each projected point, its Jacobian matrix **J** has to be computed. As mentioned in [Levoy and Whitted 1985] (see section 1) this is done by determining the local surface parametrization by choosing two orthogonal tangent vectors. In [Zwicker et al. 2001] these vectors are determined through the normal vector of the point.

When a point is finally splatted, it is projected into screen space. The resampling kernel will be centered on the screen space position and evaluated for each pixel. The contributions of each point for each pixel will be stored in an *accumulation buffer*[1]. In theory, the Gaussian resampling kernel has an infinite range, thus the contribution of a point could affect all pixels in screen space, but it is sufficient to implement a cutoff radius. Of course this cutoff radius affects the overall image quality, higher values would increase the image quality, but also increase computation time for each splat. Due to this filter cutoff a *normalization* by the sum of accumulated contributions *for each pixel* is required.

[Zwicker et al. 2001] also proposed to use *deferred shading*, a technique which recently also has become more attention in real time graphics applications. After all points have been splatted, the final pixels are shaded using the filtered normal, circumventing the unnecessary computations of invisible splats.

[1] A-buffer, see [Carpenter 1984]

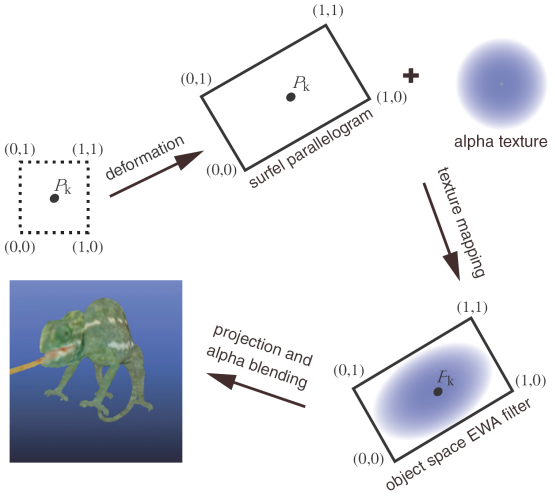


Figure 3: EWA splatting using textured polygons.)

### 2.1.2 Object Space EWA Surface Splatting

The first hardware accelerated approach was implemented by [Ren et al. 2002], by formulating the resampling filter as a function on a parameterized surface in *object space*. Then the surface is projected to screen space by the graphics hardware.

In order to do that, equation (2) is rearranged, yielding an object space resampling filter $p'_k(u)$ defined in coordinates $u$ of the local surface parameterization. As in [Zwicker et al. 2001] a Gaussian basis function and a prefilter is added. This yields the following equation for a continuous output function (see equation (1)) of an object space EWA resampling filter:

$$g_c(x) = \sum_{k \in \mathbb{N}} w_k G f_{V'_k + J_k^{-1} J_k^{-1^T}}(m^{-1}(x) - u_k))$$

**Rendering** The rendering process is divided into two passes. The first pass is called *visibility splatting*. In this pass a quad is rendered at each surfel, perpendicular aligned to the surfel's normal. The side length of a quad is chosen as the doubled maximum distance between surfels in a small neighborhood. The output is passed to the Z-Buffer only.

When the splats are rasterized in the second pass, pixels are discarded or accumulated by comparing the depth value of the splat with the depth image. This requires to translate the depth image backwards along the viewing rays by a depth treshold $z_t$ to prevent false disavowal of pixel contributions.

The actual surface splatting is done in a second rendering pass, using textured polygons that represent a discrete form of the object space EWA resampling filter. A single alpha texture is used, that encodes a discrete Gaussian basis function. For each surfel, a textured quad is drawn, stretched and scaled (cf. Fig. 3). So the filter itself will be determined through the vertex position, since the texture is stretching and scaling with the quad through texture mapping. During splatting, each pixel is compared with the stored $z$ values, but the depth values aren't updated. So every splat within the depth range $z_t$ is blended together.

In order to determine the object space EWA resampling filter we again need the Jacobian matrix of a point $u$. It is calculated through

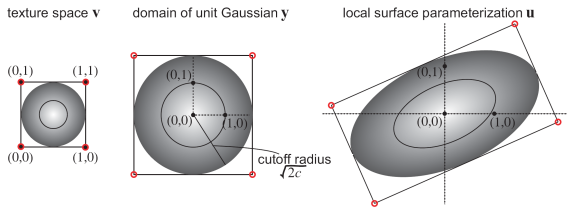


Figure 4: Construction of the EWA resampling filter: Mapping of texture coordinates to Gaussian domain (left to middle) using equation (5), rotation and scaling of vertices in screen space (middle to right) using equation (3).)

the local surface parametrization spanned by two orthogonal vectors $s$ and $t$. For screen space mapping a scale factor $\eta$ is required (determined by the view frustum). Then the EWA resampling filter is defined as

$$p'_k(u) = G_{Mk}(u) \quad \text{where} \quad M_k = V_k^r + J_k^{-1} {J_k^{-1}}^T$$

This can be decomposed to a scale matrix $\Lambda$ and rotation matrix $Rot(\theta)$ through the eigenvectors and eigenvalues of $M_k$. This leads to the linear relationship

$$u = Rot(\theta) \cdot \Lambda \cdot y \tag{3}$$

and yields

$$G_{Mk}(Rot(\theta) \cdot \Lambda \cdot y) = \frac{1}{2\pi \cdot r_0 r_1} e^{-1\frac{1}{2}y^T y} = \frac{1}{r_0 r_1} G_I(y) \tag{4}$$

for $G_{Mk}$ where $y^T y = u^T {M_k}^{-1} u$. Equation (4) represents a unit Gaussian in y, which is mapped to the elliptical Gaussian resampling filter using (3). This is used to rotate and scale the unit quad, which contains the alpha texture, representing the Gaussian filter.

The camera space position of each vertex representing one single surfel is calculated through the texture coordinates $v \in \{(0,0),(0,1),(1,1),(1,0)\}$. First, the texture coordinates are mapped to the domain of the unit Gaussian $y$ using the cutoff radius $c$.

$$y = 2\sqrt{2c}\left(\begin{bmatrix} v_0 \\ v_1 \end{bmatrix} - \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}\right) \tag{5}$$

(cf. Fig. 4)

The vertices are rotated and scaled in screen space using equation (3) and then the vertex positions in camera space are computed using the local surface parameterization

$$p^c(u) = o + u_s \cdot s + u_t \cdot t$$

**Vertex Shader** The first rendering pass is implemented straight forward, while in the second rendering pass the computation for the vertex positions of each surfel polygon is implemented completely on the shader. The Vertex shader will output the constant part of equation (4) to the alpha channel, in order to be accesed through the pixel shader.

In the Vertex Shader lies also the major drawback of this method. For every surfel, the same computations have to be made 4 times in order to be applied on 4 different vertices.

**Pixel Shader** The pixel shader computes the weighted sum (1) for each pixel and every color through additive alpha blending. The alpha texture is multiplied with the constant value of the input color's alpha channel (hence the constant part of equation (4)).

**Per-Surfel Normalization** Since point sets can be positioned irregularly in space, the basis function $r_k$ will not always sum to unity, because of the cut off Gaussian function. Usually this is compensated by a *per-pixel normalization* after splatting [Zwicker et al. 2001], which was not supported by hardware at the time. So [Ren et al. 2002] used a a per-surfel normaliziation in a pre-processing step, where the sum of alpha-weights is approximated to be constant for each splat. This is only a low quality solution and will be further addressed in section 2.1.3.

**Performance** Since each surfel is represented by another 4 vertices, in order to render a splat as a textured rectangle in object-space, the number of process is multiplied by four. This slows the rendering down to about 2M - 3M splats per second. It is still an improvement to software based splatting. The method of [Zwicker et al. 2001] achieved around 1M splats per second.

The enourmous technological advance in graphics processing power made it clear, that the benefit of harware implemented point set rendering techniques will also increase over the years. Since this first hardware implementation several other hardware accelerated implementations emerged over the years, with increasing performance and image quality.

### 2.1.3 High Quality Splatting

[Botsch and Kobbelt 2003] introduced and summarized some methods to increase the image quality of hardware accelerated splatting techniques. Usually high quality splatting could only be obtained through a pure software implementation due to the limitations of hardware accelerated implementations.

**Splat Size** Computing the image space size of a splat is very expensive operation. It can be approximated through the bounding sphere of the splat and neglecting its $x$ and $y$ offsets from the optical axis.

$$size_{win} = r \cdot \frac{n}{z_{eye}} \cdot \frac{h}{t-b} \tag{6}$$

$z_{eye}$ is the distance of the splat from the camera, $r$ the splat's radius and $n,t,b,h$ are the projection parameters. This is done in the *Vertex Shader*. It transforms the vertex position representing a point and computes the eye-space normal vector. Using *point sprites* the shader can adjust the size of the square to be rasterized through equation (6).

**Splat Shape** To generate an elliptical splat (representing a disc in object space), the depth values of generated image-aligned squares are adjusted. [Botsch and Kobbelt 2003] propose to compute the depth offset $\delta_z$ of each pixel in the parallel projection of the view frustum (unit-cube $(x,y) \in [-1,1]^2$) using the eye-space normal vector of the splat.

$$\delta_z = -\frac{n_x}{n_z}x - \frac{n_y}{n_z}y$$

The depth offset can also be used to compute the distance of a pixel from the splat center in object space. The pixel corresponds to a point inside a splat if the length $\left\|(x,y,\delta_z)^T\right\|$ is less than 1. So pixels with a value greater than 1 can be discarded in the fragment shader.

**Splat Filtering** As in [Zwicker et al. 2001] a radially decreasing Gaussian weight function will be mapped onto the splat ellipse in image space, in order to filter overlapping splats. This can be simplified by using the 3D distance of a pixel to its splat center, that just has been computed to determine the splat shape. Therefore the final weight can be looked up in a simple 1D texture, containing a discrete version of the Gaussian filter, beginning from its center.

$$\alpha(x,y) = GaussTextureID\left[\left\|(x,y,\delta_z)^T\right\|\right]$$

Although using only an object space Gaussian function alone is not the way EWA splatting is done (which is combining an object-space reconstruction filter and image space band-limiting filter into one Gaussian), the results are sufficient in practice.

The fragment shader should only blend pixel using this alpha value, if their $z$-distance is significantly small. Otherwise the front splat should overwrite the splat behind. This can't be done on hardware in one pass, therefore *visibility splatting* has to be done in a first render pass, like proposed in [Ren et al. 2002], described in section 2.1.2 of this paper.

In this case a per-pixel depth correction of the splats has to be done, since they are only image aligned rectangles, where each pixel contains the same depth value as the splat center. This is done using the already computed depth offset $\delta_z$ in this equation:

$$z_{win} = \frac{a(z_{eye}+\delta_z)+b}{z_{eye}+\delta_z}$$

where $a = \frac{f}{f-n}$ and $b = \frac{-2fn}{f-n}$ are derived from the projection and viewport. The depth offset and depth correction is done in the fragment shader using the texture coordinates of the splat's point sprite.

**Per-Pixel Normalization** When all splats are accumulated in the second rendering pass additive blending is used. Therefore each pixel contains $(\sum_i \alpha_i(rgb)_i, \sum_i \alpha_i)$. To normalize the RGB values they simply have to be divided by its $\alpha$ component. In [Zwicker et al. 2001] this is done in software, after the splatting is finished. While [Ren et al. 2002] used *per-surfel normalization* due to insufficient hardware capabilities, [Botsch and Kobbelt 2003] are proposing an efficient hardware accelerated method by appending a third render pass.

The normaliziation can't be done directly in the pixel shader of the second render pass, because not all contributions are accumulated yet by the time it is executed. So instead of rendering the second pass into the framebuffer, it is rendered to an *offscreen buffer*. This buffer is then mapped as a texture on a window sized rectangle. A simple fragment shader will then perform the necessary division of the pixel's (or in this case texel's) color (cf. Fig. 5).

This technique avoids sending any pixel data over the AGP or PCIe bus as opposed to reading the framebuffer, performing the normalization on the CPU and writing the results back into the framebuffer. The time it takes to perform this normalization in a fragment shader for a simple rectangle is negligible compared to the prior rendering passes.

**Performance** By using effictive simplifications of the splatting process, computing the splats completely on the GPU and minimizing the data transfer between the CPU and GPU this implementation can render up to 30M splats per second (on a GeForceFX 5800 Ultra), providing an average image quality. Using Gaussian filtering the perforamnce drops to 5M - 10M splats per second, which is still a big improvement to software based and object space EWA surface splatting.

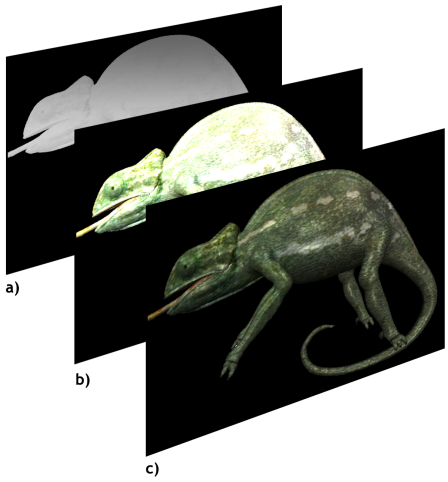


Figure 5: Output of the 3 rendering passes: a) visibility splatting, b) surface splatting, c) normalization. (Note: these are artificial images to demonstrate the principle.)

Figure 6: Comparison of different splatting methods. Left: unfiltered splats. Middle: Gaussian weighted splats. Right: Phong Splatting.

### 2.1.4 Phong Splatting

Conventional surface splatting implementations usually use only one surface normal per point or splat. This leads to a Flat shaded look of the surface, or a Gouraud shaded look, if the splats are weighted by a Gaussian filter (cf. Fig. 6). Phong shading provides a higher visual quality, by using interpolated normal vectors across triangles on polygonal meshes. In order to apply phong shading to surface splats, [Botsch et al. 2004] proposed to use a linearly varying normal field with each splat instead of a constant normal.

Perspective accurate rasterization of elliptical splats is used in this case, introduced by [Zwicker et al. 2004]. As in the methods described before, the local curvature of a surface is approximated by generating a splat on to the surface, aligned along the curvature. So each splat is defined by a center $\mathbf{c}_j$ and two tangent directions $\mathbf{u}_j$ and $\mathbf{v}_j$, which are scaled according to the ellipse radii. An arbitrary point $\mathbf{q}$ in the plane lies in the interior of the splat, if its local parameter values $u$ and $v$ satisfy the condition

$$u^2+v^2 = \left(\mathbf{u}_j^T(\mathbf{q}-\mathbf{c}_j)\right)^2 + \left(\mathbf{v}_j^T(\mathbf{q}-\mathbf{c}_j)\right)^2 \leq 1 \qquad (7)$$

The normals in the normal field used in Phong Splatting are tilted along the tangential directions of a surfel. There are scalar values $\alpha_j$ and $\beta_j$ for each tangent, the normal field $N_j$ is defined as follows:

$$N_j(u,v) = \bar{n}_j + u\alpha_j u_j + v\beta_j v_j \qquad (8)$$

The normal field (i.e. the values $\alpha$ and $\beta$) are computed in a preprocessing step. For each vertex the parameters $(u,v)$ and the normal field defined by $(\bar{n},\alpha,\beta)$ are passed to the vertex shader.

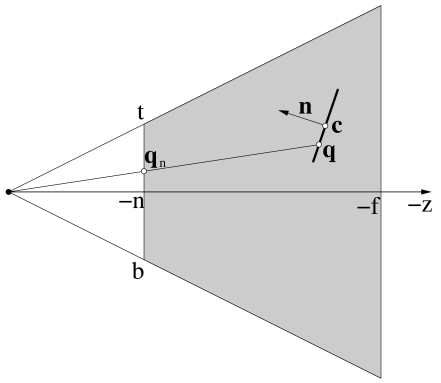


Figure 7: Computing point $\mathbf{q}$ in object space for a given window pixel by casting a ray through $\mathbf{q}_n$ on the near plane.

Creating of the surface splats is done exactly like in section 2.1.3 [Botsch and Kobbelt 2003]. But in contrast to other approaches, the exact coordinates in object space for the current pixel are computed through a ray intersection with the current splat ellipse (cf. Fig. 7). In order to compute this point $\mathbf{q}$ on the splat a point $\mathbf{q}_n$ on the near plane of the viewing frustum is determined through the inversion of the viewport transformation of the pixel position $(x,y)$. The ray intersection is calculated as follows:

$$\mathbf{q} = \mathbf{q}_n \cdot \frac{c^T n}{\mathbf{q}_n^T n}$$

$c$ and $n$ are the splat's center and normal vector in eye coordinates. The parameter values $(u,v)$ can easily be determined by equation (7). If $u^2 + v^2 \leq 1$, thus the pixel is within the splat, the normal vector is computed by equation (8). This is done in the fragment shader whereas the scaled normal $n \cdot (c^T n)^{-1}$ is already precomputed in the vertex shader.

Since the exact depth $\mathbf{q}_z$ of a point on the splat is known in eye-coordinates, the fragment's depth value can be calculated with per-pixel accuracy. Other approaches like [Botsch and Kobbelt 2003] described in section 2.1.3 only used an approximation to the projective mapping.

**Performance** Phong Splatting is capable of computing 6M splats per second. While this may seem a little low, phong splatting provides a better image quality at the same performance than previous aproaches. Most performance is lost due to the computation of the normals and the per-pixel lighting. Also more data has to be committed to the shader.

### 2.1.5 Deferred Shading

Using hardware accelerated *deferred shading* is an old idea made new. [Zwicker et al. 2001] achieved per-pixel lighting by using multiple render buffers into which they splat normal vectors and material properties. This leads to smoothly interpolated and averaged normals and colors in image space. The final pass would compute the lighting based on the accumulated normal vectors and surface materials for each pixel. Since this had to be done in software, performance was naturally slow.

While Phong Splatting is able to deliver high-quality shading by associating a linear normal field with each splat, this approach is limited to static geometries, because the parameters for the normal field have to be precomputed.

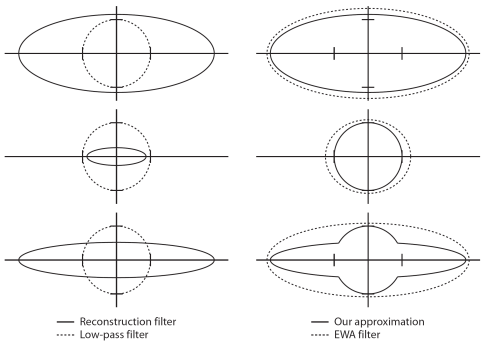


Figure 9: Comparison of the original EWA filter (left) and the approximation.

The recent generations of video cards made it possible to completely compute the per-pixel deferred shading process on the GPU. They provide floating point precision in all stages of the rendering pipeline and it is possible to use multiple render targets in one pass. The principle of deferred shading is also very easy to understand, since it provides a clear separation between the splat rasterization and the actual lighting or shading computations. Also it is relatively easy to implement.

In the context of surface splatting, 3 rendering passes are required (cf. Fig. 8):

- **Visibility Pass:** as described earlier in this paper, this pass is required to accurately blend or overwrite surface splats in the following attribute pass, depending on their $z$-value.
- **Attribute Pass:** splatting is done as usual. The difference is, that no actual shading is done here. Instead, the accumulated colors and normals are stored in 2 different output buffers.
- **Shading Pass:** for the final shading pass, a window-size rectangle is drawn, similar to the method used for per-pixel normalization in section 2.1.3. The fragment shader can then compute the correct, shaded color value with the information of the previous render passes, passed as 3 different textures.

Without deferred shading many unnecessary computations are made for pixels, that aren't actually visible or not yet fully accumulated. Due to the mutual overlap of individual splats, the number of lighting computations would be higher by a factor of 6 to 10 for typical datasets without deferred shading [Botsch et al. 2005].

### 2.1.6 EWA Approximation

Usually a complete EWA filter consists of an object-space reconstruction kernel and a band-limiting screen-space filter. Since the required computations are quite complex, many hardware accelerated approaches of surface splatting are using only the reconstruction filter. This can lead to heavy aliasing artefacts in case of extreme minifications (size of projected splat falls below one pixel). [Botsch et al. 2005] proposed a simple and efficient way to approximate the EWA screen-space filter. By making the size of projected splats to be at least $2 \times 2$ pixels it is guaranteed that enough fragments are generated for antialiasing purposes. The pixel shader is adjusted to compute 2 radii. The 3D distance from equation (7) and a 2D distance of the current fragment from the splat center.

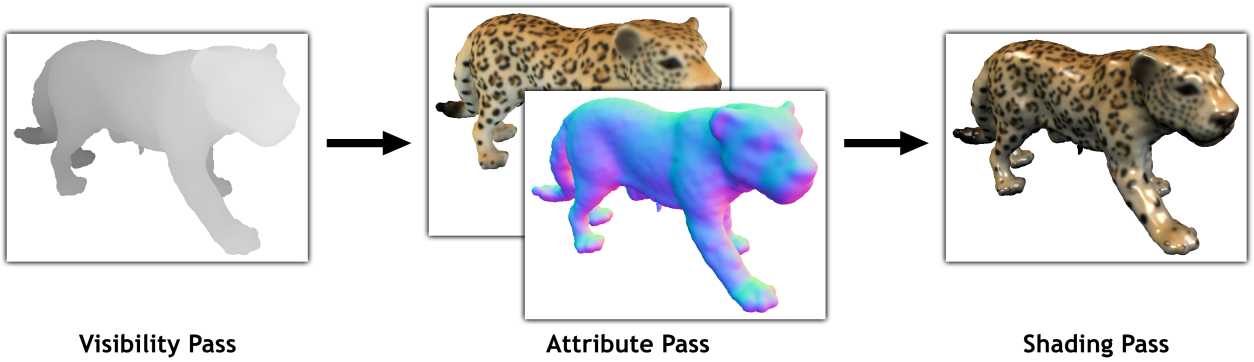


Figure 8: Deferred shading pipeline.

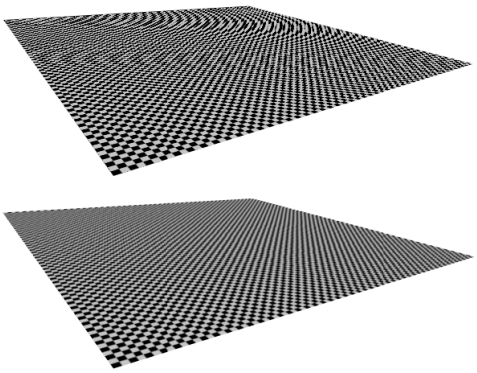

Figure 10: Top: minification errors due to missing screen space EWA filter. Bottom: usage of approximated EWA filter.

The approximation is essentially the union of the low-pass and the reconstruction filter (cf. Fig. 9). A fragment will be accepted if it falls within this union. This approximation provides high-quality anti-aliasing in magnified and minified regions of the surface (cf. Fig. 10).

Since the minimum size causes more fragments to be computed, especially for complex models with small projected splat sizes, the use of deferred shading, adressed in the previous section, becomes even more important. [Botsch et al. 2005] reported a performance of 25M splats per second. The performance decrease when using the approximated EWA filter is only about 3M splats per second.

## 2.2 Level of Detail in Surface Splatting

### 2.2.1 Principle

The idea behind level of detail is to reduce the complexity of rendered geometry when viewed from far away, since the visual difference is not noticable between e.g. one single rendered triangle or multiple triangles for only one pixel. Many methods to reduce triangles of polygonal meshes have been developed in the past, where unnecessarily small triangles are merged to larger ones. However, in most cases this is done at the expense of complex precomputations, high CPU load for on-the-fly retriangulations or popping artefacts when switching between discrete detail levels.

Adopting a Level Of Detail algorithm for point based geometry can be done easily, since these representations completely lack topological information. [Dachsbacher et al. 2003] developed a technique to create a fluent Level Of Detail processing completely on the GPU while also using hardware accelerated surface splatting.

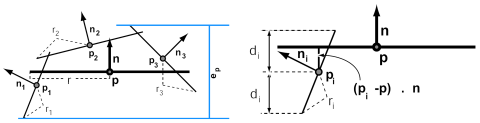

Figure 11: Calculation of the perpendicular error of the parent node's disk.

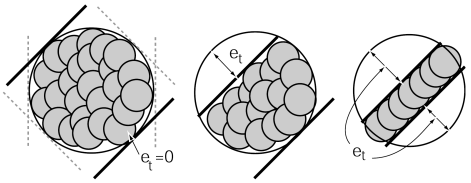

Figure 12: Calculation of the perpendicular error of the parent node's disk.

At first a point tree hierarchy of the point set is computed, by inserting these samples into an octree. For each node normal values and positions of the child nodes are averaged and stored in the node. Also the radius of the surfel representing this node is calculated. So every node is an approximated surface element of its children. There are two values, that reflect the approximation error of this element.

The *perpendicular error* $e_p$ is the minimum distance between two planes parallel to the disk, that encloses all child disks (cf. Fig. 11). This error depends on the view vector $v$ and the disk normal $n$. It decreases with the viewing distance $r$ and the algorithm will make sure, that errors along the silhouette are less acceptable.

The *tangential error* $e_t$ measures how well the parent disk encloses its child disks (cf. Fig. 12). It is calculated if there is an unnecessary large area covered by the parent disk, where no child disks are present.

Those errors can be combined to a single *geometric error* $e_g = \max_\alpha \{e_p \sin\alpha + e_t \cos\alpha\} = (e_p^2 + e_t^2)^{\frac{1}{2}}$. The error in image space $\tilde{e}_g$ depends on the viewing distance $r$, but not on the viewing angle, which is a simplification that is faster to compute.

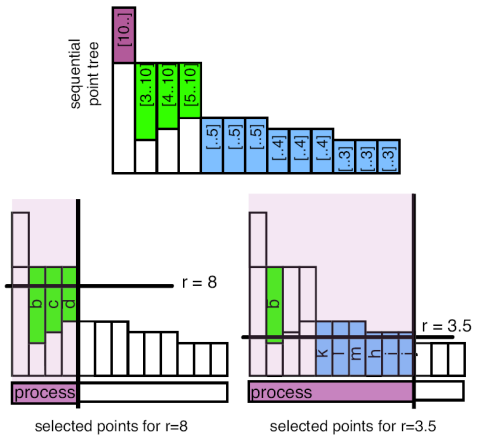


Figure 13: The sequential point tree.

When an object is rendered, the tree would be traversed, computing the image error $\tilde{e}$ for every node. If $\tilde{e}$ is above an error threshold $\varepsilon$ and the node is not a leaf, the children are traversed recursively. Otherwise a splat will be drawn, using the attributes of this node.

### 2.2.2 Sequential Point Tree

The recursive rendering method mentioned in the previous section can't be implemented on the GPU. [Dachsbacher et al. 2003] proposed a technique to sequentialize the tree data, replacing the recursive rendering by a sequential loop over a list.

This is done by formulating a minimum distance $r_{min}$ and a maximum distance $r_{max}$ for every node. $r_{min}$ indicates the distance where the error $\tilde{e}$ will still be beyond the treshold. If the viewing distance falls below $r_{min}$ of a node, child nodes have to be rendered instead. $r_{max}$ indicates the maximum distance at which child nodes are being used for rendering. If the viewing distance is above $r_{max}$ of a node, the parent node can be rendered instead.

A list of all nodes and leaves is created and sorted by $r_{max}$. The CPU will preselect a range of this list by searching an entry with $r_{max} \leq r$. The vertex shader will do a $[r_{min}, r_{max}]$ test for each vertex. Points that pass the test are rendered using the computed splat size, points that fail are culled by moving them to infinity (no fragments will be generated).

This is explained in Fig. 13. The top image represents the calculated, sequential point tree. When $r = 8$ (bottom left image), the CPU will preselect points 0 to 3. The GPU will cull the first point, because its minimum distance is above the current viewing distance. In the case of $r = 3.5$ (bottom right image), more splats will be rendered and some parent nodes will be left out by the GPU. [Dachsbacher et al. 2003] achieved a performance of 50M splats per second with this technique, while maintaining a low CPU load.

## 2.3 Ray Traycing Point Set Surfaces

### 2.3.1 Principle

Ray Tracing is still one of the preferred methods to render scenes with reflective or transmissive surfaces in high quality. Although it is primarily used for non-interactive rendering, parallelization and

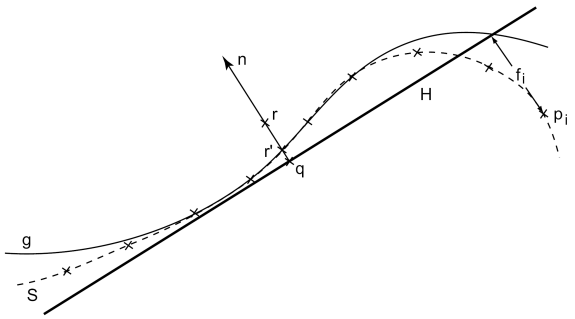


Figure 14: Projecting a point r on the surface, using a polynomial approximation in a reference domain H.

efficient spatial data structures can enhance the performance of ray tracing algorithm in order to be interactively used.

[Adamson and Alexa 2003] developed an algorithm to apply ray tracing for point set surfaces. In order to provide accurate high quality ray traycing, a surface definition is needed. Since point set surfaces are only arbitrarily placed points in 3D space to a certain extent, a specific surface definition has to be made. An other approach would be to extract the surface from the point set in a preprocessing step.

The surface $S$ is being defined by points of the point set $S_p$ that project onto themselves. In order to project a point $r$ a local approximation of the surface is computed by using a bivariate polynomial $g$ in a reference domain (cf. Fig. 14). The reference domain is determined by minimizing the weighted distance of points to a plane $H$.

Since the polynomial approximates the surface within a neighborhood of the projected point $r'$, the intersection of a ray and the polynomial is close to the intersection of the ray and the point set surface, assuming the point $r$ is in the neighborhood of the point set. To find the point $r$ on the ray, the ray is interesected with several spheres, enclosing the point set surface.

The ray-surface intersection algorithm is schown in Fig. 15:

- **top:** The ray is intersected with the bounding spheres of the point set surface, beginning with the nearest one.
- **middle:** By projecting the center of the bounding-sphere $r_0$ onto itself, the first polynomial surface approximation is determined (this can be done in a pre-processing step for all bounding spheres, or it can be stored once it is computed on the fly, for the intersection of other rays). Then the ray is intersected with the polynomial. In this case no intersection is found within the bounding sphere, so the algorithm goes on to the next bounding sphere.
- **bottom:** In this case the ray is intersecting the polynomial of the bounding sphere's center within the bounding sphere itself. The intersection $r_1$ will be projected on the surface. If the distance between $r_1$ and the new projected point $r_1'$ is smaller than a certain threshold $\varepsilon$, the ray-surface intersection is found. Otherwise a new polynomial for the projected point will be calculated.

Usually 2 - 3 iterations have to be made, in order to find an intersection close enough to the point set surface.

**Results** This method of ray-tracing point set surfaces achives smooth, accurate images of high quality. It is noticable that even

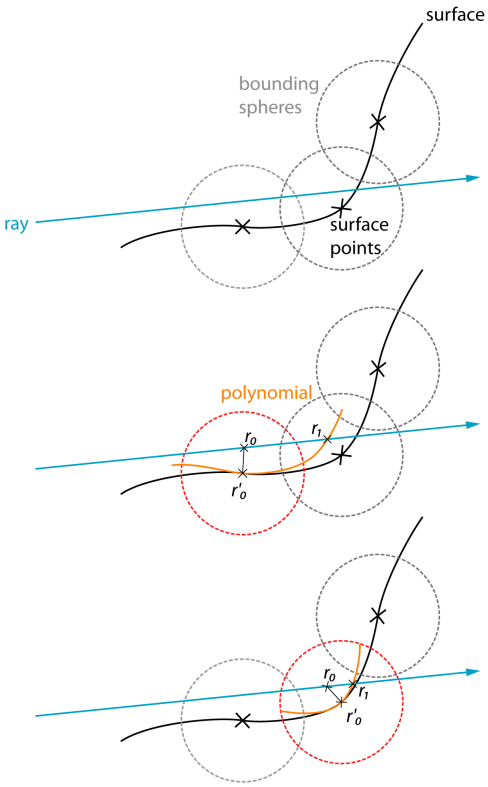


Figure 15: ray-surface intersection algorithm

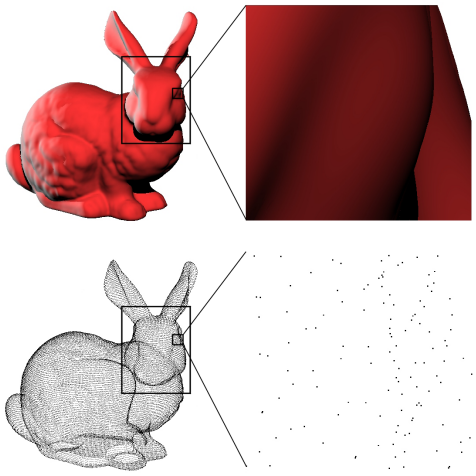

Figure 16: Ray traced point set surface of the Stanford Bunny.

for very sparse point information the resulting image of the surface is smooth (cf. Fig. 16 [Adamson and Alexa 2003]).

### 2.3.2 Hardware Acceleration

The introduction of Shader Model 3.0 on recent GPUs made dynamic loops and conditional branching (alongside other new features) available in the shader stage. This made it possible to use ray-casting and ray-tracing algorithms on the GPU alone which enables real-time performance for volume rendering and the rendering of ray-traced objects.

[Tejada et al. 2006] are presenting a technique to render point set surfaces through ray-casting completely on the GPU, using a complex multi-pass render system. There are 6 render passes in total, which will be summarized in the following part of this paper.

**Initial approximation** This is only a pre-processing step where the local polynomial approximations are calculated for each sample point. This is done in the pixel shader, by rendering a single fragment for each point and using a 3D texture to pass the neighbours of the sample point. The resulting coefficients are rendered to 32-bit float texture for further use.

**Intersection** In this pass, the first, nearest intersection of the ray with the point set surface is determined. For each point, viewport aligned discs are rendered. Each fragment of the discs calculates the intersection of the ray with the precomputed polynomial of the corresponding sample point (cf. Fig. 17). Only the fragment with the nearest intersection is kept (using depth tests). This interesection $r$ is stored in a float texture for the next render passes.

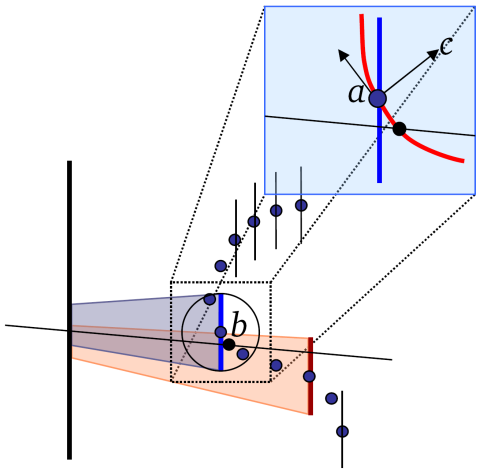


Figure 17: Calculating the intersection of the ray with the local approximation stored in each sample point.

**Form covariance matrix** In order to find the normal vector of point $r$, the covariance matrix has to be computed. For this step the same discs as in the previous pass are rendered. Each fragment generated this way calculates its distance to the intersection point on the ray passing through it in order to ensure that it is in the neighborhood of the intersection. The results of the fragments in the neighborhood of $r$ are accumulated using blending. These results are stored in 3 different 16-bit float textures that hold the $3 \times 3$ matrix.

**Find normals** In this step a window sized quad is rendered and the normal for each pixel is computed through the eigenvector associated to the smallest eigenvalue of the matrix generated in the previous step. The output is stored in a float texture.

**Form system for polynomial fitting** The polynomial approximation at interesection $r$ is computed through the normals calculated in the previous step. Discs are rendered again and each fragment formes a linear system, providing the coefficients for the polynomial. This time the calculated value is twofold, a $4 \times 4$ matrix and a vector of size 4. These results are accumulated to four float textures which are used in the next step.

**Solve linear system and find projection** A window sized quad is rendered again in order to solve the linear system of the previous step for every fragment. The intersection $r$ will be projected onto the new polynomial. As described in the ray-tracing algorithm in section 2.3.1 the distance between the new projected point and $r$ is calculated. If it is smaller than a threshold $\varepsilon$, the intersection has been found. Otherwise the intersection of the ray with the new polynomial will be determined. If it is within the bounding sphere of the corresponding sample point, a new iteration has to be made for this ray. This information is stored in a float texture, in order to use it in the next iteration.

The next iteration will start in the step *Intersection*. If the result of the last iteraion is valid, all the following steps will be performed using this intersection as the new point $r$.

**Performance** Tested on an NVidia GeForce 6800 GT video card, this hardware implementation of ray-tracing delivers about 6.6 frames per second on the Standford Bunny dataset, using 2 iterations for each ray.

## 3 Conclusion

The algorithms and techniques for hardware accelerated rendering of point set surfaces got more and more efficient and sophisticated over the last years. While [Zwicker et al. 2001] layed the ground work for efficient, high-quality surface splatting for irregularely placed points, [Ren et al. 2002] provided the first mathematical framework for an object space EWA filter, which is rather complex. Following approaches like [Botsch and Kobbelt 2003] omitted parts of the full object space EWA filtering method, to increase rendering speed, which is done completely on the GPU. The development in graphics hardware made it possible, to decrease the complexity of the rendering framework, while increasing the image quality, using deferred shading for example [Botsch et al. 2005]. Combining this with other techniques like sequential point trees of [Dachsbacher et al. 2003] can offer a comparatively high performance for rendering high-quality surface splats. But as pointed out by [Kobbelt and Botsch 2004], splats still cannot keep up with triangles, since current hardware is highly optimized for that (although point based geometries can deliver a more accurate representation of an object). Therefore [Weyrich et al. 2007] are actually proposing a hardware architecture for point based rendering.

Also ray-tracing and ray-casting were rarely used hardware accelerated until the recent developments and availability of Shader Model 3.0. [Tejada et al. 2006] shows how a complex rendering algorithm like ray-tracing for point sets can be done completely on the GPU, showing a good performance increase compared to software implementations. The render algorithm is quite complex, using multiple passes and render targets.

## References


ADAMSON, A., AND ALEXA, M., 2003. Ray tracing point set surfaces.

BOTSCH, M., AND KOBBELT, L., 2003. High-quality point-based rendering on modern gpus.

BOTSCH, M., SPERNAT, M., AND KOBBELT, L. 2004. Phong splatting.

BOTSCH, M., HORNUNG, A., ZWICKER, M., AND KOBBELT, L. 2005. High-quality surface splatting on today's gpus.

CARPENTER, L. 1984. The a-buffer, an antialiased hidden surface method. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 103–108.

DACHSBACHER, C., VOGELGSANG, C., AND STAMMINGER, M. 2003. Sequential point trees. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM Press, New York, NY, USA, 657–662.

GREENE, N., AND HECKBERT, P. S. 1986. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter.

KOBBELT, L., AND BOTSCH, M., 2004. A survey of pointbased techniques in computer graphics.

LEVOY, M., AND WHITTED, T. 1985. The use of points as a display primitive.

REN, L., PFISTER, H., AND ZWICKER, M., 2002. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering.

TEJADA, E., GOIS, J., NONATO, L. G., CASTELO, A., AND ERTL, T. 2006. Hardware-accelerated Extraction and Rendering of Point Set Surfaces. In *Proceedings of EUROGRAPHICS - IEEE VGTC Symposium on Visualization*, 21–28.

WEYRICH, T., HEINZLE, S., AILA, T., FASNACHT, D. B., OETIKER, S., BOTSCH, M., FLAIG, C., MALL, S., ROHRER, K., FELBER, N., KAESLIN, H., AND GROSS, M. 2007. A hardware architecture for surface splatting. *ACM Transactions on Graphics (Proc. SIGGRAPH) 26*, 3 (Aug.).

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface splatting. In *SIGGRAPH 2001, Computer Graphics Proceedings*, ACM Press / ACM SIGGRAPH, E. Fiume, Ed., 371–378.

ZWICKER, M., RSNEN, J., BOTSCH, M., DACHSBACHER, C., AND PAULY, M., 2004. Perspective accurate splatting.