High-Quality Point-Based Rendering on Modern GPUs

Mario Botsch, Leif Kobbelt
Computer Graphics Group
RWTH Aachen, Germany

Abstract

In the last years point-based rendering has been shown to offer the potential to outperform traditional triangle based rendering both in speed and visual quality when it comes to processing highly complex models. Existing surface splatting techniques achieve superior visual quality by proper filtering but they are still limited in rendering speed. On the other hand the increasing availability and programmability of graphics hardware lead to the develoepement of very efficient hardware-accelerated rendering methods. However, since no filtered splats are used, these approaches trade visual quality for rendering speed.

In this paper we propose a rendering framework for point-based geometry providing high visual quality as well as efficient rendering. Our approach is based on a two-pass splatting technique with Gaussian filtering, resulting in a visual quality comparable to existing software rendering systems. Using programmable graphics hardware we delegate all expensive rendering tasks to the GPU, thereby minimizing data transfer and saving CPU resources. The proposed system renders up to 28M mid-quality or up to 10M high-quality surface splats per second on the latest graphics hardware.

1 Introduction

Due to their simplicity and efficiency triangles meshes are the de facto standard geometry representation in computer graphics. As the hardware components for the complete mesh processing pipeline, i.e. mesh generation (3D scanners), mesh processing (CPU) and finally mesh rendering (graphics hardware), gets more and more powerful, the typical surface or scene complexity is steadily increasing. Meshes containing several millions of triangles are nowadays commonly used.

In contrast the resolution of displays is not increasing at the same speed. Therefore rendering highly complex models results in triangles whose projected area is less than a few pixels. Using standard scanline-conversion methods for the rendering of these tiny triangles becomes inefficient because of the necessary overhead for the triangle setup.

Hence, above a certain complexity, points are the conceptually more efficient rendering primitive. Holes in the rendered image (e.g. when zooming in) can be avoided by image-based filters, by adjusting the sampling density, or by so-called surface splatting. In the latter case each point is associated with a radius and a normal vector and therefore represents a small disc in 3-space, that is projected onto the image plane.

Another advantage of point-based rendering (PBR) besides the higher efficiency is that it can also provide superior rendering quality compared to standard polygon-based rendering. For PBR the lighting computations are performed on a per point basis, corresponding to high quality Phong shading in the surface case. For anti-aliased rendering sophisticated splatting techniques assign a Gaussian filter kernel to the splats, resulting in an elliptically weighted average (EWA) filtering of the image — similar to anisotropic texture filtering [11].

An additional benefit of point-based geometry representations is their conceptual simplicity. Since no connectivity information exists only a set of points has to be stored and processed. Hierarchical encoding schemes for point-based geometry provide compact storage and efficient progressive transmission of these datasets. Recently, several mesh processing algorithms have been reformulated for point-based surface representations, like e.g. spectral processing [17], geometry simplification [18], surface editing [25] and multiresolution shape modeling [19].

The focus in this paper is on the final stage of the point-based geometry processing pipeline, i.e. the rendering of point-sampled geometry. In this topic, existing approaches offer only a trade-off between rendering speed and visual quality. On this scale one extreme are the sophisticated purely software-based implementations of filtered splatting that provide the highest rendering quality. The major drawback of these approaches is that they put high load on the main CPU, but still do not achieve higher rates than 4M splats per second on current hardware [2].

On the other extreme people are trying to free the CPU for other tasks by making use of graphics hardware for point-based rendering, motivated by the steadily increasing performance and programmability of modern graphic processing units (GPUs). But since hardware-acceleration is mainly targetting polygon-based rendering there is no obvious way how to (mis-)use graphics hardware for high-quality filtered surface splatting. Hence, rendering quality had to be sacrificed for rendering speed, leading to a rendering performance of above 50M points per second — if the points are rendered as small unfiltered squares [7].

In this paper we propose a rendering framework for point-based geometry providing high visual quality as well as efficient rendering. Our approach is based on a two-pass filtered splatting technique, resulting in the a visual quality comparable to existing software rendering systems. Using programmable graphics hardware we delegate all expensive rendering tasks to the GPU, thereby minimizing data transfer and saving CPU resources. The proposed system renders up to 28M mid-quality or up to 10M high-quality filtered surface splats per second on the latest graphics hardware.

2 Related Work

Using points as rendering primitives was first proposed in the pioneering work of Levoy and Whitted [14], followed by Grossman and Dally [10], presenting algorithms for the generation as well as and for the rendering of point sets. This work has been improved in the Surfels paper by Pfister et al. [20]. They sample objects using 3 orthogonal LDIs and use image-space filters to achieve a hole-free rendering.

Alexa et al. [1] use local Least Squares approximations to adjust the point sampling for displaying. Their point set surfaces have been extended to a progressive representation in [9].

Zwicker et al. [26] introduce surface splatting by image-based EWA filtering, resulting in high quality anti-aliased rendering, comparable to anisotropic texture filtering [11]. Similar to the footprints of Westover [24] disc-shaped splats in object-space project to elliptical splats with Gaussian intensity distribution in image-space. While this software-based approach is only able to process 250k splats per second it provides the highest visual quality. Botsch et al. [2] present an adaptive octree encoding scheme for point-based geometry that provides very compact storage and a hierarchical rendering algorithm. Their method is able to process up to 14M points or 4M high quality filtered splats per second by using a quantization of splat shapes.

All the above software-based rendering methods have proven that point-based rendering can be superior to polygon-based rendering for highly complex scenes. While they can provide very high visual quality, their rendering speed is limited to about 4M splats per second on current hardware. Even if this point rate may be sufficient for today's models these software implementations completely block the CPU from other tasks besides rendering.

Therefore several authors propose to use graphics hardware for point-based rendering. Sophisticated rendering techniques used in software implementations, like e.g. A-buffers [3], are not available on today's graphics hardware. Hence, the proposed approaches either lose some visual quality or try to compensate for the missing functionality by multiple rendering passes.

The first to use hardware acceleration for PBR were Rusikiewicz and Levoy [22]. In order to be able to render the large datasets of the Digital Michelangelo project [13] they combine a hierarchy of bounding spheres with a splatting technique. In order to blend overlapping fuzzy splats in some ϵ-depth-slap they propose a two-pass rendering approach.

Stamminger and Drettakis [23] dynamically adjust the point samping rate for rendering of complex procedural geometry. This approach is extended to a mixed point and polygon rendering approach for complex plant ecosystems in Deussen et al. [8]. Further approaches mixing the rendering of points and polygons have been proposed by Chen et al. [4] and Cohen et al. [6]. Although we are targetting pure point-based rendering, our methods could be integrated into their algorithms since we are using standard OpenGL rendering only.

Ren et al. [21] reformulate the image based EWA filtering of [26] to object-space filtering in order to map the surface splatting approach to graphics hardware, also using a two-pass rendering method. They render each splat as a textured rectangle in object-space. This concept causes the number of processed points to be multiplied by four, slowing down the rendering to about 2M–3M splats per second.

Coconu and Hege [5] propose to use an octree-based spatial data structure containing points and triangles to do the visibility calculations. By sorting and rendering the octree cells from back to front in each frame they avoid using the z-buffer at all. Although this avoids an expensive second rendering pass, it leads to the problem that front and back sides of objects are blended without depth control.

In a very recent paper Dachsbacher et al. [7] present a hierarchical LOD structure for points that is adaptively rendered by sequentially processing it by the GPU. They report impressive point rates above 50M points per second, but the points are rendered as unfiltered view-plane aligned small squares.

3 GPU-Based Splatting

In our approach we also propose the use of splats as rendering primitives, as they have major advantages compared to pure one-pixel points. Since splats are not just a piece-

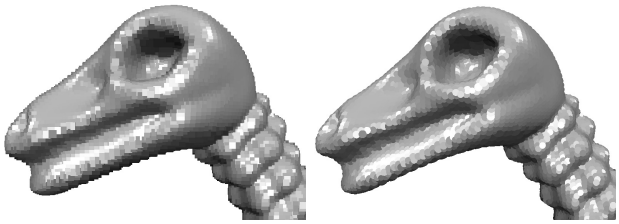

Figure 1. The typical thickening and aliasing effects of square splats (left) is effectively avoided by using the correct ellipitical splats shapes (right).

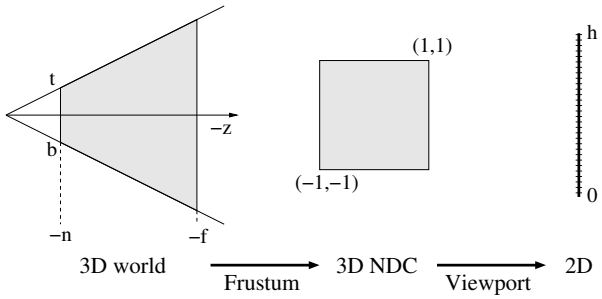

Figure 2. Transformation pipeline. The mapping from eye-space to image-space consists of a projective warp of the viewing frustum to the unit cube, a parallel projection and a 2D-transform to match the window extent.

wise constant but a piecewise linear geometry representation, they exhibit the same quadratic approximation order as triangle meshes. Therefore a decent approximation can be achieved with a relatively low number of splats, offering a good compromise between polygonal meshes on the one hand and one-pixel points on the other hand.

The concept of splat filtering by blending overlapping splats also provides a much higher rendering quality compared to unfiltered point rendering that often leads to high-frequency noise in the image, especially in the case of textured models.

Since for splats the point sampling rate does not have to be adjusted each frame the static geometry data can be stored in the video or AGP memory where it can be directly accessed in DMA mode, thereby minimizing data transfer costs during rendering.

The rendering of point splats involves several sub-tasks: first the size and shape of the splats have to be determined from the current viewing parameters so that we get a hole-free image (Sec. 3.1 and 3.2). Using these techniques alone already results in mid-quality elliptical but still unfiltered surface splats. Nevertheless it provides a much better representation of the geometry than fixed splat shapes, especially noticable near contours (cf. Fig. 1).

Further improvement in visual quality can be achieved by blending overlapping splats using splat filtering (Sec. 3.3), resulting in high quality anti-aliased rendering. During rendering, splat contributions are accumulated by additive blending, so that each pixels contains a weighted sum of color values $\sum_i w_i(rgb)_i$. Therefore a final normalization step dividing each pixel's RGB color by the corresponding sum of weights is required to get the correct filtering $(\sum_i w_i(rgb)_i) \,/\, (\sum_i w_i)$. This final normalization step will be described in Sec. 3.4.

For all these rendering tasks our goal is the consequent delegation to the GPU. Therefore the algorithms have to be formulated in a way they can directly be mapped to the programmable vertex and fragment shaders of the latest graphics hardware.

3.1 Splat Size

In order to obtain a watertight rendering, the projected size of a splat has to be determined from the viewing parameters and the splat's position and radius.

The OpenGL projection pipeline from eye/camera coordinates to the final 2D image is depicted in Fig. 2: The viewing volume is given by the distances to the near (n) and far (f) clipping planes and the parameters top (t) and bottom (b) controlling the opening angle. After this frustum has been projectively warped to the unit cube $[-1,1]^3$, a simple parallel projection is done by omitting the z coordinate, resulting in 2D coordinates in $[-1,1]^2$. Finally these 2D coordinates are scaled up by the viewport mapping to match the (integer) window coordintes $\{0,\ldots,w\} \times \{0,\ldots,h\}$. This can also be considered as projecting the viewing volume onto the near plane $z = -n$ and scaling the resulting coordinates by $\frac{h}{t-b}$.

The image-space size of a splat is the size of its projection onto the image plane. The exact result is quite expensive to compute, since it depends on the splat position as well as on the splat normal. Instead we approximate the size by projecting the bounding sphere of the splat and neglecting its x and y offsets from the optical axis, as also proposed in [4].

The image-space splat size $size_{win}$ can then be computed by a projection onto the near plane and a scaling to transform from near-plane coordinates to image-plane coordinates:

$$size_{win} = r \cdot \frac{n}{z_{eye}} \cdot \frac{h}{t-b}, \tag{1}$$

where z_{eye} is the splat's distance from the camera, r is its radius, and n, t, b, h are the respective projection parameters depicted in Fig. 2.

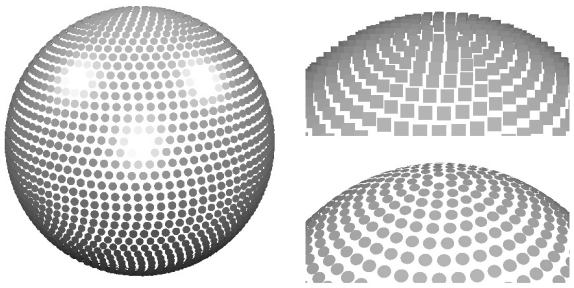

Figure 3. Splat size and shape. Adjusting the splat size just results in screen-space squares to be rendered at the splat center's position (top). Additionally using the correct elliptical splat shape gives a much better approximation, especially near contours (splat radii have been decreased to better show the effect).

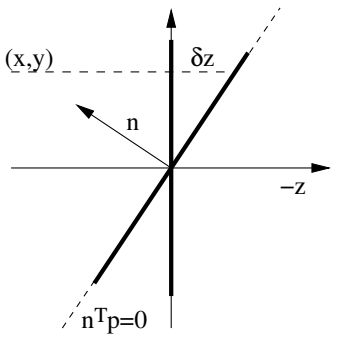

Figure 4. Depth correction. Adjusting the image-space splat size yields a square parallel to the image plane. The required depth correction δz can be computed from the image-plane square coordinates $(x, y) \in [-1, 1]^2$ and the splat's eye-space normal vector n.

Adjusting the image-space size causes a $size_{win} \times size_{win}$ image-space square to be rendered centered at the splat center's projected position (cf. Fig. 3, top right).

3.2 Splat Shape

Since a splat represents a small disc in object-space, its projection onto the image plane is an ellipse. The radii and orientation of this ellipse depend on the splat's normal vector transformed to eye-coordinates. Adjusting the splat's shape based on its eye-space normal vector results in the desired behaviour (cf. Fig. 3).

Initially the adjustment of splat size causes small image-plane aligned squares to be rendered. For each of its pixels we have to determine whether it is the projection of a point inside or outside the splat, i.e. whether the point corresponding to the pixel has a distance to the 3D splat center that is less than the splat radius or not.

Similar to [5] we use the `NV_point_sprite` extension to get a parameterization of the image-space square over $[-1, 1]^2$. For a pixel having coordinates $(x, y) \in [-1, 1]^2$ we compute its depth offset δz from the splat center as a linear function determined by the eye-space normal vector $(n_x, n_y, n_z)^T$, cf. Fig. 4:

$$\delta z = -\frac{n_x}{n_z}x - \frac{n_y}{n_z}y \qquad (2)$$

The depth offset δz can then be used to compute the 3D-distance from the splat center: the pixel corresponds to a point inside the splat iff the length $\left\|(x, y, \delta z)^T\right\|_2$ is less than one. Note that the depth offset δz is just an approximation, since we assume a parallel projection in Eq. 2, neglecting the angle between the viewing ray and the splat's normal. Since this may cause ellipses to become too flat (resulting in holes), we bound the maximum foreshortening of the ellipses, as also proposed in [22, 5].

Using the correct splat size and splat shape results in a hole-free rendering with a much better visual quality of contours, since the typical thickening effect of square splats is effectively avoided by elliptical splats (cf. Fig. 1 and Fig. 3). Nevertheless for high quality anti-aliasing we should use the splat filtering described in the next section.

3.3 Splat Filtering

For splat filtering each splat in object-space is associated a radially decreasing Gaussian weight function. The projection of this Gaussian results in an image-space elliptical Gaussian, whose values are used to blend the respective pixels. Therefore the image-space weight of a pixel is a function of the 3D distance of its corresponding object-space point to the splat's center, i.e. the norm $\left\|(x, y, \delta z)^T\right\|_2$ that has already been computed to determine the splat shape. Hence, the final weight can be looked up in a 1D Gaussian texture:

$$\alpha(x, y) = \textit{GaussTexture1D}\left[\left\|(x, y, \delta z)^T\right\|\right] \qquad (3)$$

For the concept of splat filtering overlapping splats should be blended if and only if their z-distance is sufficiently small, otherwise the splat in front should overwrite the splat behind. While software-based algorithms can easily implement this behaviour using, e.g., modified A-buffers [3], there is no way to map this ϵ-depth-test to current graphics hardware. Although some splats can be culled based on their backfacing orientation, this is not sufficient in the general case.

Figure 5. Splat filtering including per-pixel normalization results in high quality image reconstruction. This effect becomes especially visible in the case of high frequency textures. While in the left unfiltered image alias-problems appear, the splat filtering on the right provides a smooth result.

Therefore we use a two-pass rendering approach, like proposed in [22, 21]: In a first pass the scene is rendered just to the z-buffer, with all z-values having an ϵ-offset added to them. If for the second pass the z-buffer update is turned off, i.e. the z-values from the first pass are used read-only, this results in the desired blending of splats whose depth distance is less than ϵ.

For this ϵ-depth test to work reliably we have to make sure that each pixel's depth value is correct. Since the splats we render are up to now just image-plane aligned elliptically trimmed rectangles, all of their pixels' depth values equal the depth value of the splat's center vertex (cf. Fig. 6). Especially when viewing a splat from a flat angle this will lead to large errors in the depth component, resulting in blending artifacts near contours.

To address this issue we use a per-pixel depth correction, i.e. for each pixel we compute its correct depth value in window coordinates based on the splat's eye-space normal vector. The corresponding object-space depth offset δz w.r.t. to the splat center has already been computed to determine the splat shape, see Eq. 2.

The required window z-coordinate z_{win} has to be computed from the adjusted eye-space z-coordinate $z_{eye}+\delta z$ by applying the frustum and viewport mapping to it (cf. Fig. 2). This transformation can be written as

$$z_{win} = \frac{a\left(z_{eye} + \delta z\right) + b}{z_{eye} + \delta z}, \tag{4}$$

where $a = f/(f-n)$ and $b = -2fn/(f-n)$ are derived from the composition of projection and viewport mapping. The frustum parameters f and n are again the distances to the near and far plane, respectively.

In the second rendering pass we accumulate all splats passing the ϵ-depth test by weighted additive blending, such that each pixel stores $(\sum_i \alpha_i (rgb)_i, \sum_i \alpha_i)$. Hence, in a final normalization step the RGB part each pixel has to be divided by its α component.

Figure 6. In order to show the effect of depth correction the z-buffer of the left scene has been re-projected using the techniques of [12]. While splats rendered without depth correction (center) have constant depth, like being parallel to the image plane, per-pixel depth correction solves this problem and avoids blending artifacts (right).

3.4 Per-Pixel Normalization

In [5] all splats are blended in back-to-front order without taking the depth buffer into account. The resulting artifacts are accepted for the advantage of a one-pass rendering algorithm. In contrast the issue of correct EWA splatting including the required normalization has been addressed in [21]. Instead of doing a per-pixel normalization they switch to a lower quality per-surfel normalization, where the sum of α-weights is approximated to be constant for each splat.

On todays graphics hardware, however, a per-pixel normalization can be performed very efficiently. Reading the buffer, doing the normalization on the CPU and writing the resulting buffer back may be a first idea, but is in fact prohibitively expensive because of the required data transfer.

Instead we propose to accumulate the weighted splats in an offscreen buffer. This buffer can then be used as a texture for one single rectangle of the window's size. Rendering

this rectangle will cause each pixel to go through the fragment pipeline again, so that a pixel shader can do the necessary division by α.

Since this technique effectively avoids sending the pixel data over the AGP bus, the per-pixel normalization can be performed at the cost of rendering one textured rectangle using a small pixel shader program. Compared to the time the two rendering passes take, this is basically negligible.

The resulting images provide a high visual quality comparable to existing software- or hardware-based implementations of EWA filtered surface splatting (c.f. Fig. 5 and Fig. 8). But, as we will describe in the next sections, our approach is siginicantly faster than existing methods.

4 Implementation

Our implementation is based on OpenGL, the results we present are measured on a 2.8GHz Pentium4 with a GeForceFX 5800 Ultra graphics card, running Linux. We delegate the different rendering tasks described in the previous sections to the programmable vertex shaders [15] and pixels shaders, respectively. Although we used NVIDIA specific OpenGL extensions, the same would have been possible using the recently released vendor independent `ARB_vertex_program` and `ARB_fragment_program` extensions or even using the more comfortable and platform independent language Cg [16].

Data layout: When processing complex point datasets including additional data like, e.g. normals and colors, the data transfer can become the limiting factor. Therefore the static scene geometry is stored in video or AGP memory so that it can be accessed by the GPU in DMA mode. Although we use the `NV_vertex_array_range` extensions for this, it should also be possible using the new `ARB_vertex_buffer_object` extension. In order to prevent cache misses the data should be arranged in interleaved vertex arrays.

Vertex stage: A vertex program is used to transform the vertex position and to compute the eye-space normal vector. Using Eq. 1 it computes the image-space splat size, where rounding the resulting size to an integer value turned out to increase efficiency. Finally the vertex program sets up the data required for computing the per-pixel depth offset and depth correction. This data is passed to the fragment stage in the texture coordinate registers, thereby keeping 32 bit of accuracy.

Fragment stage: The fragments generated from the image-space squares are processed by a fragment shader. This program first combines lighting and texture/color information, where the lighting can be precomputed in a cube map for higher efficiency. In order to compute the depth offset a parameterization of the splat square is required. Using the `NV_point_sprite` extension these parameter values can be derived from the point sprite texture coordinates. They are used to compute the depth offset according to Eq. 2. Either the (squared) norm $\left\|(x, y, \delta z)^T\right\|^2$ (no filtering) or the correct Gaussian weight (filtering) are stored in the α component, so that the α test can discard pixels outside the elliptical image-space splat. For splat filtering and blending additionally the per-pixel depth correction has to be done (Eq. 4).

Blending: In order to accumulate the weighted contributions of splats in the form $(\sum_i \alpha_i (rgb)_i, \sum_i \alpha_i)$ different blending modes have to be used for the RGB and α components, which can be achieved using the `EXT_blend_func_separate` extension.

Normalization: For the final per-pixel normalization the two rendering passes are rendered to an offscreen buffer that is then used as texture image for a window-sized rectangle. In order to allow for non-power-of-two viewport width and height the `NV_texture_rectangle` extension is used.

Performance The bottleneck of our rendering algorithm is the fragment processing, mainly because of the per-pixel depth correction. In order to avoid generating unnecessary fragments we added a backface-culling method to the vertex shader that drops a splat depending on its eye-space normal vector. This simple technique effectively removes about 40% of the splats and speeds up the rendering significantly. However, since fragment programs are a very recent feature they are supposed to have room for improvements, so that future drivers or cards will yield better performance.

5 Discussion

Our point-rendering method can be used on two different quality levels: for the faster but the lower quality solution we only adjust splat size and splat shape as described in Sec. 3.1 and 3.2. As shown in Fig. 1, this already gives much better results than using fixed splat shapes and effectively avoids the typical thickening effect near contours. On this quality level our method is able to render about 28M elliptical un-filtered splats per second on a 2.8GHz Pentium4 with a GeForceFX 5800 Ultra graphics card.

For high quality filtered surface splatting we have to use two rendering passes, because on current graphics hardware there is no other way to implement the ϵ-depth test described in Sec. 3.3. As proposed by Dachsbacher et al.

		512×512		1024×1024	
	# points	unfiltered	filtered	unfiltered	filtered
Charlemagne	1.63M	17.2 (28.1)	6.3 (10.4)	16.9 (27.6)	6.2 (10.1)
St. Matthew	1.57M	17.2 (27.1)	6.4 (10.1)	16.9 (26.6)	6.3 (9.9)
David Head	1.08M	25.6 (27.9)	9.4 (10.3)	24.1 (26.2)	9.0 (9.8)
David	1.06M	26.4 (28.0)	9.5 (10.0)	25.7 (27.3)	9.3 (9.9)
Max	655k	42.7 (27.9)	15.3 (10.0)	40.4 (26.4)	14.2 (9.3)
Male	148k	124.2 (18.4)	56.5 (8.4)	112.4 (16.7)	46.6 (7.0)
Balljoint	137k	172.6 (23.7)	73.2 (10.0)	124.3 (17.0)	45.3 (6.2)
Chameleon	101k	178.7 (18.2)	74.9 (7.6)	150.3 (15.3)	55.3 (5.6)

Table 1. The resulting timings for several models, given for window resolutions of 512×512 and 1024×1024, and unfiltered or filtered rendering. The first values are frames per second, the values in brackets are million splats per second.

[7] adding this additional feature to future GPUs would increase the rendering performance for high quality filtered primitives by a factor of two. However, our current implementation is based on a two-pass rendering and the necessary per-pixel normalization. It achieves a splat rate of about 10M high quality filtered splats per second.

We tested the performance of our implementation using models of strongly varying complexities from 100k points up to 1.6M points. Table 1 lists the respective rendering times in frames per seconds and in million splats per second. Since our rendering approach is limited by the fragment processing speed, models containing more points results in higher splat rates as the projected size of splats will be smaller. Comparing the timings for a window resolution of 512×512 or 1024×1024, respecively, confirms this result. Since for the more complex models projected splat sizes are still just a few pixels, their rendering speed stays almost the same. Low complexity models generate larger image-space splats, putting more load on the fragment processing.

Comparing our approach to the one of [21], we follow more closely the idea of point-based rendering, since we represent and render each splat using just one vertex. Since our resulting splat rendering is mainly pixel-based, several computations can be formulated easier and solved more efficiently. E.g. in order to compute the ϵ depth offset for the first rendering pass, Ren et al. have to shift each vertex along the viewing rays in object-space. Since we update the fragment's depth value anyway we just have to change one constant parameter of the pixel shader. Another limitation of [21] is the symmetric matrix decomposition that has to be done for each vertex in order to compute the corresponding object-space rectangle's corner positions. Again, we determine the splat shape and Gaussian α-mask by simple image-space computations as described in Sec. 3.2. While the approach of [21] renders up to 3M surface splats per second using the heuristic per-surfel normalization, our implementation achieves significantly higher splat rates using the per-pixel normalization.

It is harder to compare our approach to the work of Coconu and Hege [5] since both methods are targeting different goals. By using only one rendering pass and blending all splats regardless of their depth distance, [5] trade visual quality for rendering speed. In constrast we aim at a depth-correct implementation of filtered surface splatting, requiring the expensive two-pass approach. Nevertheless we manage to achieve higher frame rates. The major bottleneck of [5] seems to be the CPU intensive sorting of the octree cells from back to front, whereas our method puts no load at all on the main CPU.

While our approach is significantly slower than [7], we achieve improved visual quality even when using less surface splats, since the piecewise linear splats have better approximation properties. Nevertheless our rendering method should seamlessly integrate into the sequential point trees. This promising combination would result in a GPU-based hierarchical level-of-detail rendering of high-quality splats.

6 Conclusion

In this paper we presented an algorithm for the rendering of point-based geometry that closes the gap between high quality software implementations and lower quality hardware-accelerated approaches.

We propose the consequent delegation of the involved rendering tasks to the GPU, since this keeps the CPU free for other tasks. As more and more point-based geometry processing methods are used, it is even more important that

these algorithms do not have to share CPU resources with the rendering process. Since the efficiency as well as the programmability of current GPUs is increasing at a much higher rate than CPU performance, using the GPU for most efficient rendering seems to be the straightforward consequence.

Future work includes an implementation based on vendor independent ARB extensions only or even using the high-level language Cg. This would provide high quality splat rendering for on a wider range of graphic cards. A very promising direction of future research is the integration of this work into the sequential point trees of Dachsbacher et al. [7], since this would result in a high quality hierarchical level-of-detail rendering algorithm that is completely processed by the GPU.

Acknowledgements

The St. Matthew and David models have been taken from the Stanford 3D Scanning Repository. The models Male, Balljoint and Chameleon are from the Pointshop3D homepage.

References

[1] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, and C. Silva. Point set surfaces. In *Proc. IEEE Visualization 2001*, pages 21–28, 2001.

[2] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proc. Eurographics Workshop on Rendering 2002*, 2002.

[3] L. Carpenter. The a-buffer, an antialiased hidden surface method. In *Siggraph 1984 Conference Proceedings*, pages 103–108, 1984.

[4] B. Chen and M. X. Nguyen. Pop: a hybrid point and polygon rendering system for large data. In *Proc. IEEE Visulization 2001*, pages 45–52, 2001.

[5] L. Coconu and H.-C. Hege. Hardware-accelerated point-based rendering of complex scenes. In *Proc. Eurographics Workshop on Rendering 2002*, pages 41–51, 2002.

[6] J. D. Cohen, D. G. Aliaga, and W. Zhang. Hybrid simplification: combining multi-resolution polygon and point rendering. In *Proc. IEEE Visualization 2001*, pages 37–44, 2001.

[7] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. In *Siggraph 2003 Conference Proceedings*, 2003.

[8] O. Deussen, C. Colditz, M. Stamminger, and G. Drettakis. Interactive visualization of complex plant ecosystems. In *Proc. IEEE Visualization 2002*, 2002.

[9] S. Fleishman, D. Cohen-Or, M. Alexa, and C. T. Silva. Progressive point set surfaces. *to appear in ACM Transactions on Graphics*.

[10] J. P. Grossman and W. J. Dally. Point sample rendering. In *Eurographics Workshop on Rendering 1998*, pages 181–192, 1998.

[11] P. S. Heckbert. Fundamentals of Texture Mapping and Image Warping. Master's thesis, University of California at Berkley, 1989.

[12] L. Kobbelt and M. Botsch. An interactive approach to point cloud triangulation. In *Eurographics 2000 Conference Proceedings*, 2000.

[13] 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 *Siggraph 00 Conference Proceedings*, pages 131–144, 2000.

[14] M. Levoy and T. Whitted. The use of points as display primitives. Technical report, CS Departement, University of North Carolina at Chapel Hill, January 1985.

[15] E. Lindholm, M. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Siggraph 2001 Conference Proceedings*, pages 149–158, 2001.

[16] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Siggraph 2003 Conference Proceedings*, 2003.

[17] M. Pauly and M. Gross. Spectral Processing of Point-Sampled Geometry. In *Siggraph 2001 Conference Proceedings*, 2001.

[18] M. Pauly, M. Gross, and L. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proc. IEEE Visualization 2002*, 2002.

[19] M. Pauly, R. Keiser, L. Kobbelt, and M. Gross. Shape Modeling with Point-Sampled Geometry. In *Siggraph 2003 Conference Proceedings*, 2003.

[20] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Siggraph 2000 Conference Proceedings*, pages 335–342, 2000.

[21] L. Ren, H. Pfister, and M. Zwicker. Object space ewa surface splatting: A hardware accelerated approach to high quality point rendering. In *Eurographics 2002 Conference Proceedings*, pages 461–470, 2002.

[22] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In *Siggraph 2000 Conference Proceedings*, pages 343–352, 2000.

[23] M. Stamminger and G. Drettakis. Interactive sampling and rendering for complex and procedural geometry. In *Eurographics Workshop on Rendering 2001*, pages 151–162, 2001.

[24] L. Westover. Footprint evaluation for volume rendering. In *Siggraph 1990 Conference Proceedings*, pages 367–376, 1990.

[25] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. PointShop 3D: An Interactive System for Point-Based Surface Editing. In *Siggraph 2002 Conference Proceedings*, 2002.

[26] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Siggraph 2001 Conference Proceedings*, pages 371–378, 2001.

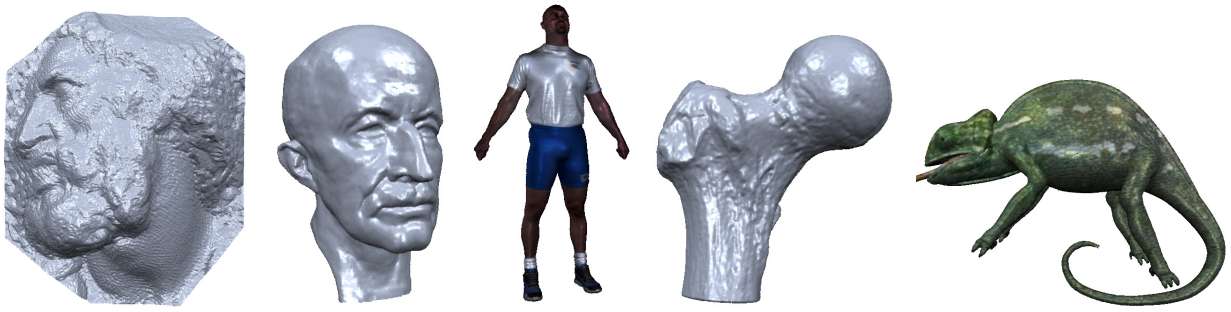

Figure 7. Some models we tested our implementation on and whose rendering timings and complexities are shown in Table 1. From left to right: St. Matthew, Max Planck, Male, Balljoint, Chameleon.

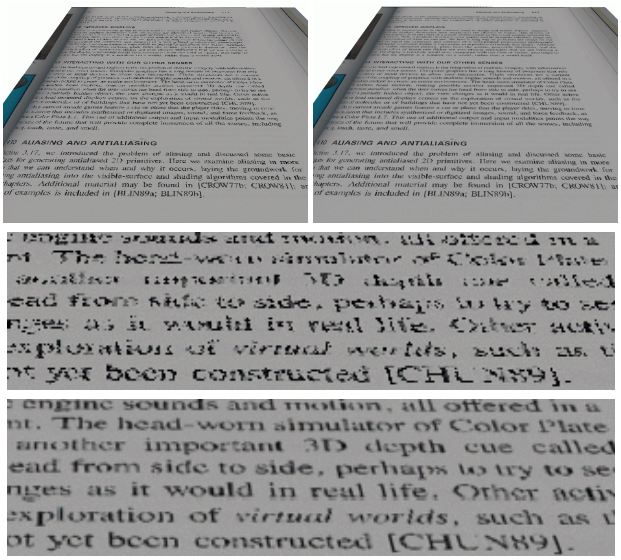

Figure 8. Splat filtering results in high quality image reconstruction similar to anisotropic texture filtering. In the upper left image and the upper close-up no filtering has beed used, leading to strong alias-artifacts. Splat filtering instead effectively removes these artifacts (upper right and lower close-up).

Figure 9. The models of Michelangelo's David and Charlemagne are both 3D range scanned statues. For the David a consistent decimated triangle mesh has been sampled. The input data for the Charlemagne model are just the set of registered but unconnected range scans.