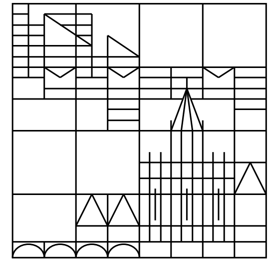
Universität Konstanz



Single-Pass High-Quality Splatting

Tilo Ochotta
Stefan Hiller
Dietmar Saupe

Konstanzer Schriften in Mathematik und Informatik
Nr. 219, September 2006

ISSN 1430–3558

© Fachbereich Mathematik und Statistik
© Fachbereich Informatik und Informationswissenschaft
Universität Konstanz
Fach D 188, 78457 Konstanz, Germany
Email: preprints@informatik.uni–konstanz.de
WWW: http://www.informatik.uni–konstanz.de/Schriften/

Single-Pass High-Quality Splatting*

Tilo Ochotta† Stefan Hiller Dietmar Saupe

Department of Computer and Information Science
University of Konstanz, Germany

Abstract

In this paper, we introduce a novel real-time splat rendering pipeline that relies on components of state-of-the-art splat and mesh rendering methods to enable high-quality shader effects such as reflection and refraction for splats. Our pipeline consists of one single rendering pass and aggressively utilizes the most practical components of both techniques. The entire pipeline is compactly implemented in the vertex and fragment stages on current GPUs. Our renderer simplifies current splatting techniques, accelerates rendering, and, moreover, achieves excellent visual quality.

1 Introduction

The rapid growth of 3D-content in entertainment applications demands the data display at interactive rates with best possible visual quality. In addition to standard illumination, enhanced shader effects simulate physical surface properties, such as reflection, refractions, and shadows, and allow for improved visual quality. In recent years, points have been proposed as an alternative to traditional mesh-based rendering. Unlike meshes, point sets do not rely on connectivity information for rendering. Dropping connectivity saves memory and has the advantage that the input data can directly be rendered without preprocessing.

However, the loss of connectivity implies two drawbacks. Firstly, when implemented on current graphics hardware, splatting pipelines rely on time-consuming multi-pass rendering, since attributes are blended by overlapping splat contributions. Secondly, the loss of connectivity complicates continuous interpolation of attributes, such as normals vectors and texture coordinates. To alleviate this problem, the number of primitives can be increased to ensure a smooth reconstruction of attributes. However, this leads to increasing memory requirements and causes an increase of the GPU/CPU workload, which in turn decreases the rendering speed due to the limited bandwidth.

We introduce a novel splat rendering technique that is inspired by both, splat and mesh rendering methods. We combine them into a simple and fast rendering pipeline that facilitates enhanced shading effects in high visual quality that have not been possible up to now using current splatting approaches.

In order to exploit advantages of mesh rendering, we modify the existing splat rendering pipeline. The core idea is the following. In contrast to previous splatting approaches in which reconstruction of smooth attributes is carried out by blending in screen space, we interpolate attributes at an earlier stage in object space.

We reorganize the input data in patches. Each patch is parameterized as an elevation map over a planar domain and resampled on a regular grid structure. In previous work [16], it was shown that the elevation map data structure is well suitable for the application of compression of point-sampled surfaces. We consider the scenario in which the models are available in compressed form, e.g., in a digital library. The rendering of these data requires two consequtive operations; firstly, decoding the data; and secondly, reconstructing the models, yielding unorganized sets of points. We propose to skip the second step and render the elevation maps directly in their compact form. The regular sampling on the patch enables us to exploit point neighbor information during rendering by projecting pixel shader fragments back onto the patches in order to interpolate attributes from nearby points.

This interpolation scheme has the following major advantages:

- Utilizing the patch information, we replace disadvantageous splat blending by efficient attribute interpolation. The complete pipeline

* August 2, 2006, Copyright ©by the authors

† ochotta@inf.uni-konstanz.de

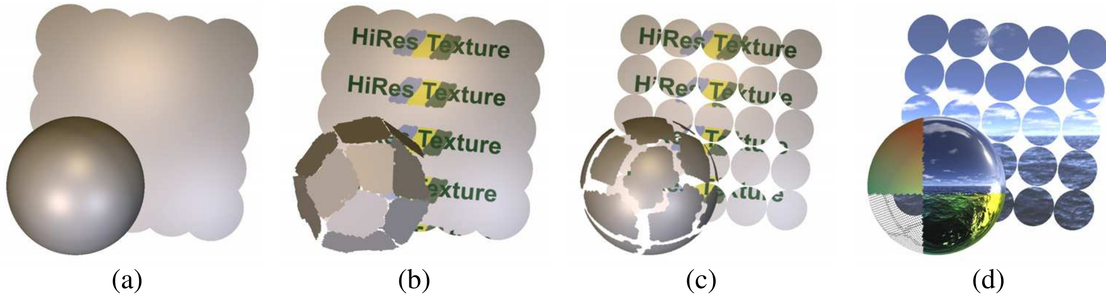


Figure 1: A scene consisting of a sphere (9858 splats) and a plane of 25 splats with varying normals, rendered with our system; (a) per-pixel lighting using our splat interpolation method; (b) visualized patches of the sphere with zero elevation values, and the plane with texture of a resolution much higher than the sampling density of the plane; (c) patches with elevation values with indicated patch boundaries, and rendering the splats on the plane with reduced radii; (d) combined shading effects on the sphere, such as lighting (upper left), reflectance mapping with an envrironment map (upper right), combined lighting and reflectance (lower right), and visualized splats with reduced radii and unique color (lower left); the plane shows reflectance mapping derived from an environment map.

consists of one single rendering pass. This simplifies the overall rendering complexity characteristic of current GPU-based splatting techniques, which, without exception, require at least three rendering passes.

- Our attribute interpolation considerably improves the quality of reconstructed attributes, such as normal vectors, texture coordinates, etc. This not only results in an improved visual quality of the rendered image, but it also allows applying enhanced shader effects to splats, e.g., reflection and refraction without having to rely on densely sampled models.
- The resampled model is stored completely on graphics hardware, while the introduced connectivity information is only given implicitly, it does not require additional memory. The neighborhood information can directly be reconstructed in the fragment shader when fragments are processed in the GPU pipeline.

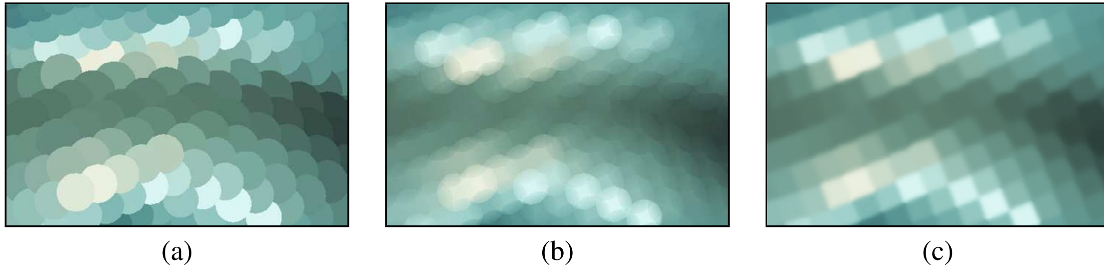
The rest of the paper is organized as follows: Section 2 briefly reviews previous research on splat rendering methods. Section 3 provides a detailed description of our rendering pipeline. In Section 4, we present experimental results, and in Section 5, we conclude and introduce our future work.

2 Related Work

The point rendering primitive was invented by Levoy and Whitted [13] and improved by Grossman and Dally [8]. Pfister et al. [18] and Zwicker et al. [24] propose to approximate a given surface by a set of oriented discs (splats). Several approaches were proposed that use the point primitive for fast rendering [20], [6]. However, these methods rely on reorganizing the data in complex data structures which are hard to implement on current GPUs. In ellipitcal weighted surface splatting [24], [6], [5], [3], high quality renderings of point sampled geometry and attributes are achieved by blending prefiltered point and attribute data.

An attractive property of splat-based point rendering is that splats may cover several pixels in screen space during rendering, just like polygons in traditional rendering. However, as a principle limitation, the sampling density of the point model which determines the splat sizes also governs the amount of high frequency detail in the model for geometry as well as for attributes, such as normal and texture maps. Therefore, prefiltering is necessary to avoid aliasing, which eliminates some of the high frequency detail that may be present in the original data.

In terms of hardware accelerated point rendering, splatting is simulated using the standard vertex and fragment shader pipeline on common GPUs [19], [11], [4], [5], [25], [9], [3]. The splats are rendered as point-sprites that are rotated according to their respective surface normal. This rendering pipeline is implemented in three rendering passes. In the first *visibility pass,* the splats are rendered slightly shifted in z-direction in order to define the depth range for blending of overlapping splats. In the second *accumulation pass,* weighted contributions are accumulated in order to smoothly blend the overlapping regions using a Gaussian weighting function. In the final *rendering pass* the accumulated contri-



(a) (b) (c)

Figure 2: Traditional Gauraud-like rendering with splats; oriented disks may be rendered directly without blending, which leads to worst visual quality (a); employing standard per-pixel blending with Gaussian weights leads to smoothing artifacts that are visible in the close-up range (b); decreasing the Gaussian weights leads to Voronoi-like structures and visible edges between the splat centers (c).

butions are normalized. Botsch et al. [3] considered a kind of Phong splatting by extending the blending to normals and texture attributes using multiple render targets for deferred shading. For a survey of hardware-accelerated point rendering techniques, see [21].

However, to date, high-quality splatting techniques all rely on multi-pass rendering, which possibly results in slower rendering speed compared to state-of-the-art hardware accelerated mesh rendering. Moreover, previous approaches tend to produce blurring artifacts, if blending of overlapping splats is performed (Fig. 2). While traditional mesh rendering pipelines continuously interpolate normal vectors, emulating this interpolation with splat blending requires for large overlap of splats and leads to a significant loss in rendering speed. Thus, complex shader effects that strongly rely on accurate normals, such as reflectance or refraction are hard to obtain in splat rendering pipelines.

In recent work, [14], [23], [9], [10], these problems were addressed by increasing the total number of primitives, thereby reducing the number of pixels covered by each splat. This approach improves the quality of reconstructed normals. However, increasing the number of points increases the total memory usage and consequently, leads to a significant loss in rendering speed due to the limited bandwidth on the GPU.

In our method we consider partitioning a point based model into patches. Pauly and Gross [17] used a patch layout to process the geometry of a given model in the frequency domain. In order to apply the standard discrete Fourier transform the patches were resampled yielding elevation maps, i.e., rectangular arrays of scalar heights above corresponding patch base planes. Ochotta and Saupe [16] used similar but irregularly shaped patches in order to apply a shape-adaptive wavelet transform followed by entropy coding for 3D geometry compression. In [15] patches of the same kind were used for hardware-accelerated rendering of point- and mesh-based 3D models.

3 Rendering Pipeline

In this section we describe how (1) to obtain and organize splat primitives and associated attributes such as colors, normals, and textures in the form of collections of regular height fields, (2) to efficiently represent and render the data in one single pass using current computer graphics technology, and (3) to ensure that attribute values are well interpolated at the fragment shader level.

The basic entity of a 3D model with attributes as used in this work is a patch given by a regularly sampled height field. This height field is defined on an irregularly bounded subset of a plane. This plane can be thought of as an average tangent plane of the 3D surface part that corresponds to the patch. The set of 3D points from all patches provides a point set approximation of the 3D model. Each of these points is obtained by offsetting a grid point of the corresponding plane within the support of the height field patch along the plane normal vector by a given signed distance. The corresponding data structure thus contains the plane parameters (a 3D point, a normal vector, and the grid resolution) together with a rectangular array of height values. Some of the grid points may be flagged using a special height value indicating that the height field patch does not include a point above the grid point thereby allowing for arbitrarily shaped supports of height fields.

The ”geometry” of the surface patch given by the

height field may be complemented by corresponding attribute fields of arbitrary resolution, such as color texture map, and normal map.

3.1 Patch Construction

In our framework we aim to achieve a small number of height fields. The regular height field structure is crucial for efficient rendering. Moreover, the patches must match up at their boundaries ensuring continuous error free rendering, requiring irregular boundaries.

Parameterization. We follow the patch construction in [16] that fulfills these criteria. In a nutshell, the method produces a partitioning of an input 3D point set using a split-merge strategy that employs the normal cone condition as in [17]. Each region of the partition consists of a set of point samples corresponding to a piece of the 3D model that can be parameterized as a height field over a planar domain.

Geometry sampling. This surface patch is resampled on a regular grid of the corresponding parameter plane using the moving least squares surface reconstruction [2, 1]. For further details see [16]. For this construction it cannot be guaranteed that no gaps between patches occur. In order to obtain a continuous rendering, we propose to let the patches slightly overlap. This can be achieved by including input points near a patch boundary into the corresponding region of the input point partition. In practice, one chooses $\epsilon > 0$ and includes an input point if its ϵ-neighborhood intersects the point set of the region.

Attribute sampling. The attribute fields texture and normal vectors must also be resampled from the original 3D model. Normal vectors can be computed at arbitrary height field locations as normal vectors of the associated MLS surface [1]. Alternatively, if normal vectors are given per input point, then normal vectors can be resampled on a regular grid on the support plane using bilinear interpolation in a Delaunay triangulation of the projected input points on the support plane. Texture attributes can be handled the same way.

3.2 Data Representation for Rendering

Our model consists of a set of elevation and attribute maps; the resolution of each map can be chosen independently. The model side information consists of one 4×4 transformation matrix per patch.

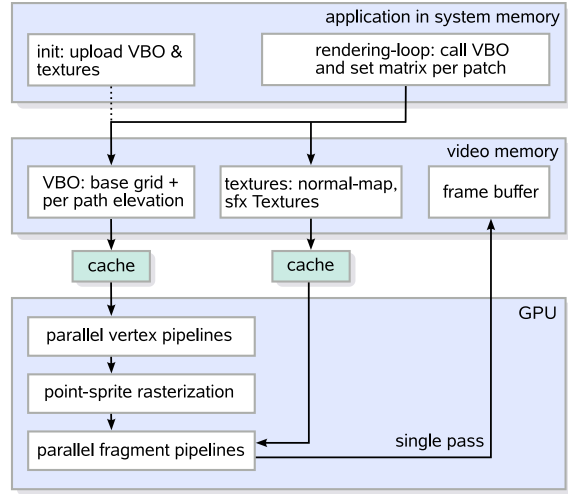


Figure 3: Overview of our rendering pipeline; the model data consists of elevation and attribute maps that are completely stored in graphics memory using VBOs and texture objects, respectively. The rendering pipeline operates in one single rendering pass.

For fast vertex processing, we use vertex buffer objects (VBOs) to store all elevation maps on the graphics card, see Fig. 3. More precisely, we set up a VBO that holds point positions and elevation data for each patch to be rendered. Since the patches are regularly sampled, the point positions on the patch plane can be expressed by one single index value, namely, the combined row and column, which is the address of the point in the sampling grid.

Using VBOs requires setting up an index buffer. In our framework, these are similar for all patches. The index buffer is a linear list $1, 2, \ldots, n$, where n is the number of points in the patch. Consequently, our index buffers are considerably simpler than those used for meshes, since we do not need to encode connectivity between vertices. Moreover, this type of index buffer does not stress the vertex caches on the GPU.

Additional attribute maps such as color or normal maps are stored as texture objects for each patch individually. Both VBOs and texture objects are uploaded into graphics memory once before rendering.

3.3 Splat Radii

One important task in splat-based rendering is the construction of the splat-radii for a given point set.

In our framework, the samples are regularly distributed on each base plane. This induces to set all radii to $\frac{1}{2}\sqrt{2}\mu$, where μ is the sampling step size of the respective grid. Nonetheless, the samples are elevated in normal direction of the plane, which may result in holes between the splats. We propose to adjust the radii depending on the angle between the splat normal and the normal of the plane. Specifically, we set

$$r = \frac{\frac{1}{2}\sqrt{2}\mu}{0.1 + 0.9\langle n, n_B \rangle}, \tag{1}$$

where r is the radius, and n and n_B denote the normal vector of the splat and the base plane, respectively. The motivation behind this definition is the expectation that a large angle between the splat normal and the plane normal indicates a large variance between neighboring elevation values, which results in increased spatial gaps. Consequently, the splat radii have to be increased in these areas. The two constants are included to establish scaling and maximal radii.

3.4 Rendering Loop

To render the complete model, our rendering loop iterates over all patches. In each step, the respective texture objects are addressed, and the corresponding VBO is triggered for rendering. In the following we describe the details of our rendering pipeline, which is implemented using common GPU vertex and fragment shader functionality, and the point-sprite extension.

Vertex stage. In the vertex shader, the point positions are translated according to their elevation values, and finally are rotated using the 4×4 matrix. Now the points are on the surface of the original model. The radius of each point is set according to Eq. (1), which defines the extension of the point-sprite for the following rasterization step.

Fragment stage. After point-sprite rasterization, the fragment shader transforms the point-sprite fragments for simulating splatting [4]. This is mandatory, since point-sprites are defined as screen aligned primitives. In the next subsection, we describe how the fragment shader continues to determine the per-fragment UV coordinates that are used to reconstruct normal and texture attributes, followed by the description of the final illumination procedure.

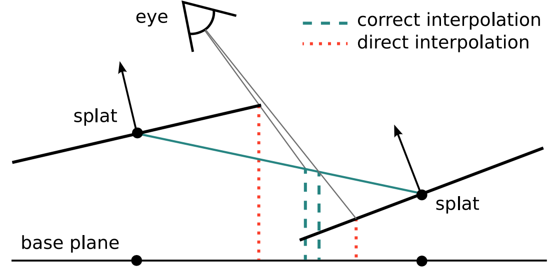


Figure 4: Illustration of our attribute interpolation scheme; the view rays are intersected with the splats for direct evaluation of UV coordinates (dotted lines); intersecting the view ray with the plane through neighboring splat centers (solid line which connects the splat centers) provides our corrected interpolation of UV coordinates (dashed lines).

3.5 Continuous Attribute Interpolation

We consider two slightly different variants of attribute interpolation.

Direct interpolation. Figure 4 illustrates interpolation at the fragment shader. In a naive direct version, we intersect the view rays with oriented splats in object space and read off attribute parameters on the base plane. The interpolation from the regular grid can be carried out using either simple bilinear interpolation or higher order filtering techniques. However, for neighboring fragments, this direct mapping approach may lead to large distances of UV coordinates on the patch plane (see red dashed lines), which may result in visual artifacts in the closeup view.

Correct interpolation. To obtain a continuous reconstruction of UV coordinates, we consider the plane that is given by elevated neighboring points. We intersect the view ray with this plane and project the intersection point onto the support plane in order to determine the corrected UV coordinates (green dashed lines). To compute the plane, we determine UV coordinates from the direct interpolation and identify the grid points that correspond to the virtual triangle, containing the UV point. We emphasize that the final interpolation is carried out over four adjacent samples on the attribute map as in the direct approach, yielding bilinear interpolation. In rare cases where there is no such intersection, we discard this fragment, thereby preventing visual artifacts.

Figure 5 illustrates the result of our attribute interpolation. A small patch with nine points is considered along with a cubical environment map, featuring a grid structure painted on each of its faces.

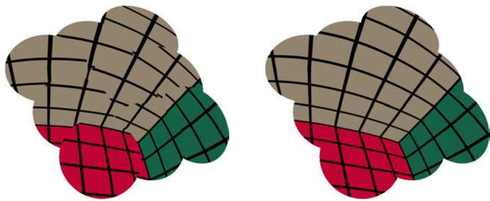


Figure 5: Illustration of rendering results achieved by our attribute interpolation using a nine splat patch with equal elevation values and varying normals (compare Fig. 1); Comparing the grid structures, the left figure shows discontinuities arising when using the direct interpolation method; applying the correct interpolation avoids these artifacts completely (right).

In the left image the discontinuous direct normal vector interpolation from Fig. 4 is used for acquiring texture from the environment map. Some discontinuities at splat boundaries are visible. On the right, the correct interpolation scheme for normal vectors yields a patch with continuously mapped environment texture.

3.6 Final Illumination Step

After determining the correct per-patch UV coordinates, we can directly utilize built-in trilinear texture interpolation functionality to smoothly reconstruct normal vectors, which we use for lighting techniques such as Phong shading. Due to the continuous per-fragment correct normals, we are able to implement reflection and refraction by using built-in cube mapping functionality. To set the final fragment color, we sum up contributions from lighting, texture mapping, and reflectance and refractance mapping. After setting the final fragment color, in contrast to previous approaches, our pipeline is complete after only one rendering pass. Moreover, we neither have to stress performance-hungry GPU functionality, such as multiple render targets, nor additional blending passes. With our per-fragment interpolation scheme, we obtain continuous attributes on the splats as well as between the splats, which is not possible with current techniques.

4 Results and Discussion

We demonstrate the performance of our pipeline with renderings of various 3D splat models; Rabbit and Balljoint are available at the Cyberware courtesy [7]; the Dragon model is available at the Stanford Scanning Repository [22], and the Shakyamuni statue was taken from the Konstanz Model Repository [12]. For constructing the patch-based model representations, we used the partitioning and resampling method, described in section 3. Our experiments are carried out using a Pentium4 platform, 3 GHz, 1 GByte RAM and a GeForce6800 Ultra graphics card. Our viewer system is implemented in C++ and OpenGL2.0. Our shaders are coded in Nvidia Cg shading language, using vp30 and fp30.

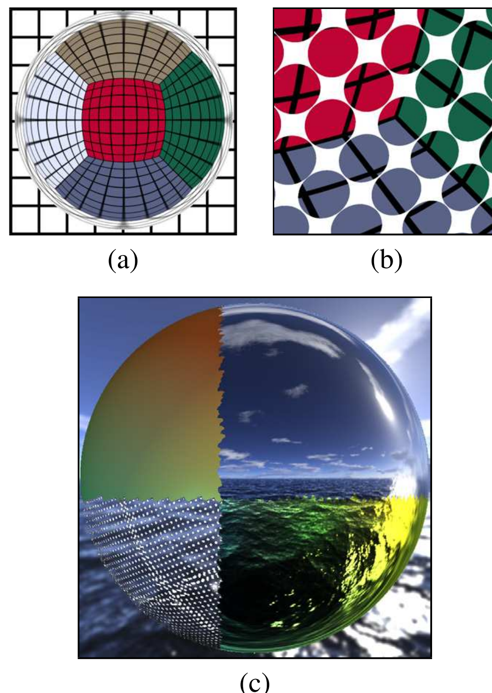


Figure 6: Rendering of the Sphere with 9858 splats; (a) shows reflectance using a cube map with differently colorized faces and black grid lines on each face; in the close-up view (b) the grid lines are correctly reconstructed normals on the splat surfaces using the interpolation method and hardware texture filtering; (c) combined shading effects as shown in Fig. 1.

For shading we use three light sources that are encoded in a cube map. We combine lighting with reflectance mapping using environment mapping (resolution of 256×256).

Figure 6 demonstrates the quality of reconstructed normals on a sphere with 9858 splats. In order to show the quality of interpolated normals, we use a cube map with colorized faces, containing a continuous grid. Subfigure (b) shows that the grid details are correctly reconstructed at a low patch sampling resolution. In (c) we show the summation of two shading components, such as per-fragment lighting with three light sources (upper left), re-

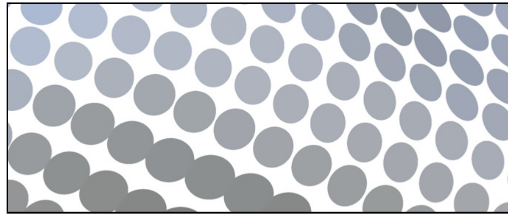
	Rabbit	Balljoint	Shakya	Dragon
# Patches	18	19	245	297
# Splats	71532	138654	268206	462814
# fps (direct)	69	45	33	30
# fps (correct)	55	38	26	24
# fps (standard^1)	58	36	23	18
# fps (standard^2)	32	20	12	11

Table 1: Rendering statistics for various splat models at a screen resolution of 1280×1024 using color depth of 24 bits; the frame rates in the third and fourth line correspond to rendering with the direct and with the correct interpolation method; the last two lines denote the frame rates using the traditional splat rendering technique in [3], whereas in (^{1}) the radii are assigned according to Eq. (1), as in the interpolation scheme; in (^{2}) the radii are doubled, such that a visual quality is achieved, which is comparable to the interpolation scheme.

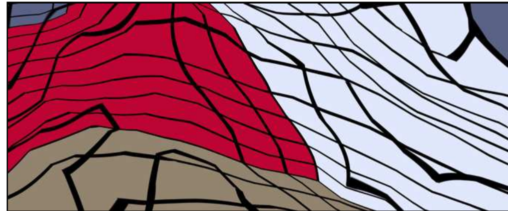
flectance mapping using a cube map (upper right), combined lighting and reflectance mapping (lower right), and constant splat colors with reduced radii (lower left).

Table 1 lists rendering statistics that we achieved with our renderer. The first and second row shows the number of patches obtained and the total number of splats, respectively. We show frame rates using the direct and correct interpolation schemes. We observe that the correct interpolation leads to a slight loss in rendering speed compared to the simple approach. We also compare the performance of our method to our implementation of the state-of-the-art renderer in [3]. We give frame rates for two different scenarios; firstly, with equal splat radii as used in the interpolation scheme; and secondly, with splats of doubled radius, so that each splat touches the centers of its four neighbors. This guarantees that neighboring points contribute their attributes over the *complete* area of each splat, which is required to achieve the same visual effects as in our scheme. A corresponding visual comparison for a closeup-view of the Dragon is shown in Fig. 7.

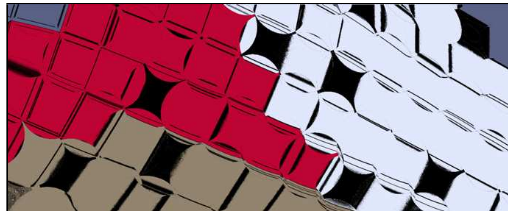
Figure 8 shows renderings of the Dragon model using per-pixel lighting (left) and reflectance mapping (right). The close-up views show the high visual quality, even at large zoom factors. Similar excellent quality is achieved by real-time rendering of the Shakyamuni statue (268206 splats). We show the same split view to illustrate the shader components (compare Fig. 1d). In contrast to Fig. 8, we used the simple direct interpolation in Fig. 9, which still provides visually pleasant results. We found that this behavior is typical, if the splat normals are close to those derived from the geometry of the corresponding elevation maps. We emphasize that this is the case for most practical models.



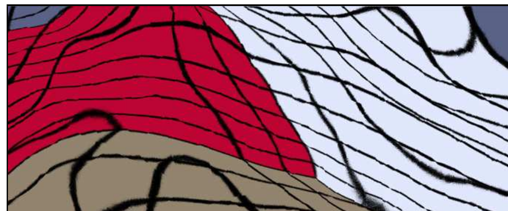
standard illumination, reduced radii



interpolation, radii set to r in (1)



standard blending, radii set to r in (1)



standard blending, radii set to $2r$ in (1)

Figure 7: Comparison of our method to standard splat blending with deferred shading [3] for a closeup-view of the Dragon with environment mapping as in Fig. 6a; the splat radii have to be doubled to achieve similar visual results as with the interpolation scheme. At the same time, the frame rate drops down significantly (see Tab. 1), since the number of fragments grows by a factor of four.

The figures in this section show that we achieve smooth reconstruction of attributes even along the patch boundaries. Since we let patches slightly overlap, we avoid gaps between patches in the rendered image. Moreover, in the overlapping regions of neighboring patches, geometry and attributes are consistently sampled from the same portions of the original model.

Our approach offers new perspectives for splat-based rendering. We incorporated advantageous interpolation as used for meshes in a splat-based environment. Our framework can be understood as

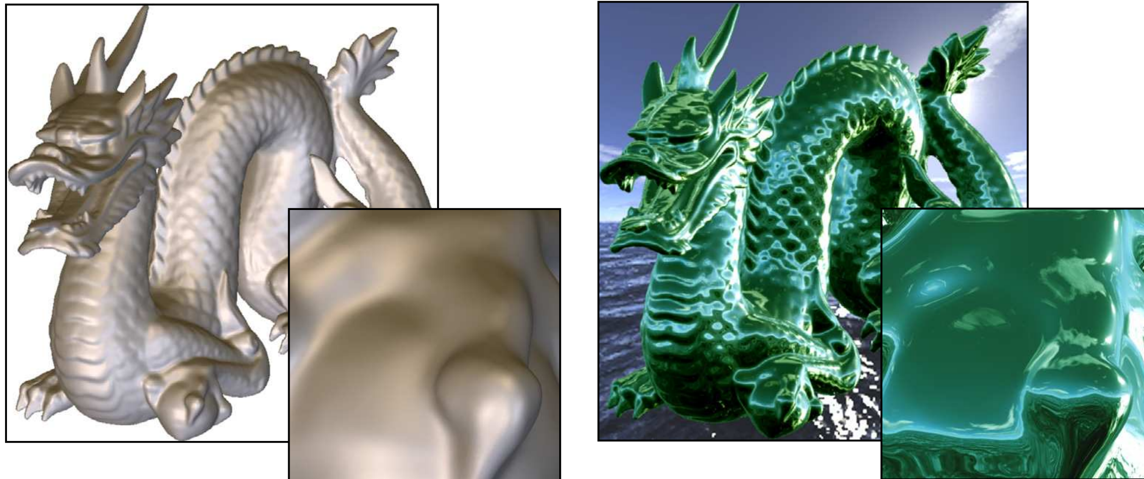


Figure 8: Real-time rendering of the Dragon model with 462814 splats in 297 patches using per-fragment lighting (left) and reflection mapping using cube maps (right); the close-up shows that normals are smoothly reconstructed even when splat radii become large.

an interim local simulation of a mesh-like structure without explicitly storing connectivity information. Our approach is still more general than just using a triangular mesh, since we are free to choose the final interpolation method. Consequently, we may use higher order neighborhoods to construct per-fragment attributes. Our pipeline also allows extending MIP mapping to geometry data (elevation maps). This offers new aspects for static level-of-detail rendering of complex geometries. The next generations of graphics hardware will incorporate primitive insertion in the vertex stream. This capability will further improve our approach with regards of dynamic LOD. Our framework moves away from the traditional concept of splatting: we separate splat shapes and position from attribute reconstruction through interpolation on the patch.

As in traditional hardware-accelerated splatting approaches, the bottleneck is still remaining in the fragment stage. However, the produced overdraw in our pipeline is typically lower that in previous blending approaches, since overlap is only required for gap free rendering, and not for attribute blending.

5 Conclusions

We presented a generic framework for rendering arbitrary surfaces using our interpolation-based splat renderer. Our contribution is to combine the most efficient components of state-of-the-art splat and mesh rendering. We achieve an improvement over previous approaches in that we interpolate splat attributes, yielding continuous normal vectors allowing us to implement advanced shader effects, which produces high-quality renderings. Additionally, our renderer consists of a simple rendering pipeline that is implemented in one single rendering pass on current GPUs.

The major advantages of our method are its simplicity of the attribute interpolation. Our interpolation scheme not only provides high-quality normal vectors, but also per-fragment interpolation texture coordinates, which is not possible with state-of-the-art splatting techniques. Please note that we achieve this quality using one single rendering pass. Moreover, we do not rely on resource-hungry multiple render targets.

Due to our underlying patch based representation, there are additional advantages: (1) patches that are directed away from the viewpoint can be discarded, (2) the clustering enables clipping of patches that are completely outside of the view volume, and (3) the scalability of the patch resolution allows implementing MIP mapping.

Currently, our approach requires an overhead, in that we rely on regularly sampled models, which are not given a priori, but rather have to be computed in a preprocessing step. However, in most applications, geometry is fixed, and thus the degree of freedom of unorganized point sets is not crucial in these cases. We will address this issue in future work.

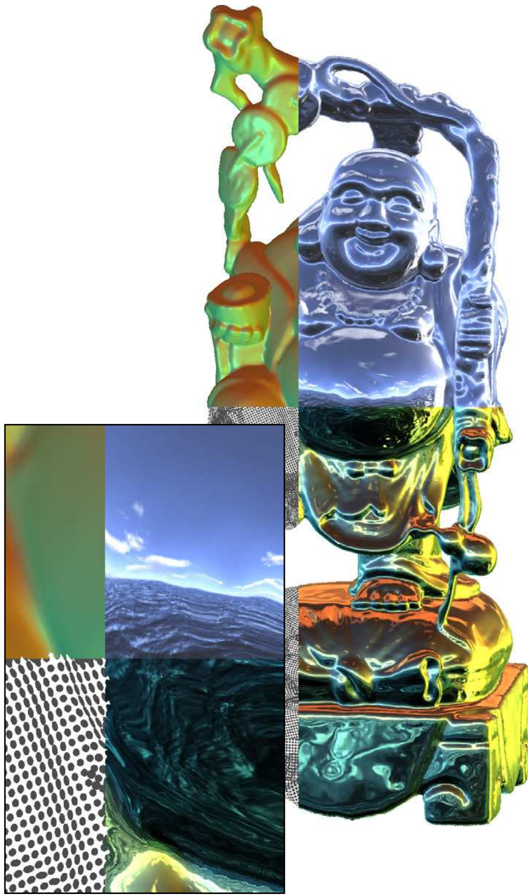


Figure 9: Rendering of Shakyamuni with 268206 splats in 245 patches; the various shown shader effects correspond to Fig. 1(d). Please note, that we still achieve smooth rendering, even when the splats cover a relatively large number of pixels, while the splat overlap is minimal.

References

[1] M. Alexa and A. Adamson. On normals and projection operators for surfaces defined by point sets. In *Proc. Symposium on Point-Based Graphics*, pages 149–155, 2004.

[2] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Computer Graphics and Visualization*, 9(1):3–15, 2003.

[3] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proc. Symposium on Point-Based Graphics*, pages 17–24, 2005.

[4] M. Botsch and L. Kobbelt. High-quality point-based rendering on modern GPUs. In *Proc. Pacific Graphics*, pages 335–343, 2003.

[5] M. Botsch, M. Spernat, and L. Kobbelt. Phong splatting. In *Proc. Symposium on Point-Based Graphics*, pages 25–32, 2004.

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

[7] Cyberware sample models archive, Apr. 2006, http://www.cyberware.com/samples/.

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

[9] G. Guennebaud, L. Barthe, and M. Paulin. Dynamic surfel set refinement for high-quality rendering. *Computers & Graphics*, 28(6):827–838, 2004.

[10] G. Guennebaud, L. Barthe, and M. Paulin. Interpolatory refinement for real-time processing of point-based geometry. *Computer Graphics Forum*, 24(3):657–667, 2005.

[11] P. Guennebaud and M. Paulin. Efficient screen space approach for hardware accelerated surfel rendering. In *Proc. Workshop on Vision, Modeling, and Visualization*, pages 485–493, 2003.

[12] Konstanz 3D Model Repository, Apr. 2006, http://www.inf.uni-konstanz.de/cgip/projects/surfac/.

[13] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report TR 85-022, Computer Science Department, University of North Carolina at Chapel Hill, January 1985.

[14] W. Matusik, H. Pfister, R. Ziegler, A. Ngan, and L. McMillan. Acquisition and rendering of transparent and refractive objects. In *Proc. Workshop on Rendering*, pages 267–278, 2002.

[15] T. Ochotta and S. Hiller. Hardware rendering of 3D geometry with elevation maps. In *Proc. International Conference on Shape Modeling and Applications*, pages 45–56, 2006.

[16] T. Ochotta and D. Saupe. Compression of point-based 3D models by shape-adaptive wavelet coding of multi-height fields. In *Proc. Symposium on Point-Based Graphics*, pages 103–112, 2004.

[17] M. Pauly and M. Gross. Spectral processing of point-sampled geometry. In *Proc. ACM SIGGRAPH*, pages 379–386, 2001.

[18] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface elements as rendering primitives. In *Proc. ACM SIGGRAPH*, pages 335–342, 2000.

[19] L. Ren, H. Pfister, and M. Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum*, 21(3):461–470, 2002.

[20] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In *Proc. ACM SIGGRAPH*, pages 343–352, 2000.

[21] M. Sainz and R. Pajarola. Point-based rendering techniques. *Computers & Graphics*, 28(6):869–879, 2004.

[22] Stanford 3D scanning repository, Apr. 2006, http://graphics.stanford.edu/data/3Dscanrep/.

[23] T. Weyrich, H. Pfister, and M. Gross. Rendering deformable surface reflectance fields. *IEEE Trans. on Visualization and Graphics*, 11(1):48–58, 2005.

[24] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. Surface splatting. In *Proc. ACM SIGGRAPH*, pages 371–378, 2001.

[25] M. Zwicker, J. Räsänen, M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting. In *Proc. Graphics Interface*, pages 247–254, 2004.