Multi-View Point Splatting

Thomas Hübner[*] Yanci Zhang[†] Renato Pajarola[‡]

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

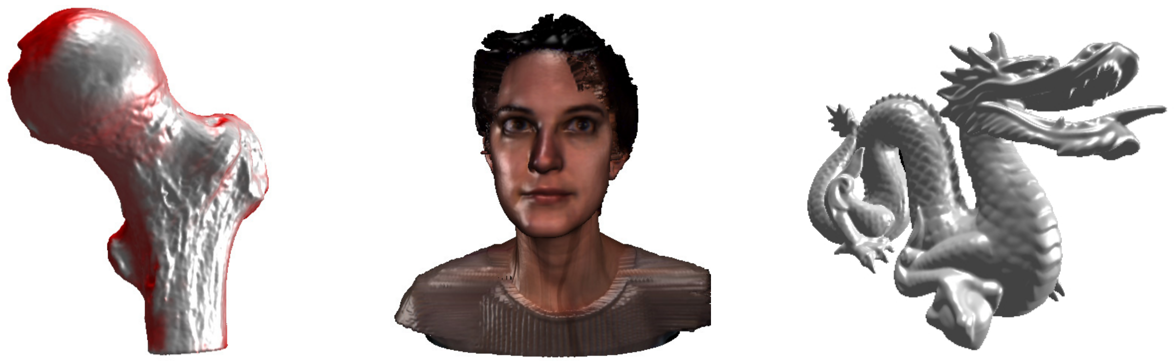


Figure 1: Different point data sets rendered with our multi-view splatting algorithm. Performance results compared to the standard multi-pass approach are shown for 512^2 window and N=8 views. a) Balljoint 137k points. b) Female 303k points. c) Dragon 1,100k points.

Abstract

The fundamental drawback of current stereo and multi-view visualization is the necessity to perform multi pass rendering (one pass for each view) and subsequent image composition + masking for generating multiple stereo views. Thus the rendering time increases in general linearly with the number of views.
In this paper we introduce a new method for multi-view splatting based on deferred blending. Our method exploits the programmability of modern graphic processing units (GPUs) for rendering multiple stereo views in a single rendering pass. The views are calculated directly on the GPU including sub-pixel wavelength selective views. We describe our algorithm precisely and provide details about its implementation. Experimental results demonstrate the performance advantage of our multi-view point splatting algorithm compared to the standard multi-pass approach.

CR Categories: I.3.m [Computer Graphics]: Miscellaneous—Multi-View Splatting Algorithm I.3.1 [Computer Graphics]: Hardware Architecture—Three-Dimensional Displays I.3.3 [Computer Graphics]: Picture/Image Generation—Display, Viewing Algorithms

Keywords: point based rendering, auto-stereoscopic visualization, multi-view rendering, hardware acceleration, GPU processing

[*]e-mail: huebner@ifi.unizh.ch
[†]e-mail: zhang@ifi.unizh.ch
[‡]e-mail:pajarola@acm.org

1 Introduction

Multi-view auto-stereoscopic display systems [Dodgson 2005] provide greatly enhanced spatial understanding of 3D data through perceptual cues for stereo- and motion-parallax. However, despite that auto-stereoscopic displays are becoming a common technology, spatial visualization has received far less attention in the field. In multi-view display systems, performance is generally a significantly increased bottleneck, as compared to single-view displays multiple images have to be rendered simultaneously for every frame. An extremely high computation and graphics power is required to do so with conventional approaches. Without the exploitation of similarities, rendering two views (for stereo) can require already twice the rendering time t of a mono-view rendering algorithm [He and Kaufman 1996; Ebert et al. 1996; Koo et al. 1999; Portoni et al. 2000; Miniel et al. 2004; Wan et al. 2004]. Correspondingly, an N-view display requires a rendering time of up to $N \cdot t$. Even exploiting modern graphics hardware (GPU) acceleration, this overhead prohibits effective multi-view rendering of large geometric data sets.

Point-based rendering (PBR) has shown to be a powerful alternative to polygon based graphics [Levoy and Whitted 1985; Gross 2001; Pfister and Gross 2004]. Research in point-based graphics has mostly concentrated on efficient rendering algorithms, see also surveys [Sainz et al. 2004a; Kobbelt and Botsch 2004; Sainz and Pajarola 2004], modeling with points (e.g. [Alexa et al. 2001; Zwicker et al. 2002; Pauly et al. 2003; Adams and Dutre 2004; Botsch and Kobbelt 2005]) as well as capturing and processing of point data (e.g. [Liu et al. 2002; Rusinkiewicz et al. 2002; Mitra and

Nguyen 2003; Sainz et al. 2004b; Weyrich et al. 2004; Sadlo et al. 2005; Pajarola 2005]). Efficient rendering of points in the context of immersive, auto-stereoscopic multi-view display systems has not been addressed in the past.

In this paper we address the problem of efficiently generating N views simultaneously in the context of interactive PBR, and we present a solution that exploits the programmability of modern GPUs to calculate the multiple views directly on a per-fragment basis. Since our new approach renders the geometry only once per frame, our solution renders the point samples in $K \cdot t$ time independent of the display resolution and the number N of views. The constant K indicates a commonly used sub-pixel resolution, i.e. different views for RGB with $K = 3$.

After related work in Section 2, in Section 3 we first provide a theoretical framework for multi-view auto-stereoscopic displays together with the standard approach used for generating spatial images for such displays. This is followed by an in-depth description of our new point splatting implementation in Section 4. Section 5 reports experimental performance results for different standard data sets and implementations. Our paper is concluded in Section 6.

2 Related Work

The point splatting technique as described in [Pfister et al. 2000; Zwicker et al. 2001] is the most common approach to point-based rendering and allows for efficient hardware (GPU) accelerated high-quality rendering. A wide range of GPU-accelerated point splatting algorithms such as [Ren et al. 2002; Botsch and Kobbelt 2003; Zwicker et al. 2004; Botsch et al. 2004; Pajarola et al. 2004; Botsch et al. 2005] have been proposed in the past and are surveyed in [Sainz et al. 2004a; Kobbelt and Botsch 2004; Sainz and Pajarola 2004]. Our multi-view point-splatting algorithm is based on the same concepts of rasterizing and blending projected disks in image space. In addition to conventional point splatting methods, however, our algorithm generates fragments for multiple views within one and the same rendering pass.

Specific rendering algorithms for multi-view auto-stereoscopic displays have not been proposed so far, however, methods used for two-view stereo rendering [Hodges and McAllister 1993; Hill and Jacobs 2006] can be multiplied and applied to N-view auto-stereo rendering. In stereo rendering environments, generally the displayed 3D data set is rendered twice, once for each left/right eye view. Consequently, as done to date the N views need to be rendered individually, and combined and masked for a multi-view auto-stereoscopic display [Portoni et al. 2000].

SGI Japan recently introduced the *interactive stereo library* (ISL) [SGI]. ISL is a thin layer on top of OpenGL supporting a wide variety of auto-stereoscopic displays. This library provides automatic rendering of N views, together with compositing and masking for final display. Except for shielding the user from implementation details of the specific display, the underlying methods are unchanged. Hence N views are rendered at $N \cdot t$ cost individually into N full-resolution frame buffers, despite the fact that most pixels are masked out later in the compositing and masking step. In ISL additional memory is required to store N frame buffers together with N compositing masks, but as it is not yet shader based, an acceleration for multi-view compositing through fragment shaders can be expected.

3 Multi-View Rendering

3.1 Auto-Stereoscopic Displays

Multi-view auto-stereoscopic displays provide a *spatial image* without requiring the user to use any special device [Dodgson 2005]. A spatial image is a 3D image that appears to have ”real” depth, that is, cues are provided to the human visual system to derive the depth of the displayed 3D data. To avoid confusion, we will use the terminology spatial if we refer to these images. A spatial image can be created by presenting different views of the 3D data independently to each eye and thereby simulating stereopsis. Spatial images can dramatically improve the spatial understanding and interpretation of complex three-dimensional structures. A general introduction to auto-stereoscopic 3D display types and technologies is given in [Dodgson 2005]. Here we focus on passive auto-stereoscopic 3D displays which generate multiple views for multiple possible observers and view positions, in contrast to active systems which track the user and generate exactly two views for each tracked observer position.

Multi-view auto-stereoscopic displays generally incorporate stereo as well as horizontal motion parallax. As shown in Figure 2, the infinite number of views a observer can see in the real world is partitioned into a finite number of available viewing zones. An auto-stereoscopic display emits the different views directionally-dependent into the viewing space in front of the display system. Each view is constant, or at least dominant, for a given zone of the viewing space. The observer perceives a spatial image as long as both eyes are in the viewing space, and observe the image from different view zones respectively. Changes in the observer's position result in different spatial depth perceptions. A feature of multi-view auto-stereoscopic displays is that multiple observers can be accommodated simultaneously, each having a different spatial perception according to his point of view in the viewing space.

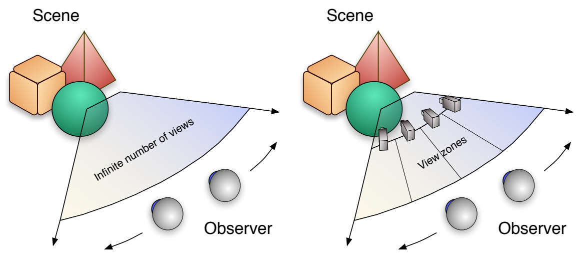


Figure 2: Multi-view auto-stereoscopic display principle: Discretization of an infinite view space into a finite number of view zones.

Modern auto-stereoscopic displays are mostly based on conventional flat-panel LCDs. With the help of optical filter elements, barriers and lenses, RGB-pixels are restricted and emitted into different zones in the viewing space [Dodgson 2002; Schmidt and Grasnick 2002; Dodgson 2005]. Two major optical filter elements are used today: lenticular sheets and wavelength-selective filter arrays. A lenticular sheet consists of long cylindrical lenses. They focus on the underlaying image plane and are aligned so that each viewing zone sees different sets of pixels from the underlaying image plane. Lenticular sheets provide a different view for each pixel for a given eye position [Hodges and McAllister 1993; Dodgson 2005]. Wavelength-selective filter arrays are based on the same principle, except that the lenses are diagonally oriented and each of the three color channels of a RGB-pixel corresponds to a different view zone. Therefore, wavelength-selective filter arrays provide a sub-pixel resolution of view zones [Schmidt and Grasnick 2002; Lee and Ra 2005].

As illustrated in Figure 3, the lens arrays distort the optics of the LCD panel such that each element – a pixel or sub-pixel RGB component in lenticular sheets or wavelength-selective filters respectively – corresponds to a certain view zone. Generally, the lens optics is calibrated such that at a given eye distance the different zones line up adjacently without mutual interference. The separation between views at this distance is matched to an average eye separation. Hence the observer can move his head sideways at this distance, within limits of the covered viewing space, and experiences stereo and motion parallax as each eye sees a different image view accordingly.

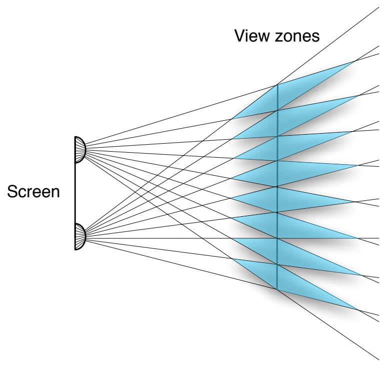


Figure 3: Viewing zones converge at a given observer distance.

For a given horizontal pixel resolution m of the LCD panel, multiple views can only be generated at the expense of reduced spatial screen resolution using *spatial multiplexing*. In principle, for generating N different views, the horizontal display resolution is split by the number N of views. Pixels are interleaved horizontally in such a way that pixel x basically corresponds to view zone $x \mod N$. However, using wavelength-selective filters, the reduction in horizontal resolution is limited to

$$X_{Res} = \frac{m}{N} \cdot 3, \qquad (1)$$

because each pixel's RGB components correspond to three different view zones. Equation 1 demonstrates the advantage of wavelength-selective filter arrays. The use of sub-pixel view directions causes less degradation of the horizontal resolution for multiple views.

In this paper we explain our multi-view point splatting algorithm in the context of auto-stereoscopic displays featuring such wavelength-selective filter arrays. Though the presented method can easily be adopted for different filter arrays. In fact, without sub-pixel view resolution, the performance of our algorithms increases by up to a factor of three compared to conventional multi-view rendering.

3.2 View Geometry

In Figure 4 the viewing configuration of a single-view setup is illustrated with a standard view frustum capped by the near and far clipping planes. The corresponding 2D image resulting from perspective projection is not considered a spatial image in the sense as introduced earlier. Nevertheless, this 'flat' image can contain important basic depth cues such as perspective distortion, visibility occlusion and distance attenuation. In contrast, a multi-view configuration as shown in Figure 5 provides additional depth cues such as stereo and motion parallax. The visible object space is defined by the intersection of the mono-view frusta of all individual views. This multi-view frustum is defined by the number N of views and the used *intraocular distance*. The focal plane where all views converge defines the perceived spatial distance of the multi-view display.

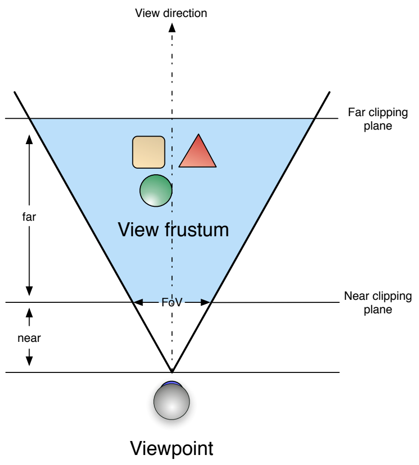


Figure 4: Single-view frustum with near and far clipping planes.

As shown in Figure 5, each perspective view is generated from a translational offset virtual camera placement which corresponds to different off-axis asymmetric sheared view frusta with parallel view directions. A spatial image captures the perspective projection of all N views, hence contains multiple perspective images. As also indicated in Figure 3, the focal plane of an auto-stereoscopic display device is well defined by the distance where the viewing zones converge. To match a multi-view rendering configuration to the physical display for optimal parallax effects, and for simplicity of the discussion, we set the *focal distance* of the viewing configuration in Figure 5 to the convergence distance indicated by the display device.

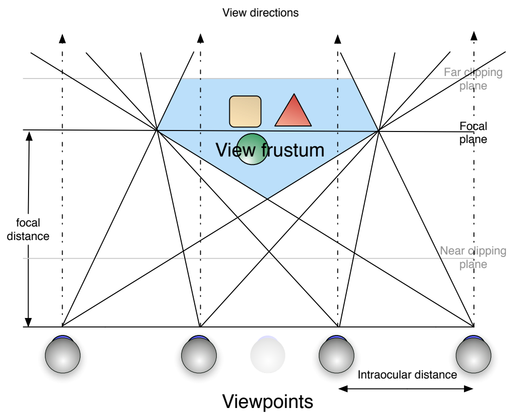


Figure 5: Visible multi-view frustum as intersection of individual off-axis asymmetric sheared view frusta.

A spatial image provides stereo parallax depth cues in that for a given pixel on the screen it presents a color originating from different spatial positions to the left and right eye-view respectively, see also Figure 6. The difference of the image presented to the left and right eye depends on the distance of the visible object from the focal plane and is resolved per pixel. If the spatial image contains information from more than two views then motion parallax depth cues are supported in addition to stereo parallax. Motion parallax refers to the observation that small movements of the eye position reveal small changes in visibility occlusion (see also [McAllister 2002]).

Let us call the plane in which the virtual cameras of the N different views are placed the camera plane since an observer can experience the best stereo effect from views on this plane. Given the camera plane at a focal distance fd from the focal plane, the point q_i of the splat visible in a view v_i depends on the normal n and the center position p of the splat as illustrated in Figure 6. The contributing pixel values of each view originate hence from different positions on the splat. Furthermore, for multiple viewpoints the projections of the same pixel may not necessarily all be located on the (same) surface element.

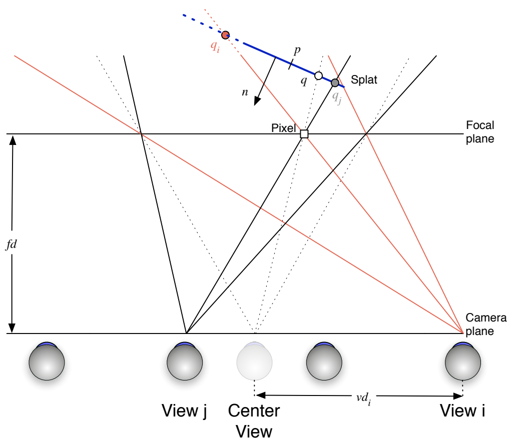


Figure 6: Multi-view splat parallax geometry.

Our multi-view point splatting algorithm detailed in Section 4 is built on the observations outlined in this section and in Figure 6. On a per-pixel basis, an intersection with a point splat is generated for each view. For wavelength-selective filter arrays, 3 such intersections are taken into account per fragment, one for each RGB component. The view assignment to pixels, or RGB channels, is further explained below.

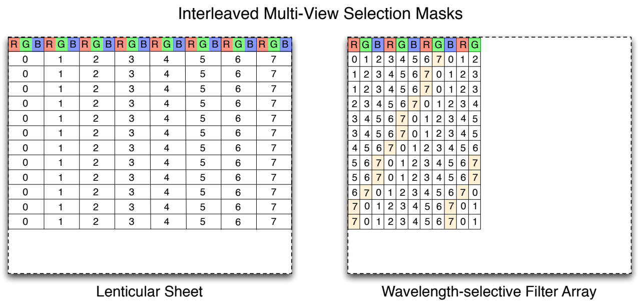
3.3 Image Generation Method

As we assume a passive auto-stereoscopic 3D display system, the number of observers and positions are not known. Hence each viewing zone may equally likely be visible to the observer and we need to generate all N views simultaneously for each frame. For each view we need to render a separate perspective image of the 3D data set. Eventually, the spatial image is generated by combining the perspective images from N different camera angles.

The conventional approach of a multi-view rendering system, e.g. such as [SGI], [Portoni et al. 2000], [Schmidt and Grasnick 2002] and [Miniel et al. 2004] is to render the 3D data in N passes using N different perspective (off-axis asymmetric sheared) view frustum configurations according to the multi-view setup as indicated in Figure 5. The resulting N images must then be combined into a spatial image conforming to the auto-stereoscopic display device. For lenticular sheets and wavelength-selective filter arrays, this combination is achieved by masking the (sub-)pixels of each view according to the multi-view mask as shown in Figure 7. This mask is initialized once for the used N views during the render setup phase. After rendering the N views to N target images I_i, the final spatial image is combined by

$$I(x,y) = I_{\text{Mask}(x,y)}(x,y). \tag{2}$$

For lenticular sheet displays the function Mask(x,y) is given by $x \bmod N$. For wavelength-selective filter arrays the masking function is only slightly more complex as it incorporates masking of individual RGB color components per pixel and includes a diagonal shift of the views as illustrated in Figure 7.



Interleaved Multi-View Selection Masks
Lenticular Sheet
Wavelength-selective Filter Array

R G B	R G B	R G B	R G B	R G B	R G B	R G B	R G B
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

R	G	B	R	G	B	R	G	B	R	G
0	1	2	3	4	5	6	7	0	1	2
1	2	3	4	5	6	7	0	1	2	3
1	2	3	4	5	6	7	0	1	2	3
2	3	4	5	6	7	0	1	2	3	4
3	4	5	6	7	0	1	2	3	4	5
3	4	5	6	7	0	1	2	3	4	5
4	5	6	7	0	1	2	3	4	5	6
5	6	7	0	1	2	3	4	5	6	7
5	6	7	0	1	2	3	4	5	6	7
6	7	0	1	2	3	4	5	6	7	0
7	0	1	2	3	4	5	6	7	0	1
7	0	1	2	3	4	5	6	7	0	1

Figure 7: Multi-view masks for lenticular sheets and wavelength-selective filter arrays ($N = 8$).

It is clear that the conventional multi-view rendering solution requires rendering of N perspective images followed by N-fold image masking and compositing into the final spatial image. This leads basically to a N times cost increase compared to single-view rendering. If pixel-fill rate is a limiting factor either during rendering (e.g. complex light shaders) or during multi-view image compositing and masking (e.g. compositing 8 1900×1600 images requires transferring 70MB/frame), the intermediate target images can optionally be rendered at sub-resolution. During compositing and masking the full-resolution can be restored by up-sampling at the expense of blurring artifacts. However, geometry processing cost overhead remains at $N - 1$ times, which will be the major limiting factor for large data sets.

In the following section we present our multi-view point splatting algorithm which requires rendering the point samples only once, and no subsequent image masking, in contrast to N times as for conventional multi-view display. Since the geometry is rendered only once, the full resolution can be supported and sub-resolution rendering is not necessary.

4 Multi-View Point Splatting

4.1 Point Splatting

A set of overlapping point samples $\mathcal{S}$ covering a surface is rendered by smoothly interpolating the projected splats in image space, that is for each fragment the contribution of multiple overlapping splats are weighted and blended together. This is achieved by rasterization of a (perspectively projected) disk and using α-blending. As splats from occluded surface layers must not contribute to this interpolation, a fuzzy visibility occlusion test, commonly referred to as ε-z-buffering, is applied.

The prevalent implementation solution is to apply a $2+1$-pass rendering algorithm. First, all splats are rasterized at an ε offset to initialize the depth buffer. Then, using this depth buffer read-only, the splats are rendered again and per-fragment α-blending is applied for the weighted interpolation. Third, in an image normalization post-process the final fragment color is generated. Further detailed information on this basic point splatting technique is beyond the scope of this paper and can be found in [Sainz et al. 2004a; Kobbelt and Botsch 2004; Sainz and Pajarola 2004] and its surveyed articles. Some more details of our multi-view implementation are given in Section 4.4.

4.2 Splat Intersection

A perspective accurate projection of an object-space disk to image-space can be achieved via projective mapping [Zwicker et al. 2004], a parametrized distance function or by per-pixel ray-disk intersection tests. We opt for the latter two which can be implemented more efficiently for multi-view rendering using vertex and fragment shaders. We first outline the parametrized perspective splat intersection, which conceptually has an elegant solution supported by hardware accelerated interpolation of texture coordinates, followed by a simple but efficient per-pixel ray-intersection approach.

Parametrized Splat Intersection Given a point sample $\mathbf{p}$ and its normal $\mathbf{n}$ as in Figure 8, the splat plane can be parametrized by

$$P(r,s) = \mathbf{p} + r \cdot \mathbf{u} + s \cdot \mathbf{v} \tag{3}$$

given $\mathbf{u} = \|(0, -n_z, n_y)\|$ and $\mathbf{v} = \mathbf{n} \times \mathbf{u}$. The splat disk with radius R is then defined by points with $\sqrt{r^2+s^2} \leq R$.

With all coordinates specified in the camera coordinate system, given the viewpoint $\mathbf{vp}$ and a pixel in image-space with coordinates $\mathbf{x} = (x, y, fd)$ its corresponding projection $\widehat{\mathbf{x}}$ on the splat plane (see also Figure 8) is implicitly determined by $\mathbf{n} \cdot (\widehat{\mathbf{x}} - \mathbf{p}) = 0$ and $\widehat{\mathbf{x}} = \mathbf{vp} + l \cdot (\mathbf{x} - \mathbf{vp})$, hence

$$\widehat{\mathbf{x}} = \mathbf{vp} + \frac{\mathbf{n} \cdot (\mathbf{p} - \mathbf{vp})}{\mathbf{n} \cdot (\mathbf{x} - \mathbf{vp})} \cdot (\mathbf{x} - \mathbf{vp}), \tag{4}$$

and its plane parametrization with respect to Equation 3 is the mapping $M : \widehat{\mathbf{x}} \rightarrow (r, s)$

$$\begin{aligned} r &= (\widehat{\mathbf{x}} - \mathbf{p}) \cdot \mathbf{u} \\ s &= (\widehat{\mathbf{x}} - \mathbf{p}) \cdot \mathbf{v}. \end{aligned} \tag{5}$$

As indicated in Figure 8, the point splat disk can now efficiently be projected by drawing a quad $Q = (\mathbf{q}^A, \mathbf{q}^B, \mathbf{q}^C, \mathbf{q}^D)$ in image-space, projecting and parametrizing its corners according to Equations 4 and 5, and for each rasterized pixel testing its parameters r, s, which can be interpolated across the quad. The distance $w = \sqrt{r^2+s^2}$ is used to test against the splat radius R, and if $w \leq R$ used to calculate a blending weight factor that indicates the contribution of that pixel for splat interpolation.

The parameters $M(\widehat{\mathbf{x}})$ are a linear combination of the parameters $M(\widehat{\mathbf{q}}^{[A,B,C,D]})$ on the point splat plane. Thus we can implement the direct image-plane to plane-parameters mapping $\mathbf{x} \rightarrow (r, s)$ by linear interpolation from the quad corners using depth-corrected texture coordinates. This is achieved by drawing the image-space quad Q, specifying $(r, s, 0, \widehat{\mathbf{q}}.z)$ as the texture coordinates for each corner $\mathbf{q}$, with $\widehat{\mathbf{q}}.z$ being the distance of the corner's projection on the splat plane along the z-axis from the viewpoint.

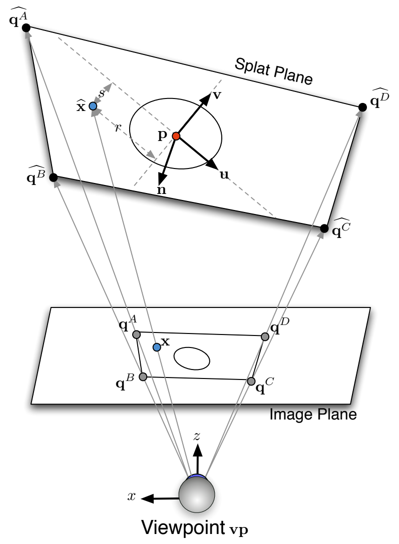


Figure 8: Parametrization of disk on splat plane.

Per-Pixel Ray-Splat Intersection The second method we considered for calculating the accurate projection of a point splat to image-space is a per-pixel ray-disk intersection test. That is the ray $\mathbf{x} - \mathbf{vp}$ from a viewpoint $\mathbf{vp}$ through an image-plane pixel $\mathbf{x}$ is intersected directly with the splat plane according to Equation 4. The splat plane intersection $\widehat{\mathbf{x}}$ then defines the distance as $w = |\widehat{\mathbf{x}} - \mathbf{p}|$.

In contrast to the parametrized intersection, where the splat plane intersection is computed for the quad corners and then their parameters r, s are interpolated and tested across the quad, the ray-plane intersection computes the image to plane projection and its distance to the splat center individually for each pixel.

4.3 Multi-View Splat Projection

In the context of a general multi-view configuration, i.e. the geometry is not coinciding with the focal plane (see also Figure 6), a point splat projects differently for the different viewpoints as illustrated in Figure 9. Therefore, the splat projections, with respect to Figure 8, for multiple views must be covered by an extended quadrilateral in image-space. As the views are only horizontally separated (Section 3), a wide quad covering all possible projections is drawn as shown in Figure 9.

The enlarged multi-view splat covering quad can be computed as follows, given the maximal view eccentricity vd, see also Figure 10. We consider the worst-case orientation of a splat $\mathbf{p}$ with its normal $\mathbf{n}$ oriented towards the viewer orthogonal to the focal and camera planes. We can see that the relation $vd : \mathit{off} = d : (d - fd)$ holds, and thus we have the offset $\mathit{off} = vd(d - fd) \cdot d^{-1}$ that has to be added horizontally to the projected splat radius r. Vertically, the quad only has to cover the height of the perspectively projected point splat. Therefore, we have the quad corners given by:

$$\begin{aligned} \mathbf{q}.x &= (\mathbf{p}.x \pm (r + vd \cdot (\mathbf{p}.z \cdot fd^{-1} - 1))) \cdot fd/\mathbf{p}.z \\ \mathbf{q}.y &= (\mathbf{p}.y \pm r) \cdot fd/\mathbf{p}.z \\ \mathbf{q}.z &= fd \end{aligned} \tag{6}$$

To benefit from the hardware supported rasterization and interpolation of the splat plane parametrization as outlined previously in

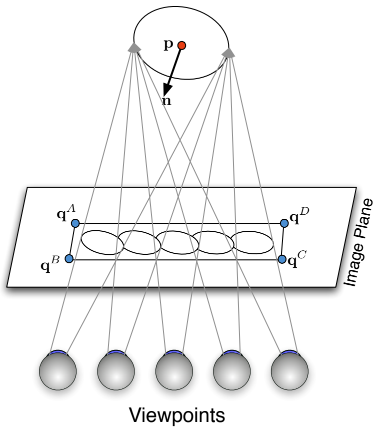


Figure 9: Multi-view dependent splat projection.

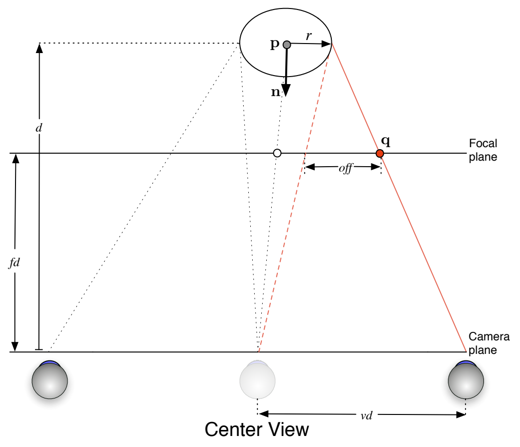


Figure 10: Multi-view offset.

Section 4.2, we can generate multiple parameter sets for the image-plane aligned quad $Q = (\mathbf{q}^A, \mathbf{q}^B, \mathbf{q}^C, \mathbf{q}^D)$, one for each of the N views. Hence for each quad corner $\mathbf{q}$ in image-space, we compute the image-plane to plane-parameters mapping $\mathbf{q} \to (r_i, s_i)$ and its z-distance $\widehat{\mathbf{q}}_i.z$ for each view i.

Labeling the corners by $L = A \dots D$ and the views by $i = 0 \dots N-1$, we basically draw an image-plane aligned quad Q with N texture coordinate sets $(r_i^L, s_i^L, 0, \widehat{\mathbf{q}}_i^{\,L}.z)$ for each corner. The corner attributes (r_i^L, s_i^L) and depth $\widehat{\mathbf{q}}_i^{\,L}.z$ are computed according to Equations 4 and 5 during the vertex processing stage, while the per fragment inside-disk test, $r^2 + s^2 \leq R^2$, is performed during fragment processing. For each pixel, or even sub-pixel color component separation, the appropriate view and texture coordinates are selected for the corresponding view to be displayed. More detail is given in the following section.

In the case of per-pixel ray-splat intersection tests as described in Section 4.2, we limit the vertex processing stage to the drawing of an image-plane aligned quad Q. Ray-splat intersection tests and the inside-disk test are solely performed per-pixel or sub-pixel in the fragment stage.

4.4 Implementation

Our multi-view point rendering implementation follows the conceptual principle of rendering point splats using an ε-z-buffer visibility test combined with α-blending based interpolation of overlapping splats in image-space as basically introduced in [Pfister et al. 2000; Zwicker et al. 2001]. The basic GPU acceleration follows the principles of [Ren et al. 2002; Botsch and Kobbelt 2003; Pajarola et al. 2004] which render some geometry (quad, triangle, sprite) that covers the point splat disk in image-space. Rasterization of a projected disk is achieved through transparent α-masking of fragments outside the disk, and weighted α-blending (accumulation) of fragments inside the disk. The accumulated weighted color is normalized eventually, dividing by the sum of weights, to form the final image.

4.4.1 1+1 Pass PBR Algorithm

To avoid the commonly used 2+1 pass rendering process, which first initializes an ε-offset depth-buffer and is followed by the α-blending of the front-most visible point splats, we employ a recently introduced approach which processes the geometry in one rendering pass [Zhang and Pajarola 2006b; Zhang and Pajarola 2006a]. The basic idea is a *deferred blending* concept that delays the ε-z-buffer visibility test as well as smooth point interpolation to an image post-processing pass. As illustrated in Figure 11, if a given point set $\mathcal{S}$ is sufficiently partitioned into multiple non self-overlapping groups $\mathcal{S}_k$, with $\mathcal{S} = \bigcup_k \mathcal{S}_k$, overlapping splats in image-space can be avoided. For each group $\mathcal{S}_k$ a partial image I_k can be formed with fragment colors $\mathbf{c}_{\text{rgb}}(f)_k = \sum_{s_i \in \mathcal{S}_k} w_i(\mathbf{f}_i) \cdot \mathbf{c}_i$ and fragment weights $\mathbf{c}_\alpha(f)_k = \sum_{s_i \in \mathcal{S}_k} w_i(\mathbf{f}_i)$ where c_i and w_i are color and weight values respectively contributed to a fragment from a point splat i. The final complete rendering result can then be formed by an image compositing step over all partial images I_k. In order to create the non self-overlapping point groups, a *minimal graph coloring* algorithm is employed in the preprocessing stage. For more details on this algorithm the reader is referred to [Zhang and Pajarola 2006b; Zhang and Pajarola 2006a].

4.4.2 Multi-View Splatting Algorithm

The multi-view splatting algorithm follows the main steps of 1+1 pass PBR algorithm: dividing the points into multiple groups in a preprocessing stage, and then 1) rendering groups $\mathcal{S}_k$ to different textures to form partial images I_k in the geometry pass, followed by 2) eventually combining partial images together to achieve the final result in the image compositing pass.

The outline of our multi-view PBR algorithm is listed in Figures 12 and 13 and corresponds to a wavelength-selective sub-pixel resolution multi-view display system as described in Section 3.

Comparing to the original 1+1 pass PBR algorithm [Zhang and Pajarola 2006b; Zhang and Pajarola 2006a], our multi-view splatting algorithm is more complex due to the fact we have to handle multiple views simultaneously instead of just one single view. The features of our multi-view PBR algorithm are:

Drawing Splats: In the case of single-view splatting, there are many options to draw splats, such as point sprites, triangles, quads. But in the context of multi-view splatting, the splat projections of different views cover slightly different areas in the image plane. As introduced in section 4.3, an enlarged quad covering splat projections of all views is rendered. One solution to render such an enlarged quad is to use a point sprite which is sized big enough to

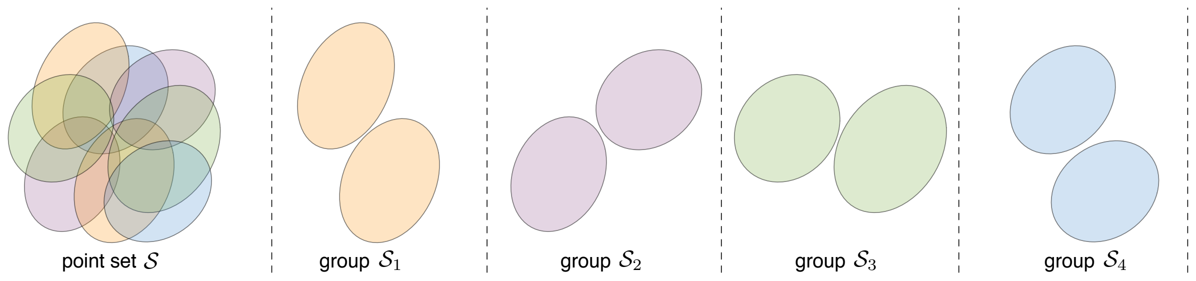


Figure 11: Separation of the input point set $\mathcal{S}$ into non-overlapping sub-sets $\mathcal{S}_k$.

Geometry Pass:

```
1  turn on $z$-test and $z$-update;
2  foreach group $\mathcal{S}_k$
3    clear $z$-depth and color of texture $I_k$;
4    render group $\mathcal{S}_k$ to texture $I_k$;
5    foreach $s_i \in \mathcal{S}_k$ do
6      calculate the corresponding corners of the quad according to Equation 6;
7      if use Parametrized Splat Intersection
8        calculate set of parametric coordinates for the $N$ views;
9      endif
10     transform and project the corners of the quad;
11     foreach generated fragment $f \in I_k$ do
12       determine sub-pixel views corresponding to current fragment;
13       if use Parametrized Splat Intersection
14         test ray-splat intersection according to parametric coordinates;
15       else
16         perform full ray-splat intersection calculations;
17       endif
18       calculate colors, kernel weights and depth values;
19       output averaged $z$-depth $\mathbf{c}_d(f)_k$;
20       pack colors and kernel weights, and output them to texture $I_k$;
21     endforeach
22   endforeach
23 endforeach
```

Figure 12: Geometry pass of our multi-view PBR algorithm.

Image Compositing Pass:

```
1  foreach $f \in I$ do
2    $\mathbf{c}_{\mathrm{rgb}}(f) = \mathbf{0}$;
3    $\mathbf{w}_{\mathrm{rgb}}(f) = 0$;
4    $d = \min_k(\mathbf{c}_d(f)_k)$;
5    for $k = 0$ to $K-1$ do
6      if $\mathbf{c}_d(f)_k \leq d + \varepsilon$ then
7        unpack color values to $\mathbf{c}_{\mathrm{rgb}}(f)_k$;
8        unpack weight values to $\mathbf{w}_{\mathrm{rgb}}(f)_k$;
9        $\mathbf{c}_{\mathrm{rgb}}(f) = \mathbf{c}_{\mathrm{rgb}}(f) + \mathbf{c}_{\mathrm{rgb}}(f)_k$;
10       $\mathbf{w}_{\mathrm{rgb}}(f) = \mathbf{w}_{\mathrm{rgb}}(f) + \mathbf{w}_{\mathrm{rgb}}(f)_k$;
11     endif
12   endfor
13   $\mathbf{c}_{\mathrm{rgb}}(f) = \frac{\mathbf{c}_{\mathrm{rgb}}(f)}{\mathbf{w}_{\mathrm{rgb}}(f)}$;
14 endforeach
```

Figure 13: Compositing pass of our multi-view PBR algorithm.

cover the enlarged quad. But according to Equation 6, the width of the enlarged splat is much bigger than its height, while point sprites are limited to square shapes which do not provide a sufficient tight bounding quad around the multi-view projected splat.

Therefore, we choose to actually draw an image-aligned quad. Note that Equation 6 is view-dependent, which means we can not calculate the quad vertices in a preprocessing stage. Now we choose to calculate the quad corners in a vertex shader according to Equation 6. In order to do that, we input the center of a splat 4 times to the vertex shader, including its position, radius, normal and color. Moreover, an index indicating the corner to be processed is attached to the point. In the vertex shader, we use this index and Equation 6 to calculate the corners of the extended quad, as indicated on the line 6 of Figure 12.

Employing Mask Map: According to the multi-view masks illustrated in Figure 7, there is one view $v = \mathrm{Mask}(x, y)$ or three views $v_{R,G,B} = \mathrm{Mask}_{R,G,B}(x, y)$ to consider for each fragment (x, y) for lenticular sheets or wavelength-adaptive filter arrays respectively. We consider the more complex sub-pixel wavelength-selective filter mask situation here. Hence before deciding on the contribution of a view v_i to a fragment we need to determine the needed views for the fragment position (x, y). The wavelength-selective filter mask in Figure 7 has a reoccurring pattern of size 8×12. Depending on the fragment (x, y) we can calculate the $xoffset$ and $yoffset$ inside this mask by Equations 7 and 8, which refer to the mask index of the first sub-pixel component (R). Fragment coordinates (x, y) in window space are available in the fragment shader through `WPOS`.

$$
\begin{aligned}
xoffset &= (3 \cdot x) \;\%\; 8 && (7)\\
yoffset &= (screenHeight - y) \;\%\; 12 && (8)
\end{aligned}
$$

The first view v_R of a fragment is computed by Equation 9 which considers the diagonal shifts of the wavelength-selective filter mask from Figure 7. The others are $v_G = (v_R + 1) \;\%\; 8$ and $v_B = (v_R + 2) \;\%\; 8$.

$$
v_R = (\lfloor \frac{1 + yoffset + \lfloor \frac{yoffset}{3} \rfloor}{2} \rfloor + xoffset) \;\%\; 8 \qquad (9)
$$

Complex Output: Comparing to single-view rendering algorithms, we have to keep the information from three different views simultaneously in a single fragment. For each view, we need to save:

- Color information (only one channel is required depending on the mask map in Figure 7);
- Weight value from the splatting kernel;
- Depth value which will be used in compositing pass;

For depth values, actually an averaged depth value for three views is employed in our implementation because the ε-z operation in composting pass render a high precision depth value unnecessary. But even with the averaged depth value used, we still need to save three different color and weight values for a single fragment. There are two methods to enable such complex image output during the geometry processing pass. The first one is to use a 32-bit floating point texture to store the output. The high precision texture allows us to use packing operations to pack four unsigned bytes or two half floating values into a single 32-bit floating value and then to unpack it in the image compositing pass. Note that packing and unpacking operations are up to date only supported in hardware by NVidia in combination with there own shading language Cg, and have to be implemented if other hardware or shading languages are employed. The second solution is to use multiple render targets which enable us to output the color and weight values to different textures. The drawback here is that even more texture lookup operations are required in the image compositing pass which may reduce the rendering performance. In our implementation, we adopt the first solution. Our strategy is:

- R channel: The averaged depth value.
- G channel: Packing color values of three different views as unsigned bytes to one 32-bit floating value.
- B channel: Packing weight values of the first two views as half floating values to one 32-bit floating value.
- W channel: Weight value of the third view.

Vertex Shader vs. Fragment Shader In our algorithm, we proposed two methods to fulfill the multi-view splatting, parametrized splat intersection and per-pixel ray-disk intersection. The basic difference between the two methods is the place where they do the ray-splat intersection.

For parametrized splat intersection, we calculate the parameterizations for all N views in the vertex shader and pass the N sets of parametric coordinates to the fragment shader, There we only have to do a simple test $r^2 + s^2 \leq R^2$ to decide whether the current fragment is inside the splat or not for the corresponding view.

For per-pixel ray-disk intersection, we mainly enlarge the quad to generate sufficient fragments in the vertex shader. No other calculation is done here so that we achieve a much simpler vertex shader than the first method at the cost of doing the full ray-splat intersection calculation for up to three views $v_{R,G,B}$ in the fragment shader.

5 Results

We have implemented our multi-view splatting methods in OpenGL using NVidia's Cg shading language. The presented results were generated on a 2.8Ghz CPU with NVidia GeForce 7800GTX GPU supported by 256 MB. The targeted multi-view auto-stereoscopic display uses a wavelength-selective filter array providing 8 views and has a resolution of 1900x1200.

In order to test our algorithm, we implemented the following four rendering algorithms for $N = 8$ views and compared their results:

- Standard multi-pass implementation (*8 MV*) which renders eight views in eight different passes. In each rendering pass, we use the render-to-texture technique to render one view to a texture and then composite the eight views to the final spatial image according to the wavelength-selective filter array;
- Single-pass vertex-based implementation (*GPU VS*) which calculates the parametric coordinates for all eight views in the vertex shader and only needs a simple test in the fragment shader to achieve the ray-splat intersection;
- Single-pass fragment-based implementation (*GPU FS*) which does the full calculation of ray-splat intersection in fragment shader for three views;
- Single-pass single-view implementation (*SV*) which renders only one view with the single geometry pass PBR algorithm.

Chart 1 shows the frame rates for generating 8-view spatial images using different window resolutions from 256 up to 1024. As expected the multi-pass implementation suffers significantly from 8 separate rendering passes. Depending on the window resolution the performance decreases by a factor up to ≈ 8, representing an immense performance decrease. Our *GPU FS* implementation improves over the multi-pass rendering by an average factor of 3 having at least $\frac{1}{3}$ of the single-view performance. This corresponds to the theoretical performance expected for rendering three views instead of one per-fragment. In most cases our *GPU FS* implementation even surpasses this result, performing with $\geq \frac{1}{2}$ of the single-view performance. Independent of the window resolution our fragment based multi-view implementation is always at least twice as fast as the standard multi-pass approach.

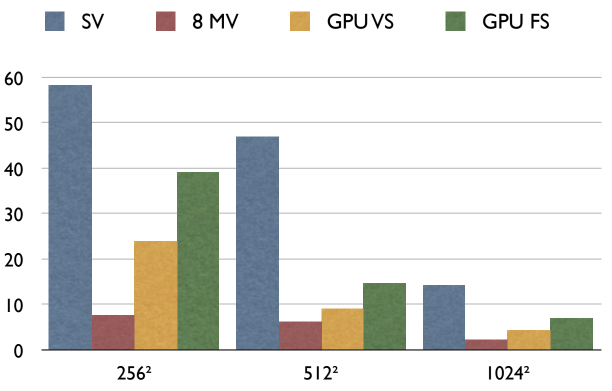


Chart 1: Performance comparison (fps) of single-view and multi-view PBR at different window resolutions (N=8 views, female dataset).

Pre-computing 8 views in the vertex stage and selecting the appropriate result in the fragment stage is an elegant conceptual solution, however, our current *GPU VS* implementation shows, that it can not keep up with the fragment based solution *GPU FS*. There are two possible reasons for this result:

- Optimized fragment processing: Graphics hardware manufacturers optimize graphics cards with focus on fragment processing. For example, the used GeForce 7800GTX has only 8 vertex shaders but provides 24 fragment shaders. Meaning that at least three times more fragments as vertices can be processed at a time.
- Handling of condition processing and arrays: We have to use extensive branching in the fragment stage because of the lack of variable array indices. Those indices need to be currently resolved at compile time and conditions require several computation cycles, thus adding to the reduced performance.

The performance comparison for different data sets is shown in Chart 2. Our multi-view implementations perform better with increasing data set sizes (see also Chart 3). Single-view PBR shows a strong dependance on the data set size while the multi-view performance reduces much less. It is important to note, that the performance of our multi-view implementations also depends on the user defined focal plane. The farther away an object is placed from the focal plane the more fragments will be generated by the multi-view covering quad. As one can see in Chart 2 for the balljoint model, the performance difference of our *GPU FS* compared to the single-view rendering is so large mainly because the model was placed

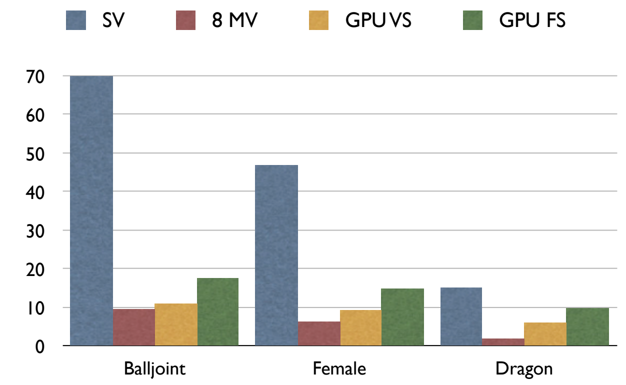


Chart 2: Performance comparison (fps) of single-view and multi-view PBR on different data sets (N=8 views, 512^2 window, Balljoint 137k points, Female 303k points, Dragon 1,100k points).

very close to the camera plane and far away from the focal plane in this test. This results in big disparity values, generating more fragments after the vertex stage and decreasing the overall performance. Nevertheless, the number of generated fragments for objects placed far away from the focal- and camera plane is limited by the maximum view eccentricity vd and the splat projection according to Equation 6.

As can be seen in Chart 3, our multi-view rendering gets better relatively to single-view rendering with increasing data set sizes. This is because with an increasing size of the data set, the point splats get smaller, and hence our multi-view methods perform increasingly better due to less overhead of fragment generation. In other terms, with high-resolution point data the overhead for multi-view rasterization of the enlarged quad disappears. Our *GPU FS* multi-view splatting algorithm achieves almost 70% of the performance of single-view rendering as shown in Chart 3, notably generating 8 different views simultaneously.

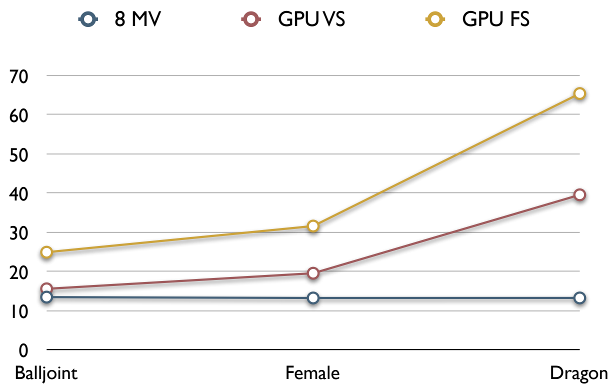


Chart 3: Performance comparison of multi-view PBR in percentage of single-view PBR for different data sets (N=8 views, 512^2 window, Balljoint 137k points, Female 303k points, Dragon 1,100k points).

Figure 14 finally shows an unaltered raster image of a spatial image rendered with our *GPU FS* multi-view splatting algorithm. The blurry appearance results from the wavelength-selective filter structure which causes the diagonal periodical shifts in pixels and RGB channels in accordance with Section 3.3.

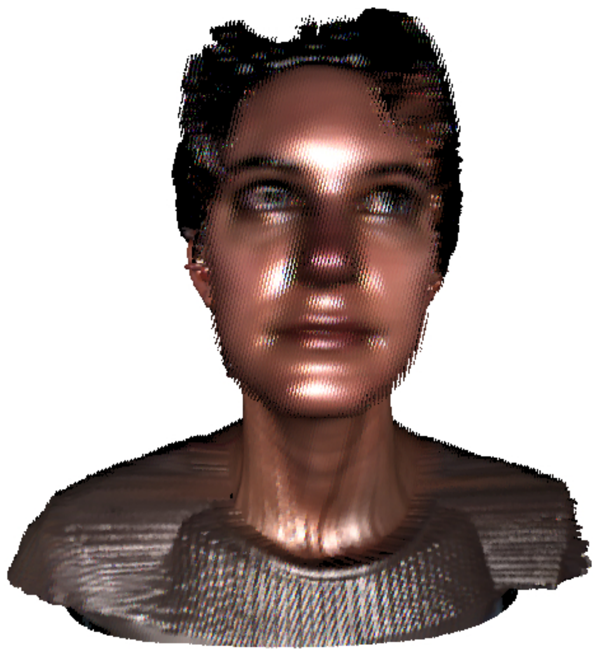


Figure 14: Spatial image, revealing the structure of the wavelength-selective filter array

6 Conclusion

In this paper we present a novel multi-view splatting algorithm based on deferred blending. We exploit the programability of modern GPUs to calculate multiple views directly on the GPU, in a single geometry rendering-pass. Two different implementation methods are proposed, one vertex-based and one which is fragment based. Both methods are compared with the performance of a standard multi-view and single-view visualization. The experimental results show, that both methods have a significant performance advantage compared to the standard multi-view multi-pass rendering. In particular, our fragment based multi-view implementation can render N-views with more than $\frac{1}{3}$ of the single-view performance compared to approximately $\frac{1}{N}$ for the standard multi-view solution. This is true for employing wavelength-selective filter arrays which use three views per fragment, and in fact approaches single-view performance for larger point data sets. For single-stereo we expect that our implementation reaches nearly the speed of single-view rendering. Future GPUs will reduce the gap between single- and multi-view visualization rendering even further by providing additional or more advanced fragment and vertex shaders. Furthermore, our vertex-based multi-view rendering method will dramatically benefit from these advanced shaders.

Acknowledgements

We would like to thank Newsight technology for their technical support for the 3D display device as well as the Stanford 3D Scanning Repository and Cyberware for providing the 3D geometry test data sets. We would also like to thank the Swiss National Science Foundation which partly supported this work by grant 200021-111746/1.

References

ADAMS, B., AND DUTRE, P. 2004. Boolean operations on surfel-bounded solids using programmable graphics hardware. In *Proceedings Symposium on Point-Based Graphics*, Eurographics, 19–24.

ALEXA, M., BEHR, J., COHEN-OR, D., FLEISHMAN, S., LEVIN, D., AND SILVA, C. T. 2001. Point set surfaces. In *Proceedings IEEE Visualization*, Computer Society Press, 21–28.

BOTSCH, M., AND KOBBELT, L. 2003. High-quality point-based rendering on modern GPUs. In *Proceedings Pacific Graphics 2003*, Computer Society Press, IEEE, 335–343.

BOTSCH, M., AND KOBBELT, L. 2005. Real-time shape editing using radial basis functions. *Computer Graphics Forum 24*, 3, 611–621. Eurographics 2005 Proceedings.

BOTSCH, M., SPERNAT, M., AND KOBBELT, L. 2004. Phong splatting. In *Proceedings Symposium on Point-Based Graphics*, Eurographics, 25–32.

BOTSCH, M., HORNUNG, A., ZWICKER, M., AND KOBBELT, L. 2005. High-quality surface splatting on today's GPUs. In *Proceedings Symposium on Point-Based Graphics*, Eurographics Association, –.

DODGSON, N. A. 2002. Analysis of the viewing zone of multi-view autostereoscopic displays. In *Proceedings of SPIE, vol. 4660*, The International Society for Optical Engineering, 254–265.

DODGSON, N. A. 2005. Autostereoscopic 3D displays. *IEEE Computer 38*, 8 (August), 31–36.

EBERT, D. S., SHAW, C. D., ZWA, A., AND STARR, C. 1996. Two-handed interactive stereoscopic visualization. In *Proceedings IEEE Visualization*, Computer Society Press, 205–210, 486.

GROSS, M. H., 2001. Are points the better graphics primitives? Computer Graphics Forum 20(3). Plenary Talk Eurographics 2001.

HE, T., AND KAUFMAN, A. 1996. Fast stereo volume rendering. In *Proceedings IEEE Visualization*, 49–56.

HILL, L., AND JACOBS, A. 2006. 3-D liquid crystal displays and their applications. In *Proceedings IEEE*, 575–590.

HODGES, L., AND MCALLISTER, D. F. 1993. *Stereo Computer Graphics and other True 3D Technologies*. Princeton University Press.

KOBBELT, L., AND BOTSCH, M. 2004. A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 6, 801–814.

KOO, Y.-M., LEE, C.-H., AND SHIN, Y.-G. 1999. Object-order template-based approach for stereoscopic volume rendering. *Journal of Visualization and Computer Animation 10*, 3 (July-September), 133–142.

LEE, Y.-G., AND RA, J. B. 2005. Reduction of the distortion due to non-ideal lens alignment. In *Proceedings of SPIE, vol. 5664*, The International Society for Optical Engineering, 506–516.

LEVOY, M., AND WHITTED, T. 1985. The use of points as display primitives. Tech. Rep. TR 85-022, Department of Computer Science, University of North Carolina at Chapel Hill.

LIU, G. H., WONG, Y. S., ZHANG, Y. F., AND LOH, H. T. 2002. Adaptive fairing of digitized point data with discrete curvature. *Computer Aided Design 32*, 4, 309–320.

MCALLISTER, D. 2002. *Ecyclopedia of Imaging Science and Technology*. John Wiley & Sons.

MINIEL, S., BRUNO, P., VATTA, F., AND INCHINGOLO, P. 2004. 3D functional and anatomical data visualization on auto-stereoscopic display. In *Proceedings EuroPACS-MIR*, 375–378.

MITRA, N. J., AND NGUYEN, A. 2003. Estimating surface normals in noisy point cloud data. In *Proceedings Symposium on Computational Geometry*, ACM, 322–328.

PAJAROLA, R., SAINZ, M., AND GUIDOTTI, P. 2004. Confetti: Object-space point blending and splatting. *IEEE Transactions on Visualization and Computer Graphics 10*, 5 (September-October), 598–608.

PAJAROLA, R. 2005. Stream-processing points. In *Proceedings IEEE Visualization*, Computer Society Press, 239–246.

PAULY, M., KEISER, R., KOBBELT, L., AND GROSS, M. 2003. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics 22*, 3, 641–650.

PFISTER, H., AND GROSS, M. 2004. Point-based computer graphics. *IEEE Computer Graphics and Applications 24*, 4 (July-August), 22–23.

PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: Surface elements as rendering primitives. In *Proceedings ACM SIGGRAPH*, ACM SIGGRAPH, 335–342.

PORTONI, L., PATAK, A., NOIRARD, P., GROSSETIE, J.-C., AND VAN BERKEL, C. 2000. Real-time auto-stereoscopic visualization of 3D medical images. In *Medical Imaging 2000: Image Display and Visualization*, S. K. Mun, Ed., vol. 3976 of *Proceedings SPIE*, SPIE, 37–44.

REN, L., PFISTER, H., AND ZWICKER, M. 2002. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Proceedings EUROGRAPHICS*, 461–470. also in Computer Graphics Forum 21(3).

RUSINKIEWICZ, S., HALL-HOLT, O., AND LEVOY, M. 2002. Real-time 3D model acquisition. *ACM Transactions on Graphics 21*, 3, 438–446.

SADLO, F., WEYRICH, T., PEIKERT, R., AND GROSS, M. H. 2005. A practical structured light acquisition system for point-based geometry and texture. In *Proceedings Symposium on Point-Based Graphics*, Eurographics Association, 89–98.

SAINZ, M., AND PAJAROLA, R. 2004. Point-based rendering techniques. *Computers & Graphics 28*, 6, 869–879.

SAINZ, M., PAJAROLA, R., AND LARIO, R. 2004. Points reloaded: Point-based rendering revisited. In *Proceedings Symposium on Point-Based Graphics*, Eurographics Association, 121–128.

SAINZ, M., PAJAROLA, R., SUSIN, A., AND MERCADE, A. 2004. A simple approach for point-based object capturing and rendering. *IEEE Computer Graphics & Applications 24*, 4 (July-August), 24–33.

SCHMIDT, A., AND GRASNICK, A. 2002. Multiviewpoint autostereoscopic dispays from 4D-Vision GmbH. In *Proceedings of SPIE, vol. 4660*, The International Society for Optical Engineering, 212–221.

SGI. Interactive Stereo Library (ISL). http://www.sgi.co.jp/solutions/visualization/isl/isl.pdf.

WAN, M., ZHANG, N., QU, H., AND KAUFMAN, A. E. 2004. Interactive stereoscopic rendering of volumetric environments. *IEEE Transaction on Visualization and Computer Graphics 10*, 1 (January-February), 15–28.

WEYRICH, T., PAULY, M., KEISER, R., HEINZLE, S., SCANDELLA, S., AND GROSS, M. 2004. Post-processing of scanned 3D surface data. In *Proceedings Symposium on Point-Based Graphics*, Eurographics, 85–94.

ZHANG, Y., AND PAJAROLA, R. 2006. GPU-accelerated transparent point-based rendering. In *ACM SIGGRAPH Sketches & Applications Catalogue*.

ZHANG, Y., AND PAJAROLA, R. 2006. Single-pass point rendering and transparent shading. In *Proceedings Symposium on Point-Based Graphics*, Eurographics Association.

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROSS, M. 2001. Surface splatting. In *Proceedings ACM SIGGRAPH*, ACM SIGGRAPH, 371–378.

ZWICKER, M., PAULY, M., KNOLL, O., AND GROSS, M. 2002. Pointshop 3D: An interactive system for point-based surface editing. In *Proceedings ACM SIGGRAPH*, ACM Press, 322–329.

ZWICKER, M., RÄSÄNEN, J., BOTSCH, M., DACHSBACHER, C., AND PAULY, M. 2004. Perspective accurate splatting. In *Proceedings of Graphics Interface*, 247–254.