Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering

Submitted December 2001.
Final Version May 2002.

EUROGRAPHICS 2002 / G. Drettakis and H.-P. Seidel
(Guest Editors)

Volume 21 (2002), Number 3

Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering

Liu Ren†
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
USA

Hanspeter Pfister‡
MERL
Mitsubishi Electric Research Laboratories
Cambridge, MA 02139
USA

Matthias Zwicker§
Department of Computer Science
ETH Zürich
8092 Zürich
Switzerland

Abstract

Elliptical weighted average (EWA) surface splatting is a technique for high quality rendering of point-sampled 3D objects. EWA surface splatting renders water-tight surfaces of complex point models with high quality, anisotropic texture filtering. In this paper we introduce a new multi-pass approach to perform EWA surface splatting on modern PC graphics hardware, called object space EWA splatting. We derive an object space formulation of the EWA filter, which is amenable for acceleration by conventional triangle-based graphics hardware. We describe how to implement the object space EWA filter using a two pass rendering algorithm. In the first rendering pass, visibility splatting is performed by shifting opaque surfel polygons backward along the viewing rays, while in the second rendering pass view-dependent EWA prefiltering is performed by deforming texture mapped surfel polygons. We use texture mapping and alpha blending to facilitate the splatting process. We implement our algorithm using programmable vertex and pixel shaders, fully exploiting the capabilities of today's graphics processing units (GPUs). Our implementation renders up to 3 million points per second on recent PC graphics hardware, an order of magnitude more than a pure software implementation of screen space EWA surface splatting.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display Algorithms

1. Introduction

Point-based surface models define a 3D surface by a dense set of sample points. Point-rendering algorithms reconstruct a continuous image of the sampled surface for display. The points on the surface are commonly called *surface elements* or *surfels* to indicate their affinity with picture elements (pixels) and volume elements (voxels). A point-based representation has advantages for geometries with complex topology, in situations where connectivity information is not required, or for fusion of data from multiple sources [11, 16]. Surfel models can be acquired directly using 3D scanning techniques [10] or by conversion from polygon models with textures [13].

Most point-rendering algorithms to date have focused on efficiency and speed [7, 13]. Some of them use OpenGL and hardware acceleration [14, 15], achieving interactive rendering performances of two to five million points per second. However, none of these algorithms supports antialiasing for models with complex surface textures. Recently, Zwicker et al. [16] introduced elliptical weighted average (EWA) surface splatting, the only point-rendering method with anisotropic texture filtering to date. The algorithm uses a screen space formulation of the EWA texture filter [8, 5] adapted for irregular point samples. However, a non-optimized software implementation of EWA surface splatting only achieves a rendering performance of up to 250,000 points per second [16].

In this paper, we propose a new formulation of EWA surface splatting called *object space EWA surface splatting*. It computes the EWA resampling filter in object space and can efficiently be implemented on modern graphics processing units (GPUs). It achieves a performance of up to three million points per second with high image quality and texture antialiasing. We anticipate that rapid increases in GPU speed

† Email: liuren@cs.cmu.edu
‡ Email: pfister@merl.com
§ Email: zwicker@inf.ethz.ch

will widen the performance gap to CPU-based implementations even further.

After a discussion of previous work in the next section, we review the screen space formulation of EWA surface splatting in Section 3. We then introduce object space EWA surface splatting in Section 4. We will show in Section 5 how the object space EWA filter can be efficiently implemented on graphics hardware using a new multi-pass approach. Our method also takes advantage of programmable vertex and pixel shaders, as shown in Section 6. Section 7 describes two alternative methods to normalize the resampling kernels for continuous surface reconstruction. Finally, we discuss results and performance in Section 8.

2. Previous Work

Levoy and Whitted [11] proposed points as an alternative primitive for the display of continuous surfaces. They discuss fundamental issues such as surface reconstruction, visibility, and rendering of transparent surfaces. Grossman and Dally [6, 7] and Pfister et al. [13] build on these ideas by presenting efficient point-rendering systems for models with complex geometry. Alexa et al. [1] and Kalaiah and Varshney [9] use methods of differential geometry to render and modify point-based surfaces. Although very efficient, none of these software based systems achieves interactive frame rates for objects with more than 200,000 points.

The Stanford QSplat system by Rusinkiewicz and Levoy [14] was developed to display the highly complex models of the Digital Michelangelo Project [14]. It uses OpenGL and hardware acceleration, achieving a rendering performance of 2.7 million points per second. Stamminger and Drettakis [15] use standard OpenGL point primitives to render point-sampled procedural geometry. Their implementation achieves about 5 million points per second. Both methods do not handle textured models.

Two recent papers [3, 4] combine polygon and point primitives into efficient rendering systems that choose one or the other based on screen space projection criteria. Both systems make use of current graphics hardware and achieve real-time performance for reasonably complex models. However, neither system handles surface textures, and the introduction of connectivity information diminishes the advantages of pure point based models.

Our goal is to design a rendering system that interactively renders complex point models with arbitrary surface textures at the highest possible quality. We also want to take advantage of the advances in PC graphics hardware, namely, the ever increasing performance of GPUs and programmable shading [12]. To provide anisotropic texture filtering, we base our approach on screen space EWA surface splatting [16]. In the next section we review the mathematical framework of screen space EWA surface splatting and introduce our new object space formulation in Section 4.

3. Screen Space EWA Splatting

In the screen space EWA splatting framework [16], objects are represented by a set of irregularly spaced points $\{P_k\}$ in three dimensional object space without connectivity. Each point 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. The basis functions r_k are reconstruction filters defined on locally parameterized domains, hence they define a continuous texture function on the surface represented by the set of points. As illustrated in Figure 1, the color of any

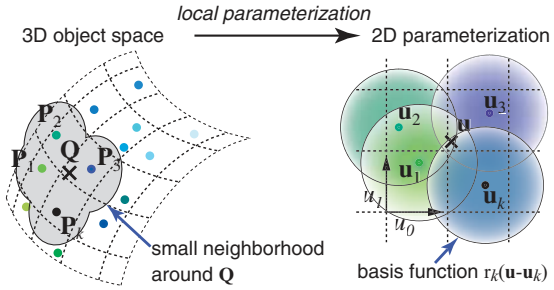

Figure 1: Defining a texture function on the surface of a point-based object.

point Q with local coordinates $\mathbf{u}$ is evaluated by accumulating the basis functions r_k, yielding a continuous texture function $f_c(\mathbf{u})$ as the weighted sum

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

where $\mathbf{u}_k$ is the local coordinate of point P_k.

In the ideal resampling framework introduced by Heckbert [8], rendering the texture function $f_c(\mathbf{u})$ yields a continuous output function $g_c(\mathbf{x})$ in screen space that respects the Nyquist criterion of the output pixel grid, thus avoiding aliasing artifacts. The rendering process includes the following conceptual steps: First, $f_c(\mathbf{u})$ is warped to screen space using a local affine mapping of the perspective projection at each point. Then the continuous screen space signal is band-limited by convolving it with a prefilter h, yielding the output function $g_c(\mathbf{x})$ where $\mathbf{x}$ are screen space coordinates. After rearranging the mathematical expressions as detailed in [8, 16], we can write the output function as a weighted sum of *screen space resampling filters* $\rho_k(\mathbf{x})$:

$$g_c(\mathbf{x}) = \sum_{k \in N} w_k \rho_k(\mathbf{x}), \tag{2}$$

where

$$\rho_k(\mathbf{x}) = (r'_k \otimes h)(\mathbf{x} - m_k(\mathbf{u_k})). \tag{3}$$

Here, the resampling filter $\rho_k(\mathbf{x})$ is written as a convolution of a *warped basis function* $r'_k(\mathbf{x}) = r_k(m^{-1}(\mathbf{x}))$ and the prefilter $h(\mathbf{x})$. To simplify the evaluation of ρ_k, at each point

$\mathbf{u}_k$ we use the local affine approximation $\mathbf{x} = m_k(\mathbf{u})$ of the projective mapping $\mathbf{x} = m(\mathbf{u})$ from the local surface parameterization to screen space. The local affine approximation m_k is given by the Taylor expansion of m at $\mathbf{u}_k$, truncated at the linear term:

$$m_k(\mathbf{u}) = \mathbf{x}_k + J_k \cdot (\mathbf{u} - \mathbf{u}_k), \tag{4}$$

where $\mathbf{x}_k = m(\mathbf{u}_k)$ and the Jacobian $J_k = \frac{\partial m}{\partial u}(\mathbf{u}_k) \in \mathbb{R}^{2\times 2}$.

In the EWA framework, elliptical Gaussians are chosen as basis functions r_k and as prefilter h because of their unique mathematical properties. Gaussians are closed under affine mappings and convolution, hence the resampling filter ρ_k can be expressed as an elliptical Gaussian as shown below. A 2D elliptical Gaussian $G_V(\mathbf{x})$ with variance matrix $V \in \mathbb{R}^{2\times 2}$ is defined as

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

where $|V|$ is the determinant of V. We denote the variance matrices of the basis functions r_k and the low-pass filter h with V_k^r and V^h, hence $r_k = G_{V_k^r}$ and $h = G_{V^h}$, respectively. Note that a typical choice for the variance of the low-pass filter is the identity matrix I. By substituting the Gaussian basis function and prefilter in Equation (3), we get a Gaussian resampling filter

$$\rho_k(\mathbf{x}) = \frac{1}{|J_k^{-1}|} G_{J_k V_k^r J_k^T + I}(\mathbf{x} - m_k(\mathbf{u_k})), \tag{5}$$

which is called the *screen space EWA resampling filter*. For more details on its derivation, please refer to [16].

4. Object Space EWA Splatting

In Equation (3), we have expressed the resampling filter as a function of screen space coordinates $\mathbf{x}$, which is suitable for software implementations such as [16]. However, today's graphics engines do not allow us to compute such a filter directly in screen space. To make EWA splatting amenable to acceleration by graphics hardware, we formulate the resampling filter as a function on a parameterized surface in object space. Then, we exploit graphics hardware to project the surface to screen space, yielding the resampling filter in screen space as in (3).

We rearrange (3) using the local affine approximation m_k:

$$\begin{aligned}
\rho_k(\mathbf{x}) &= (r'_k \otimes h)(\mathbf{x} - m_k(\mathbf{u_k})) \\
&= (r'_k \otimes h)(m_k(m_k^{-1}(\mathbf{x})) - m_k(\mathbf{u_k})) \\
&= (r'_k \otimes h)(J_k(m_k^{-1}(\mathbf{x}) - \mathbf{u_k})) \\
&= (r_k \otimes h')(\mathbf{u} - \mathbf{u_k}) = \rho'_k(\mathbf{u}),
\end{aligned} \tag{6}$$

yielding the *object space resampling filter* $\rho'_k(\mathbf{u})$ defined in coordinates $\mathbf{u}$ of the local surface parameterization. Note that in contrast to the screen space resampling filter, the object space resampling filter consists of the convolution of an *original basis function* $r_k(\mathbf{u})$ and a *warped prefilter* $h'_k(\mathbf{u}) = |J_k| h(J_k(\mathbf{u}))$. As illustrated in Figure 2a, the conversion between $\rho_k(\mathbf{x})$ and $\rho'_k(\mathbf{u})$ corresponds to the projection from object to screen space.

Similar to Section 3, we use Gaussians as basis functions and prefilter in (6). This yields an analogous expression to (5), which we call the *object space EWA resampling filter*:

$$\rho'_k(\mathbf{u}) = G_{V_k^r + J_k^{-1} J_k^{-1\,T}}(\mathbf{u} - \mathbf{u}_k). \tag{7}$$

Finally, we use Equations (6) and (7) to reformulate the continuous output function (2) as a weighted sum of object space EWA resampling filters:

$$g_c(\mathbf{x}) = \sum_{k \in N} w_k G_{V_k^r + J_k^{-1} J_k^{-1\,T}}(m^{-1}(\mathbf{x}) - \mathbf{u}_k). \tag{8}$$

Figure 2b depicts a Gaussian resampling filter in the local surface parameterization and its counterpart in screen space, and Figure 2c shows the corresponding warping and resampling of a checkerboard texture.

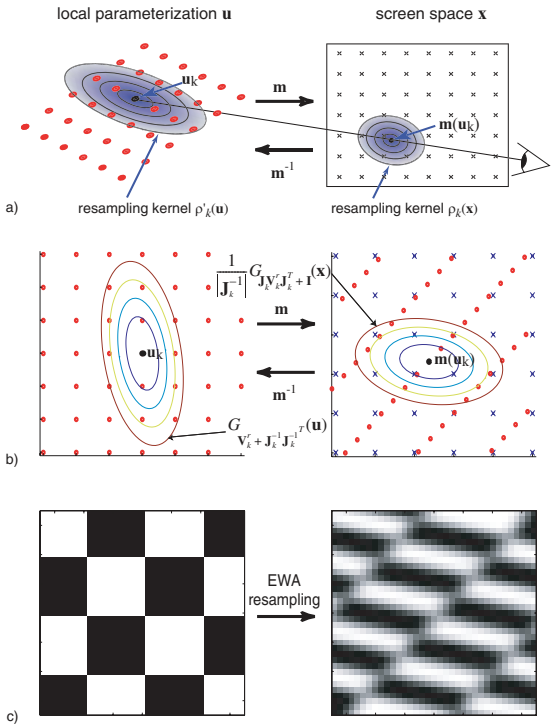

Figure 2: Object space and screen space resampling filters: a) Conversion between object space and screen space resampling filter. b) Object space and screen space EWA resampling filters. c) Checkerboard texture resampled using the EWA resampling filter.

5. Hardware Accelerated Rendering

Our hardware accelerated surface splatting algorithm is based on Equation (8) and uses a two-pass approach, emulating an A-Buffer [2]. The first pass (Section 5.1) performs visibility splatting [13] by rendering an opaque polygon for each surfel into the Z-buffer. The second pass (Section 5.2) implements Equation (8) as follows: First we set up the object space EWA resampling filter as a polygon with a semi-transparent alpha texture. On modern GPUs we can implement this using programmable vertex shaders. Then the projection of the textured polygon to screen space yields the screen space EWA resampling filter, which we also call *EWA splat*. The splats are evaluated at pixel centers, multiplied with the color w_k^r, w_k^g, w_k^b of the current surfel, and the resulting values are accumulated in each pixel. During rasterization, we perform depth tests using the data in the Z-buffer that was generated in the first rendering pass to determine whether the splats are visible. This ensures that in each pixel only those splats that represent the surface closest to the viewer are accumulated.

The QSplat rendering system [14] also proposes a two pass algorithm for point-based rendering using semi-transparent splats, although without antialiasing. Both algorithms use a textured polygon to represent a surfel. However, in their approach the vertex positions of a surfel polygon in object space are static, i.e, determined before rendering. In contrast, we compute view-dependent vertex positions on the fly during rendering to avoid aliasing, as described in Section 5.2.

5.1. Visibility Splatting

The purpose of visibility splatting is to render a depth image of the object into the Z-buffer such that it does not contain any holes [13], as illustrated in Figure 3. The depth image will be used to control the accumulation of the semi-transparent splats in the second rendering pass, as described below. We center an opaque quad at each surfel P_k, perpendicular to the normal $\mathbf{n}_k$ of the surfel (Figure 3a). The quad is rasterized into the Z-buffer only, without modifying other frame buffer attributes(Figure 3b). To avoid holes in the depth image, the side length of the quad is chosen as $2h$, where h is the maximum distance between surfels in a small neighborhood around P_k. To render a point-based object without artifacts, we must accumulate all the splats of the visible surface closest to the viewer while discarding all other splats. During rasterization of the splats, we decide for each pixel whether to discard or accumulate the current contribution by comparing the depth value of the splat with the depth image that was generated as explained above. However, to prevent contributions of the visible surface from being accidentally discarded, the depth image should be translated away from the viewpoint by a small depth threshold z_t. A simple solution is to translate the depth image by z_t *along the z-axis* in camera space, as proposed by [14]. However, this leads to occlusion artifacts as shown in Figure 4a. Surface B is partially occluded by Surface A. However, the depth image generated by translating A along the camera space z-axis wrongly occludes additional regions of B. In Figure 5a, an example image exhibits occlusion artifacts in areas close to objects silhouettes. We avoid these problems by translating the depth image along the viewing rays instead, as illustrated in Figure 4b. As a consequence, the same region in surface B is occluded by surface A and the depth image, and no occlusion artifacts appear in the example image (Figure 5b). Obviously, visibility splatting may still discard visible surfaces if z_t is too small, or lead to the blending of several surfaces if z_t is too big. A good choice for z_t is the average distance between the surfels.

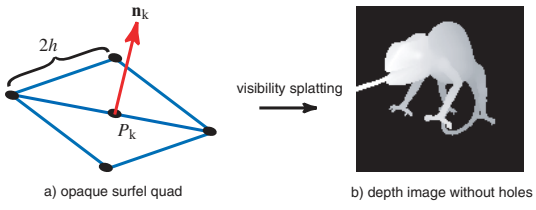

Figure 3: Visibility splatting: a) Opaque quad centered at surfel P_k. b) Depth image without holes in the Z-buffer.

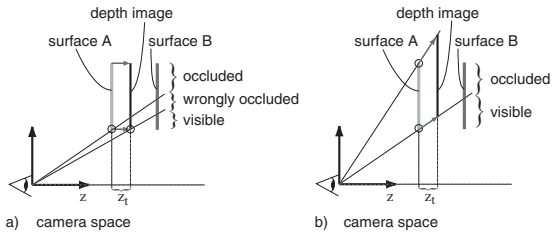

Figure 4: Applying the depth offset z_t: a) Translation along the camera space z-axis. b) Translation along viewing rays.

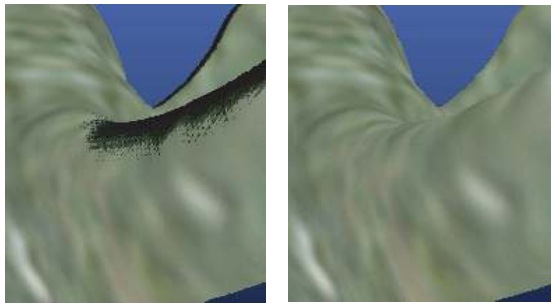

Figure 5: Comparison of the two depth offset schemes: a) Translation along the camera space z-axis leads to occlusion artifacts. b) Translation along the viewing rays shows no artifacts.

5.2. EWA Splatting Using Textured Polygons

In the second rendering pass, we perform surface splatting by rendering textured polygons representing a discrete object space EWA resampling filter (7). To accelerate the computation of the resampling filter, Heckbert [8] proposes to use a look-up table storing a limited number of precomputed, discrete resampling filters, which he calls an *angled image pyramid*. This pyramid is generated and indexed by quantizing the five parameters of an ellipse: x and y position, minor and major radius, and orientation. However, this approach is not suitable for implementation on graphics hardware, since it requires a huge amount of texture memory. Furthermore, there is no hardware support for quantizing the ellipse parameters and indexing the image pyramid to access the corresponding resampling filter, hence this would have to be done in software.

Our approach to solve this problem is shown in Figure 6. We use a single alpha texture that encodes a discrete unit Gaussian basis function. For each surfel, we then need to compute a corresponding parallelogram, which stretches and scales the unit Gaussian to match the object space EWA resampling filter. The stretching and scaling is performed implicitly by texture mapping, and we only have to compute the vertices of the parallelogram. As explained below, these computations can be performed completely in the programmable vertex shaders of current graphics processors. We then render the textured parallelograms into the frame buffer using additive alpha blending. By enabling depth comparison with the depth image generated in the first rendering pass while disabling depth update, this has the effect of blending together all splats within the depth range z_t.

In the following, we will discuss how to determine the object space EWA resampling filter, how to compute the vertex positions of the surfel parallelogram in object space, and how to choose the optimal size of the alpha texture.

Determining the object space EWA resampling filter: To compute the object space EWA resampling filter $\rho'_k(\mathbf{u})$ of Equation (7), we need to evaluate the Jacobian J_k^{-1}. First, we derive an analytical form of J_k, which maps the coordinates $\mathbf{u}$ of the local surface parameterization to viewport coordinates $\mathbf{x}$, and then we compute its inverse. Compared with the technique introduced for screen space EWA splatting [16], our approach, shown in Figure 7, avoids ray casting and can be computed with fewer instructions.

We construct a local parameterization of the object surface around a point P_k by approximating the surface with its tangent plane given by the normal $\mathbf{n}_k$. The parameterization is defined by choosing two orthogonal basis vectors $\widetilde{\mathbf{s}}$ and $\widetilde{\mathbf{t}}$ in this plane, attached to the position $\widetilde{\mathbf{o}}$ of the point P_k. Note that $\widetilde{\mathbf{s}}$, $\widetilde{\mathbf{t}}$, and $\widetilde{\mathbf{o}}$ are 3×1 vectors defined in object surface. Hence a point $\mathbf{u}$ with components u_s and u_t in local surface coordinates corresponds to a point $\mathbf{p}^o(\mathbf{u}) = \widetilde{\mathbf{o}} + u_s\cdot\widetilde{\mathbf{s}} + u_t\cdot\widetilde{\mathbf{t}}$ in object space. If we assume that the transformation from

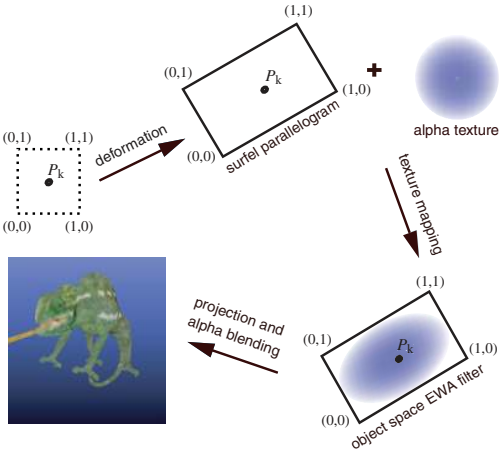

Figure 6: EWA splatting using textured polygons. Note the dashed line means the shape of the surfel polygon can not be determined before rendering, but is view dependent.

object space to camera space only contains uniform scaling $\mathbf{S}$, rotation $\mathbf{R}$ and translation $\mathbf{T}$, a point $\mathbf{u}$ corresponds to the following point $\mathbf{p}^c(\mathbf{u})$ in camera space:

$$\begin{aligned}\mathbf{p}^c(\mathbf{u}) &= \mathbf{R}\cdot\mathbf{S}\cdot\mathbf{p}^o(\mathbf{u})+\mathbf{T}\\ &= (\mathbf{R}\cdot\mathbf{S}\cdot\widetilde{\mathbf{o}}+\mathbf{T})+u_s\cdot\mathbf{R}\cdot\mathbf{S}\cdot\widetilde{\mathbf{s}}+u_t\cdot\mathbf{R}\cdot\mathbf{S}\cdot\widetilde{\mathbf{t}}\\ &= \mathbf{o}+u_s\cdot\mathbf{s}+u_t\cdot\mathbf{t}, \qquad (9)\end{aligned}$$

where $\mathbf{o}=(o_x,o_y,o_z)$ is the surfel position in camera space, while $\mathbf{s}=(s_x,s_y,s_z)$ and $\mathbf{t}=(t_x,t_y,t_z)$ are the basis vectors defining the local surface parameterization in camera space. Next, we map the points from camera space to screen space. This includes the projection to the image plane by perspective division, followed by a scaling with a factor η to screen coordinates (viewport transformation). The scaling factor η is determined by the view frustum and computed as follows:

$$\eta = \frac{v_h}{2\tan\left(\frac{fov}{2}\right)},$$

where v_h stands for the viewport height and fov is the field of view of the viewing frustum. Hence, screen space coordinates $\mathbf{x}=(x_0,x_1)$ of the projected point (u_s,u_t) are computed as (c_0 and c_1 are given translation constants)

$$\begin{aligned}x_0 &= \eta\cdot\frac{o_x+u_s\cdot s_x+u_t\cdot t_x}{o_z+u_s\cdot s_z+u_t\cdot t_z}+c_0\\ x_1 &= -\eta\cdot\frac{o_y+u_s\cdot s_y+u_t\cdot t_y}{o_z+u_s\cdot s_z+u_t\cdot t_z}+c_1. \qquad (10)\end{aligned}$$

So the Jacobian J_k consisting of the partial derivatives

of (10) evaluated at $(u_s, u_t) = (0,0)$ is

$$J_k = \begin{bmatrix} \frac{\partial x_0}{\partial u_s} & \frac{\partial x_0}{\partial u_t} \\ \frac{\partial x_1}{\partial u_s} & \frac{\partial x_1}{\partial u_t} \end{bmatrix} (0,0)$$
$$= \eta \cdot \frac{1}{o_z^2} \begin{bmatrix} s_x \cdot o_z - s_z \cdot o_x & t_x \cdot o_z - t_z \cdot o_x \\ s_z \cdot o_y - s_y \cdot o_z & t_z \cdot o_y - t_y \cdot o_z \end{bmatrix}.$$

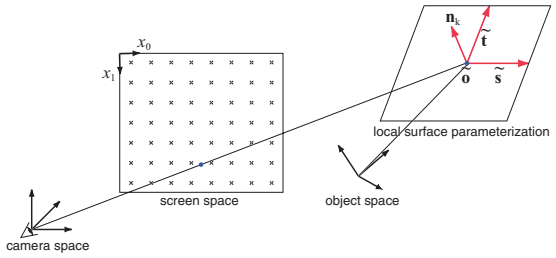

Figure 7: Calculating the Jacobian J_k.

Computing the surfel parallelogram vertex position: Once the Jacobian matrix is computed, the object space EWA resampling filter defined on the locally parameterized surface can be written as:

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

For points with approximately unit spacing, the reconstruction kernel is chosen as a unit Gaussian, i.e., $V_k^r = I$, as illustrated in Figure 8a. Otherwise, it can be determined as proposed in [16]. We decompose the symmetric matrix M_k as follows:

$$M_k = \mathbf{Rot}(\theta) \cdot \Lambda \cdot \Lambda \cdot \mathbf{Rot}(\theta)^T, \tag{11}$$

where

$$\mathbf{Rot}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \quad \text{and} \quad \Lambda = \begin{bmatrix} r_0 & 0 \\ 0 & r_1 \end{bmatrix}.$$

The rotation matrix $\mathbf{Rot}(\theta)$ consists of the eigenvectors, and the scaling matrix Λ consists of the square roots of the eigenvalues of M_k. Introducing the linear relationship

$$\mathbf{u} = \mathbf{Rot}(\theta) \cdot \Lambda \cdot \mathbf{y}, \tag{12}$$

we have $\mathbf{y}^T\mathbf{y} = \mathbf{u}^T M_k^{-1} \mathbf{u}$ and we can rewrite $G_{M_k}(\mathbf{u})$ as

$$G_{M_k}(\mathbf{Rot}(\theta) \cdot \Lambda \cdot \mathbf{y}) = \frac{1}{2\pi \cdot r_0 r_1} e^{-\frac{1}{2}\mathbf{y}^T\mathbf{y}} = \frac{1}{r_0 r_1} G_I(\mathbf{y}). \tag{13}$$

Equation (13) represents a unit Gaussian in $\mathbf{y}$, which is mapped to the elliptical Gaussian resampling filter using (12) as shown in Figure 8b. Since our alpha texture encodes the unit Gaussian, we rotate and scale a unit quad using (12), and then apply the alpha texture to the deformed quad, yielding the elliptical resampling filter as a textured parallelogram. Although the Gaussian resampling filter has infinite support in theory, in practice it is computed only for a limited range of the exponent $\beta(\mathbf{y}) = \frac{1}{2}\mathbf{y}^T\mathbf{y}$. Hence, we

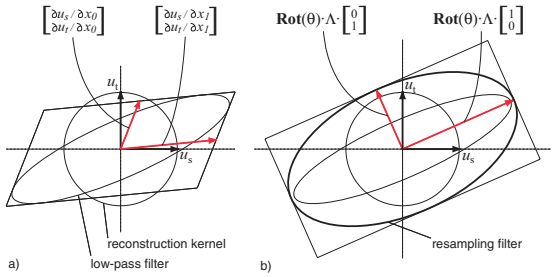

Figure 8: The object space EWA resampling filter: a) A unit reconstruction kernel and the warped low-pass filter defined by J_k. b) The resampling filter as a mapping from a unit circle to an ellipse.

choose a cutoff radius c such that $\beta(\mathbf{y}) \leq c$, where a typical choice is $c = 1$. Thus the alpha texture actually encodes the unit Gaussian in the domain

$$\mathbf{y} \in \begin{bmatrix} -\sqrt{2c} \\ -\sqrt{2c} \end{bmatrix} \times \begin{bmatrix} \sqrt{2c} \\ \sqrt{2c} \end{bmatrix}.$$

To encode the vertex positions of the deformed surfel quad and to perform texture mapping, each vertex has texture coordinates $\mathbf{v} \in \{(0,0),(0,1),(1,1),(1,0)\}$. Given vertex texture coordinates $\mathbf{v} = (v_0, v_1)$, we compute the camera space position $\mathbf{p}^c(\mathbf{u})$ as illustrated in Figure 9: First, we need to map the texture coordinates $\mathbf{v}$ to the coordinates $\mathbf{y}$ in the domain of the unit Gaussian by scaling them according to the cutoff radius c:

$$\mathbf{y} = 2\sqrt{2c}\left(\begin{bmatrix} v_0 \\ v_1 \end{bmatrix} - \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}\right). \tag{14}$$

Then we deform the textured quad using Equation (12), yielding coordinates $\mathbf{u}$ of the local surface parameterization. With Equation (9), we finally compute the vertex positions $\mathbf{p}^c(\mathbf{u})$ of the surfel parallelogram in camera space.

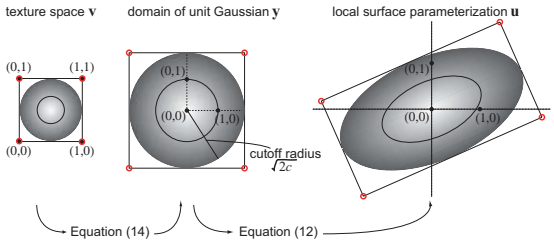

Figure 9: Constructing a texture-mapped polygon that represents the object space EWA resampling filter.

Determining the optimal texture size: To make full use of the eight bit precision of the alpha texture, we encode

the non-constant part from Equation (13) in each texel, i.e., $g(\mathbf{y}) = e^{-\frac{1}{2}\mathbf{y}^T\mathbf{y}}$. Hence, the function range $[0\ldots1]$ maps to the whole range $[0\ldots255]$ of all texel values. Although a larger texture size increases the precision of the discrete representation of the 2D Gaussian function, the quantization of the function values to eight bits leads to redundancy in high resolution alpha textures, since nearby texels may always map to the same quantized value. Assume we use a square texture with resolution $len \times len$. Since the unit Gaussian function is rotation invariant, we can represent $g(\mathbf{y})$ as $g'(r) = e^{-\frac{1}{2}r^2}$ in polar coordinates. To make full use of eight bits per texel, the following condition should be satisfied for all $r \in [0, \sqrt{2c}]$:

$$\left|\frac{d(g'(r))}{dr}\frac{\sqrt{2c}}{\frac{len}{2}}\right| = \left|r \cdot e^{-\frac{1}{2}r^2}\frac{\sqrt{2c}}{\frac{len}{2}}\right| \leq \frac{1}{2^8-1}. \qquad (15)$$

From this it follows that

$$len \geq 510 \cdot \sqrt{2c} \cdot r \cdot e^{-\frac{1}{2}r^2}. \qquad (16)$$

The optimal texture resolution corresponds to the smallest value of len that satisfies the above condition. For the typical choice $c = 1$ we find $len = 438$.

6. Implementation

6.1. Programmable Shader Computations

Programmable shaders [12] provide efficient GPU level computing. Our hardware accelerated surface splatting algorithm is implemented using programmable vertex and pixel shaders.

Vertex Shader Computations: During the first rendering pass, the depth offset along the view rays in camera space is computed using the vertex shader. In the second rendering pass, the computation for the vertex positions of the surfel polygon is also implemented using the vertex shader. Due to the simple instruction set of the vertex shader [12], the implementation of the symmetric matrix decomposition in Equation (11) requires a careful design. We make heavy use of the two most powerful and efficient shader instructions, reciprocal square root (RSQ) and reciprocal (RCP). The details of the computation are described in Appendix A. The constant part of Equation (13), $\frac{1}{2\pi \cdot r_0 \cdot r_1}$, is output to the alpha channel of the diffuse color register. In this way, it can be accessed by the pixel shader later. The scaling factor η can be precomputed and stored as a vertex shader constant. Since vertex shaders do not support the creation of new vertices, we need to perform the same per-vertex computation four times for each surfel quad.

Pixel Shader Computations: The computation of $w_k\rho_k(\mathbf{x})$ in Equation (2) is performed using the pixel shader's per-fragment processing. The colors w_k^r, w_k^g, w_k^b are retrieved from the red, green, and blue channel of the input register for the diffuse color. Multiplying the texel alpha value by the constant $\frac{1}{2\pi \cdot r_0 \cdot r_1}$, which is stored in the alpha channel of the diffuse color, yields $\rho_k(\mathbf{x})$. Finally, the accumulation of the EWA splats in Equation (2) is performed by additive alpha blending.

6.2. Hierarchical Rendering

In our implementation we have chosen the surfel LDC tree [13] for hierarchical rendering. However, other hierarchical data structures such as a bounding sphere hierarchy [14] could be used, too. While traversing the LDC tree from the lowest to the highest resolution blocks, we perform view-frustum culling of surfel blocks and backface culling using visibility cones. To choose the appropriate octree level to be projected, we perform block bounding box warping similar to [13]. This allows us to estimate the number of projected surfels per pixel, facilitating progressive rendering.

For efficient hardware acceleration, the surfels of multiple LDC tree blocks need to be stored together in a number of big vertex buffers. This minimizes the switching of vertex buffers during rendering and enhances performance. The vertex buffers are allocated in the local memory of the graphics card or in AGP (Accelerated Graphics Port) memory. Their sizes are chosen to be optimal for the graphics card and to maximize performance. To access the vertex data of a block in the LDC tree, we store the corresponding vertex buffer ID and the start position of its vertex data in the vertex buffer.

7. Pre-Processing

Due to the irregular sampling of point models and the truncation of the Gaussian kernel, the basis functions r_k in object space do not form a partition of unity in general. Neither do the resampling kernels in screen space. To enforce a partition of unity, we could perform *per-pixel normalization* in the frame buffer after splatting [16]. However, this post-processing operation is not supported by today's graphics hardware. In addition, directly locking and accessing the frame buffer during rendering for per-pixel normalization slows down the rendering speed. But without normalization, the brightness of the final image varies with the accumulated filter weights, leading to visible artifacts (Figure 10a). To solve this problem we propose a pre-processing method, which we call *per-surfel normalization*.

7.1. Per-Surfel Normalization

If the basis functions r_k in Equation (1) sum up to one everywhere, applying a low-pass filter will still guarantee that the resampling filters in screen space form a partition of unity. Consequently, our pre-processing method does not consider the prefiltering step during rendering and becomes a view

independent process. The *normalized* view independent texture function in object space can be written as follows:

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

Unfortunately, the above rational basis function $\hat{r}_k$ invalidates the derivation of a closed form resampling filter. Instead, we use the sum of the weights at each surfel position to approximate the above formula, yielding

$$f_c(\mathbf{u}) = \sum_{k\in\mathbf{N}} w_k s_k r_k(\mathbf{u}-\mathbf{u}_k),$$

where $s_k = \frac{1}{\sum_{j\in\mathbf{N}} r_j(\mathbf{u_k}-\mathbf{u}_j)}$. We call s_k the surfel's *normalization weight*, which is acquired by a view independent process discussed in Section 7.2. Based on Equation (7), we adjust our object space EWA resampling filter with s_k, yielding:

$$\rho'_k(\mathbf{u}) = s_k G_{V_k^r + J_k^{-1} J_k^{-1\,T}}(\mathbf{u}-\mathbf{u}_k), \qquad (17)$$

which is the resampling filter used by object space EWA surface splatting with *per-surfel normalization*.

7.2. Acquiring surfel normalization weights

To acquire a surfel's normalization weight, the point model is first rendered using our two pass algorithm without prefiltering and per-surfel normalization. Then the Z-buffer and the frame buffer are read back to acquire the normalization weights. In the third pass, we traverse the surfel LDC tree and calculate the depth value and the projected position in screen space of the center point of each surfel polygon. Based on the Z-buffer information, the visible surfels are detected. After rendering, the alpha channel of each frame buffer pixel stores the sum of the accumulated contributions S from all EWA splats projected to that pixel. Hence the visible surfel's normalization weight is $s_k = \frac{1}{S}$. To capture the normalization weights for surfels invisible from one view, multiple-view weight capture is applied, which can be performed automatically or interactively. For automatic capture, a bounding sphere is built around the model first. Then surfel weights are captured from different view points, which are uniformly distributed on the surface of the sphere. For interactive capture, the user manually specifies a part of the point model for capturing as shown in Figure 10b. In both methods, the normalization weight of the same surfel may be captured several times. To get rid of noise, we choose the median value as the final normalization weight. Per-surfel normalization assumes that the normalization weight is the same in the small neighborhood covered by the surfel polygon. For each surfel, the normalization weight captured at the center of the surfel quad is copied to its polygon vertices during rendering. The above assumption is not true, however, at the edges of the point model. In this case we capture the normalization weight for each vertex of the surfel polygon. Thus surfel quads at edges have different normalization weights for each vertex.

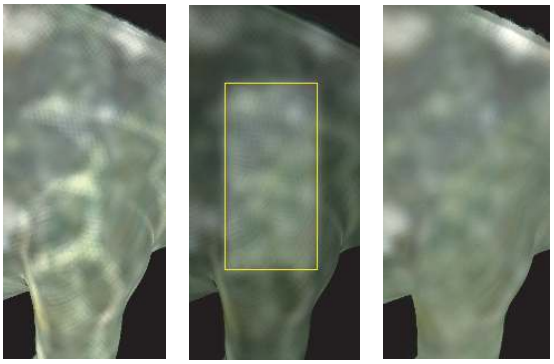

Figure 10: Pre-processing: a) Direct rendering without preprocessing. b) The surfel weights inside the yellow box are captured interactively. c) Rendering with per-surfel normalization.

In the capturing process, direct rendering of surfel models can cause overflow in the alpha channel of frame buffer pixels where the accumulation of contributions from different splats is greater than one. In this case, the surfel normalization weight is incorrectly computed due to clamping in the alpha channel. To solve the problem, we use a global parameter γ to avoid overflow. In our implementation, the weight capture process uses the following object space texture function:

$$f_c(\mathbf{u}) = \sum_{k\in N} \gamma w_k r_k(\mathbf{u}-\mathbf{u_k}).$$

By setting γ to a suitable value less than one, the accumulated contributions of the splats in a pixel will not be too large to be clamped. Consequently, the image rendered during normalization weight capture is darker, as shown in Figure 10b. A typical choice for γ is 0.73, which works for most surfel models we use. For a normalization weight s'_k and a global parameter γ, the final surfel normalization weight is $s_k = s'_k\gamma$.

8. Results

We have implemented our hardware accelerated point-rendering pipeline with DirectX 8.1 under Windows XP. Furthermore, we have implemented an automatic surfel normalization weight capture tool for pre-processing of point models. Performance has been measured on a 1GHz AMD Athlon system with 512 MB memory. We used an ATI Radeon 8500 graphics card and a NVidia GeForce4 Ti 4400 GPU. To test antialiasing and rendering speed, we used a plane consisting of 64k surfels with a checkerboard texture (Figure 11). In order to test raw performance, i.e., how many splats per second our algorithm can render, we disabled block culling and multiresolution rendering. We compare three different object space surface splatting algorithms.

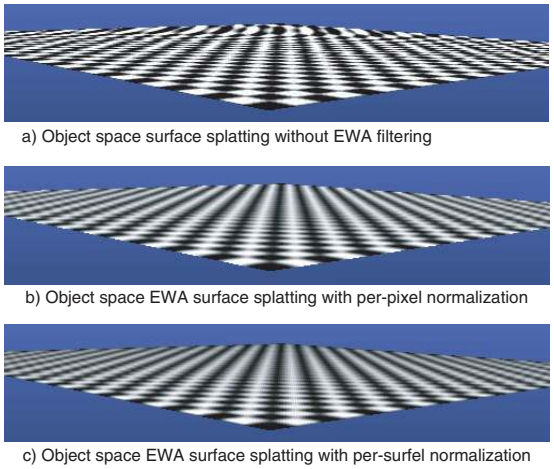

Figure 11: Checkerboard rendering using three different object space surface splatting algorithms.

Figure 11a shows the texture quality of object space surface splatting without EWA filtering. In this case, the basis function encoded in the alpha texture only consists of the reconstruction filter in object space, leading to severe aliasing artifacts. Figure 11b shows the texture quality of object space surface splatting without pre-processing, where we perform *per-pixel* normalization in the frame buffer after rendering each frame to solve the normalization problem. Figure 11c shows the texture quality of object space EWA surface splatting with pre-processing and *per-surfel* normalization enabled during rendering. The texture quality is comparable to per-pixel normalization. The visual difference between per-surfel and per-pixel normalization becomes apparent only in highly irregularly sampled surfaces.

Algorithm	**# VS 1**	**# VS 2**	**Fps**
Without EWA	13	21	57.3/59.6
EWA (per-pixel norm.)	13	78	1.3/1.0
EWA (per-surfel norm.)	13	78	16.9/23.8

Table 1: Rendering performances of the three algorithms for frame buffer resolution 512 $\times$ 512. The left performance number was measured with the ATI Radeon 8500 graphics processor, the right number corresponds to the NVidia GeForce4 Ti4400 GPU.

Table 1 shows rendering performances of the three algorithms for a frame buffer resolution of 512 $\times$ 512. For each algorithm, #VS1 and #VS2 denote the number of vertex shader instructions used in the first and second rendering pass, respectively. From Figure 11 and Table 1, we can see that EWA object space surface splatting with per-surfel normalization is much faster than per-pixel normalization, while achieving comparable image quality. The splatting algorithm without EWA filtering is the fastest method, at the expense of poor image quality due to serious aliasing.

Data	**# Points**	**256 $\times$ 256**	**512 $\times$ 512**
Salamander	103389	18.9/30.0 fps	16.9/24.9 fps
Chameleon	101685	17.0/23.2 fps	14.0/19.0 fps
Wasp	273325	6.1/8.0 fps	5.1/6.1 fps
Fiesta	352467	5.5/7.0 fps	3.8/5.3 fps

Table 2: Rendering performance for frame buffer resolutions 256 $\times$ 256 and 512 $\times$ 512. The left performance number was measured with the ATI Radeon 8500 graphics processor, the right number corresponds to the NVidia GeForce4 Ti4400 GPU.

Object space EWA surface splatting with per-surfel normalization proves to be the desirable approach for high quality and interactive point rendering (Figure 12). As can be seen in Table 2, it can handle 1.6 to 3 million points per second when object level culling is enabled. For surfel models with similar size, models that allow more culling will be rendered faster. To improve the rendering quality further, we can combine per-surfel normalization and per-pixel normalization during progressive rendering: the point model is rendered by per-surfel normalization during user interaction, and refined by per-pixel normalization afterwards. We also compare the performance of object space EWA surface splatting (per-surfel normalization) with a software implementation of screen space EWA surface splatting. For a 512 output resolution, our algorithm can render approximately 1.5 million antialiased splats per second. On the same PC, the software-only implementation of screen space EWA surface splatting only renders up to 200,000 splats per second. The

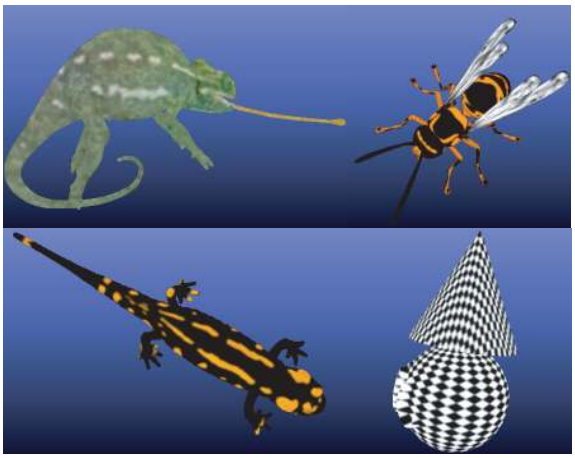

Figure 12: Various test models (chameleon, wasp, salamander, fiesta) with high frequency textures are rendered using object space EWA surface splatting.

software renderer is also more sensitive to the output image resolution. When the image resolution becomes higher, its performance drops linearly [16]. In contrast, hardware accelerated object space EWA surface splatting is less sensitive to the output resolution (see Table 2).

Table 1 also shows that the performance of the vertex shader plays an important role in our implementation. The difference in the number of vertex shader instructions used by object space surface splatting with EWA filtering (per-surfel normalization) and without filtering results in a significant difference in performance. Moreover, when the number of surfels becomes larger, the data transfer of vertex data from AGP memory across the AGP bus to the graphics board becomes a bottleneck. In our implementation, we can reduce the number of vertex shader instructions by transmitting redundant per-vertex information, but this requires more memory at the same time. Hence, when rendering large point models, there is a tradeoff between vertex shader simplicity and data transfer bandwidth, respectively.

9. Conclusions

This paper contains two main contributions. First, we presented a new object space formulation of EWA surface splatting for irregular point samples. Second, we developed a new multi-pass approach to efficiently implement this algorithm using vertex and pixel shaders of modern PC graphics hardware. We have also shown a pre-processing method for proper normalization of the EWA splats.

Besides increased performance, we believe there are other advantages of using GPUs for point-based rendering. While CPUs double in speed every two years, GPUs increased their performance by a factor of 11 in the last nine months. Undoubtedly, GPU performance will continue to increase faster than CPU speed in the near future. Due to their fixed-function processing there is more room for parallelization. For example, the GPUs of the XBox and the GeForce 4 have two vertex shaders in hardware. Because each surfel is processed independently, this will linearly increase the performance of our algorithm. Furthermore, the performance of a software implementation of EWA surface splatting drops with increased output resolution, an effect that is not nearly as serious for our hardware based implementation. Finally, using the GPU leaves the CPU free for other tasks, such as AI or sound processing in games.

As future work we will further optimize our implementation and adapt it to upcoming new hardware features, such as improved vertex shader instructions and framebuffer normalization. We plan to apply the lessons learned to propose improvements for existing GPUs and to design special-purpose hardware for EWA surface splatting. We plan to extend our approach to support interactive rendering for semi-transparent point models and deal with issues like view-dependent shading. We also envision to use our system to render animated objects. While we feel that this requires few changes to the core rendering pipeline, it is a challenging task to develop data structures that efficiently represent dynamic point clouds. Besides, we also want to extend our method to volume data and facilitate interactive texture based EWA volume splatting.

Acknowledgements

We would like to thank Jessica Hodgins and Paul Heckbert from Carnegie Mellon University for helpful discussions, Evan Hart and Jeff Royle from ATI and Henry Moreton from NVIDIA for providing us with the latest graphics cards, and Jennifer Roderick Pfister, Wei Li and Wei Chen for proofreading the paper.

References

1. M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Point Set Surfaces. In *Proceedings of IEEE Visualization*, pages 21–28. San Diego, CA, October 2001. 2

2. L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics*, volume 18 of *SIGGRAPH 84 Proceedings*, pages 103–108. July 1984. 4

3. B. Chen and M. X. Nguyen. POP: A Hybrid Point and Polygon Rendering System for Large Data. In *Proceedings of IEEE Visualization*, pages 45–52. San Diego, CA, October 2001. 2

4. J. Cohen, D. Aliaga, and W. Zhang. Hybrid Simplification: Combining Multi-Resolution Polygon and Point Rendering. In *Proceedings of IEEE Visualization*, pages 37–44. San Diego, CA, October 2001. 2

5. 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. 1

6. J. P. Grossman. *Point Sample Rendering*. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, August 1998. 2

7. J. P. Grossman and W. Dally. Point Sample Rendering. In *Rendering Techniques '98*, pages 181–192. Springer, Wien, Vienna, Austria, July 1998. 1, 2

8. 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. 1, 2, 5

9. A. Kalaiah and A. Varshney. Differential Point Rendering. In *Proceedings of the 12th Eurographics Workshop on Rendering*, pages 139–150. London, UK, June 2001. 2

10. 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. 1

11. 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. 1, 2

12. E. Lindholm, M. Kilgard, and H. Moreton. A User-Programmable Vertex Engine. In *Computer Graphics*, SIGGRAPH 2001 Proceedings, pages 149–158. Los Angeles, CA, July 2001. 2, 7

13. 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. 1, 2, 4, 7

14. 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. 1, 2, 4, 7

15. M. Stamminger and G. Drettakis. Interactive Sampling and Rendering for Complex and Procedural Geometry. In *Proceedings of the 12th Eurographics Workshop on Rendering*, pages 151–162. London, UK, June 2001. 1, 2

16. M. Zwicker, H. Pfister., J. Van Baar, and M. Gross. Surface Splatting. In *Computer Graphics*, SIGGRAPH 2001 Proceedings, pages 371–378. Los Angeles, CA, July 2001. 1, 2, 3, 5, 6, 7, 10

Appendix A: Symmetric Matrix Decomposition for Vertex Shader Implementation

We choose the following symmetric matrix decomposition method for our vertex shader implementation. M_k is rewritten as follows:

$$M_k = \mathbf{Rot}(\theta)\cdot\Lambda\cdot\Lambda\cdot\mathbf{Rot}(\theta)^T = \begin{bmatrix} A & \frac{B}{2} \\ \frac{B}{2} & C \end{bmatrix}.$$

Then we define

$$Sgn(x) = \begin{cases} -1, & x < 0 \\ +1, & x \geq 0 \end{cases}.$$

The following variables are stored in the vertex shader temporary registers:

$$\begin{aligned} p &= A - C \\ q &= A + C \\ t &= Sgn(p)sqrt(p^2 + B^2). \end{aligned}$$

With those temporary variables, the scaling matrix can be computed as

$$\Lambda = \begin{bmatrix} r_0 & 0 \\ 0 & r_1 \end{bmatrix} = \begin{bmatrix} \sqrt{\frac{(q+t)}{2}} & 0 \\ 0 & \sqrt{\frac{(q-t)}{2}} \end{bmatrix}.$$

$Rot(\theta)$ can be computed, too. If $t = 0$, $Rot(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, else if $t \neq 0$:

$$Rot(\theta) = \begin{bmatrix} \sqrt{\frac{t+p}{2t}} & -Sgn(Bp)\sqrt{\frac{t-p}{2t}} \\ Sgn(Bp)\sqrt{\frac{t-p}{2t}} & \sqrt{\frac{t+p}{2t}} \end{bmatrix}.$$

Square root and division operations in the above formulas can be computed efficiently using the standard DirectX vertex shader instructions "RSQ" and "RCP", respectively.